Large-scale 3D geospatial processing made possible

Lucas C. Villa Real
lucasvr@br.ibm.com
IBM Research

Bruno Silva
sbruno@br.ibm.com
IBM Research

Dikran S Meliksetian
dikran_meliksetian@us.ibm.com
IBM

Kaique Sacchi
sacchikaique@gmail.com
University of Sao Paulo, IBM Research

Abstract

Several industries rely on accurate and efficient processing of 3D spatial queries over increasingly large datasets for decision optimization and exploration purposes. Examples include clinical diagnosis supported by 3D imaging of human tissues, numerical simulation of aerodynamics during the design of aircraft and vehicles, and the search for profitable deposits of minerals, oil and gas guided by 3D maps extrapolated from dense collections of rock samples. Despite the clear demand for spatial data processing in 3D space, existing systems supporting these organizations are scarce and scale poorly with data volume. This paper presents a GPU-based acceleration engine for SQL database management systems that can serve spatial queries orders of magnitude faster than solutions based on traditional software stacks.

CCS Concepts • **Information systems → DBMS engine architectures**; **Geographic information systems**; **Search engine architectures and scalability**; • **Hardware → Hardware accelerators**;

Keywords spatial databases, 3D geometries, hardware accelerators, GPU, GIS

ACM Reference Format:
Lucas C. Villa Real, Bruno Silva, Dikran S Meliksetian, and Kaique Sacchi. 2019. Large-scale 3D geospatial processing made possible. In *Proceedings of 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL '19).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3347146.3359351

1 Introduction

Database management systems play an important role in today's software architectures. They allow services to grow at scale and help users gain insights through the exploration of relations in large collections of structured, semi-structured and unstructured data. And, with a continuous demand for efficient database services, new implementations continue to arrive, exploring various methods of data storage and retrieval, hence pushing forward the limits of current systems [4, 31, 32, 37].

Several database systems include support for querying spatial data through custom data types and specialized predicate functions like intersection and distance tests between designated geometries – a vital feature for geo services such as navigation and mapping. Given the economic impact of the geospatial industry, estimated at more than USD 500 billion [8], and its effects in modern society, providing high performance spatial systems is of common interest to users and software providers alike.

Despite the clear demand for spatial data processing, the majority of the platforms only operate in the 2D space: MySQL (with MySQL Spatial Extension), IBM DB2 (with Spatial Extender), IBM Informix, Microsoft SQL Server and RethinkDB are examples in this realm. Relational databases supporting 3D geometrical entities are limited to Oracle Database (with its Oracle Spatial extension) and PostgreSQL (through the PostGIS extension). Regrettably, both systems present poor scalability and fail to meet demands when collections of large 3D objects are input [38]. As a consequence, the application of 3D spatial analysis supported by relational database management systems is limited to small datasets and meshes of moderate size.

This paper presents an accelerator for spatial queries in the 3D domain that improves the performance of spatial databases by several orders of magnitude. Code-named Lumic-GIS, this accelerator improves our previous work [27] by (i) employing geometry caching strategies, (ii) dynamically migrating geometries to GPU memory, (iii) specializing its algorithms to handle different geometry types efficiently, and (iv) scaling to several more CPU cores and GPU cards. We also introduce in this paper a synthetic dataset generator that we created to support our work. The tool creates realistic 3D features seen in the mining domain, which we use as a case study, easing the evaluation of spatial data processing platforms without depending on the availability of private data.

The sections of this paper are arranged as follows. We begin with an overview of spatial data types seen in the mining domain and the insights that geologists expect to gain from the analysis of such data. Then, we present the modeling of our synthetic workload generator. Sections 4 and 5 introduce, respectively, fundamental aspects of two pillars of our work: PostGIS and PostgreSQL. Next, Section 6 details the architecture of Lumic-GIS and its algorithms, followed by a performance evaluation in Section 7. We present related work in Section 8 and make closing remarks in Section 9.

2 Spatial Data in the Mining Domain

Visual insights and 3D geometrical modeling are key in the search for new ore deposits in the mining industry. In a typical workflow, drilling rigs – machines designated to drill subsurface rocks – extract several meter-long cylindrical core samples containing minerals from the mine. Chemical properties and minerals identified by visual inspection and by laboratorial analysis of each cylinder are

SIGSPATIAL '19, November 5–8, 2019, Chicago, IL, USA

ACM ISBN 978-1-4503-6909-1/19/11…$15.00
https://doi.org/10.1145/3347146.3359351

recorded in a database system that also includes spatial coordinates of the borehole and the drilling angle.

Domain-specific software that connect to this database system are able to reconstruct the drills in 3D space [13] and aid geologists to shape the spatial distribution of different minerals underground. Selected meshes, associated with zones of potential commercial value, are discretized into a block model using geostatistics and geological data gathered by the drilling process [24]. The output of the model is a collection of blocks of the same size but with different characteristics, as the ore density and grade that may be found in that location of the mine. Those blocks are then aggregated to represent entities such as geological faults and ore deposits.

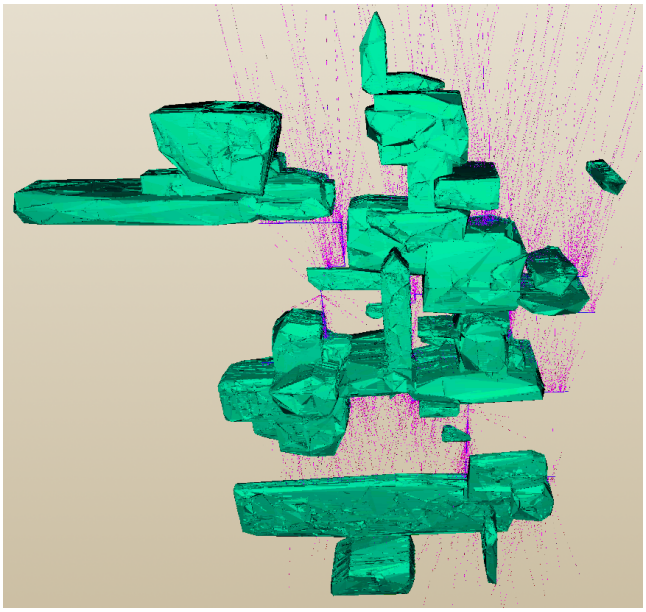

Figure 1. A synthetically generated mine plane, seen from above, featuring thousands of drill holes and complex geological shapes.

When properties related to 3D features are retrieved from the database system and combined with spatial operators capable of exploring them, geologists can submit complex queries to their systems, such as "give me the average ore grade of all block models in a zone no farther than 300 meters from this given drill hole". The problem is that even though there are off-the-shelf tools that enable users to query their spatial data like that, their performance do not allow users to explore data relationships in a timely fashion. A typical mine can feature several thousands of such spatial objects, as shown in Figure 1, and queries involving even small numbers of objects may demand several hours to complete on such systems.

3 Synthetic Mine Generation

Real-life 3D mine models may contain several thousand geometric representations of geological shapes, mineral blocks, drill holes, and mine tunnels. The utilization of real-life 3D data is very important to conduct performance experiments and draw realistic conclusions. However, the representation of these mines and the underlying information may represent strategic data to a mining company that cannot be shared. To overcome this issue, we created a synthetic mine generator[1] to enable scientists and mining practitioners to create and share mining data by using a set of configurable parameters. These parameters include:

- **Maximum/minimum number of floors.** A typical mine has a set of underground floors. On each floor, walls are drilled to learn information about the mineral compositions of that zone of the mine. We use this parameter to create a uniform probability distribution and generate the number of floors in a given synthetic mine.
- **Maximum/minimum number of geological shapes.** As in the previous parameter range, the maximum and minimum number of geological shapes are employed to create a uniform probability distribution and generate the total number of geological shapes for the whole mine. The number of shapes determined are distributed along the floors using an equal probability $P = 1/(\#floors)$.
- **Probability distribution for geological shapes dimensions.** By using the name and parameters for a theoretical probability distribution, users describe how geological shapes' dimensions (x, y, and z) are generated. For instance, dimension x may follow a normal distribution with a mean value of 1000 units and a standard deviation of 50. Our generator supports the distributions defined in Python's `scipy.stats` [15] package and the empirical probability distribution from the *AstroML* [36] package.
- **Number and size of drill holes.** The user defines the maximum and the minimum number of drill holes to create based on an uniform distribution. The sizes of drill holes are determined by a user-defined probability distribution function and its respective parameters.

3.1 Mine Creation Procedure

The recipe for the creation of individual features of a synthetic mine is given below. The number of features to be generated is based on user-provided intervals.

Floors: most underground mines feature several floors connected by elevators and access ramps. In synthetic scenarios with multiple levels, the elevator is represented by a random cell in a shared x,y coordinate that crosses all floors through the z axis. Next, on each floor, other random cells are generated and linked in a *dungeon* arrangement [12]. The connected cells are exported as a single polyhedral surface object.

Drill holes: we assume that the machinery drilling the rocks is placed next to the walls of the underground tunnels. Each drill hole is mapped to a cell on the grid of the current floor. The starting point of the drill is set to a random point on the surface of that cell's wall. The ending point is chosen by following the normal vector of that surface (so that the drill faces outward) and tilting the angle of that vector by a random value between $[-15, 15]$ degrees to make the drill look realistic.

Geological shapes: the point around which a geological shape is constructed is set to the ending point of a randomly chosen drill

[1] https://github.com/lucasvr/synthetic-mine-maker

hole. That point, or seed, expands on all three axes, placing unitary blocks along the visited paths, until the maximum growth size has been reached. The expansion begins by placing N_i blocks in the x direction. For every block generated on that axis, we let the geometry expand itself in the y and z directions by N_j and N_k blocks, respectively, for randomly chosen values of i, j, and k. The growth continues until the initial direction reaches its limit. Once the geometry has been assembled, the algorithm creates a shell around the cubes facing the exterior side of the polygon. The final geological shape is created by processing the convex hull of that shell, which elliminates jagged edges and gives the object a more natural appearance.

Block models: block models are produced by exporting the unitary blocks that formed the original geological shape.

3.2 Mine Characterization

In the current version of the synthetic mine generator, we use four parameters described by user-defined probability distribution functions (geological shape dimensions x, y, z, and drill hole sizes). To characterize these parameters, we employed *Fitter* [7] to fit theoretical probability distribution functions and *AstroML* [36] to fit empirical probability distributions.

The Fitter package can identify the most probable theoretical distribution function from which the mine data is generated among a set of more than 80 distributions. In order to verify the quality of the selected distribution, we use the Kolmogorov-Smirnov (KS) test under the null hypothesis that the generated data and real data are drawn from the same distribution. Depending on the characteristics of the mine data, we may not find a theoretical probability distribution that describes the provided mine data. In this case, we employ the empirical probability distribution from AstroML package.

4 Spatial Data Processing with PostGIS

PostGIS is a geospatial extension for PostgreSQL that serves as a pillar of our work. It provides methods to store spatial data in the database and to perform geometrical computation on that data. Unlike other popular databases, PostGIS has rich support for 3D geometries, including polyhedral surfaces and triangulated irregular network surfaces, and implements aggregation, proximity, intersection tests, and spatial indexes, among several other functions.

Many of PostGIS' queries can be processed in two-phases thanks to its support for geometry spatial indexes. This is the case with intersection tests, in which the requested operation is first performed with the minimum bounding boxes of the given geometries (which typically runs very fast). If there is no collision between the bounding boxes, then the second phase, which performs an exact computation on the actual geometries, can be safely skipped [21, 23].

Geometry blobs are saved in the main PostgreSQL storage area, next to columns that hold native data types such as text and integers. As long as a geometry fits in a single page size (commonly 8 kB), PostgreSQL is able to keep it in that area. For all other cases, the geometry is compressed, moved to a side-table, and the original field value is replaced by a pointer that instructs PostgreSQL where to find that data [34]. As we show in the performance results on Section 7, such a mechanism poses an impact on queries, because geometries have to be decompressed before processing begins. To work around this limitation it is possible to change the storage type of the geometry column to instruct PostgreSQL to move large geometries to the side-table *uncompressed*. The side effect, naturally, is a growth in the storage size for the database.

Besides being the de-facto GIS platform for PostgreSQL, PostGIS does not make good use of the computational resources when it comes to parallel processing of spatial queries. One of the problems is that the query planner may choose to delegate the query to a sub-optimal number of workers depending on how many rows might be generated by the execution path. Tables holding geometries of variable sizes may also lead to poor parallelism, as the cost of processing each tuple can vary significantly.

It is possible to cause the planner to execute a query with multiple workers by modifying the *execution cost* of the spatial function: an expensive function has more chances of enabling parallel processing than a cheap one. Still, that is not a guarantee that all available processors will process elements of that query. Further, such modifications may have an undesirable side-effect in function *inlining* decisions (i.e., whether or not to replace a function call with an incorporation of that function's body directly in the query): only inexpensive functions are candidates for inlining. This is a special concern for functions defined as a combination of two or more spatial operations (e.g., `ST_3DIntersects(geom1, geom2)`, defined as a spatial index scan on the bounding boxes followed by an exact computation on the actual geometries): the query planner may choose to "optimize" that function call (instead of inlining it) by replacing the index scan with a *sequential scan*, effectively causing exact computations to be carried out at all times [26].

The work we present in this paper aims at overcoming the aforementioned scalability problems by employing methods of function overloading and GPU kernels capable of distributing spatial workloads across thousands of dedicated processors. The next section gives more details about the infrastructure behind the SQL language that makes such improvements possible.

5 Foreign-Data Wrappers

The scope of the SQL language is published under ISO/IEC 9075. That standard defines fundamental data structures, operations on data stored in these structures, and minimum requirements of the language. Other parts of the standard define extensions to SQL. One example is SQL/MED, which adds support for managing external data through foreign-data wrappers and datalink types [20] in a federated architecture model.

Such an architecture allows a federated server to receive a SQL query and decompose it into fragments that can be executed elsewhere. The criteria for splitting the SQL query is chosen by the optimizer according to the estimated consumption of resources. Once the execution plan has been determined, fragments that operate on foreign tables are dispatched to the associated servers and remotely processed. The query results generated at each server are returned to the federated server, which merges them and forwards the result to the application that submitted the original query [16].

Lumic-GIS, our platform for the acceleration of spatial queries in the 3D domain, disguises as a PostgreSQL server that recognizes a subset of the protocol that PostgreSQL uses to communicate with foreign servers (implemented by its `postgres_fdw` module). Since installations of PostgreSQL include that module by default (including deployments of the database by cloud providers), users are not required to, e.g., install a custom module in order to connect

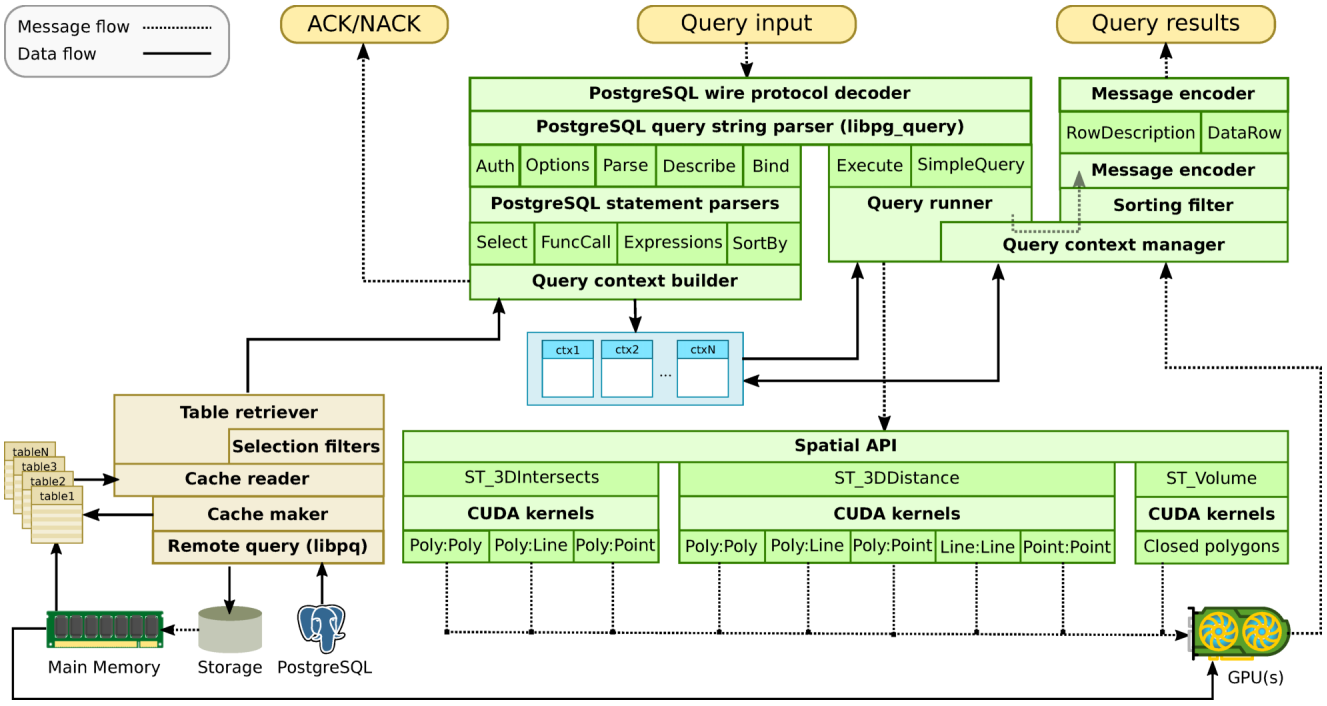

Figure 2. Architecture of our GPU-based accelerator. Query strings are parsed and converted into contexts that include information about (a) the kernels to execute, (b) the input tables and rows to process, and (c) the computed results.

an existing database to our accelerator. It suffices to configure a *foreign server* on the database that points to the IP address and port where Lumic-GIS listens for commands, and to declare the *foreign tables* exported by Lumic-GIS so they can be accessed remotely.

Despite the selection, modification, and removal of rows over remote tables, foreign-data wrappers allow the remote execution of function calls – as long as these functions are defined at both the source and destination servers. This requirement exists so that the federated server can compensate for any part of the query fragment that could not be executed by a foreign server. The standard, however, does not enforce both servers to have the same version of a given function's implementation. We exploit this fact in the design of Lumic-GIS: by configuring `postgres_fdw` to indicate the presence of a PostGIS API handler at the foreign server, query fragments involving PostGIS function calls are dispatched to Lumic-GIS, where they execute using hardware-accelerated algorithms. This design allows users to run spatial queries without having to learn details about the infrastructure supporting their execution.

Support for SQL/MED exists in other contemporary database management systems besides PostgreSQL, such as IBM DB2 [5] and MariaDB [19]. Although we chose PostgreSQL as the foundation for our work, it should be possible to extend it to other engines implementing SQL/MED and supporting spatial data types.

6 Speeding Up Spatial Databases

This section describes the software architecture of Lumic-GIS, including the design decisions and algorithms that enable our approach to scale to several thousands of processing units on modern GPU cards.

6.1 Software Architecture

The three components of Lumic-GIS, along with the flow of a spatial query, are shown in Figure 3: the application (also called the *client*), the PostgreSQL server, and the GPU-based query accelerator. Our acceleration platform hosts geometries in memory at all times in a format that can be readily ingested by the GPU kernels. Since Lumic-GIS is not meant to be a full-fledged relational database server, only two basic data types are allowed in its tables: *geometry* (which holds a representation of the in-memory geometry objects) and *integers* (which identifies them). Mappings between these tables and PostgreSQL are declared through the foreign tables mechanism.

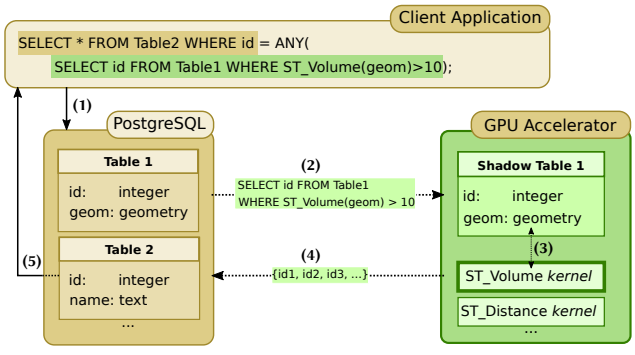

Figure 3. The three components of our solution. Query fragments with spatial components are dispatched to our acceleration engine, which processes them on high-performance GPU kernels.

On the sample query featured in Figure 3, the client requests a selection of all tuples from `Table 2` that have an associated geometry with a volume greater than 10 (*Step 1*). PostgreSQL identifies a reference to a foreign table (`Table 1`) and splits the original query string in two, subsequently forwarding the spatial query fragment to the GPU-based accelerator (*Step 2*). In *Step 3*, the volume kernel executes over all shapes from `Table 1`. The results are returned to PostgreSQL (*Step 4*), merged with the results from the other fragment that executed locally, and forwarded to the client (*Step 5*).

A more detailed picture of the software architecture that runs on top of the GPU hardware is shown in Figure 2. Geometries are initially loaded into main memory from local storage by the **cache reader**, or retrieved from a designated PostgreSQL server if the local cache does not exist or is out of sync with that server. For each `SELECT` query input to the system, the **context builder** stores the required operations and the associated geometries required by that query and stores them in a *session context*. Once the **query runner** receives a first request to fetch results[2], the actual computation is performed by one or more CUDA kernels under the **spatial APIs**. Finally, the query context is updated by the **query context manager** and the query results are returned to the client.

Note that although we present only three CUDA kernels in the paper, they can be used to compose more complex ones, such as 3D_MaxDistance, 3D_ClosestPoint, and testing if a geometry is between two other ones. Our platform currently features some of these, but they are not featured in Figure 2 due to space constraints.

Processing of spatial queries is computationally intensive. Lumic-GIS uses a combination of OpenMP [3] and CUDA *streams* [3] to efficiently exploit the computational resources available. On a server configuration based on P processors and M GPU cards, Lumic-GIS allocates $S = P/M$ CUDA Streams per GPU, each of which is associated with a unique OpenMP task during processing loops.

Differently from its predecessor, Lumic-GIS uses CUDA's unified memory management to allocate memory blobs and migrate data from host to device (and vice-versa). Thanks to this mechanism, GPU kernels can seamlessly access data from the CPU memory and process geometries larger than the GPU memory. Also, because memory allocations on the GPU cause a device-wide synchronization, they are considered very expensive. When processing a spatial query, Lumic-GIS identifies the largest geometries input to the query and allocates buffers that large. These buffers are reused for as long as possible throughout the lifespan of that operation. Memory management also plays an important role in the handling of new connections to the service. Whereas PostgreSQL spawns *processes* to handle them, Lumic-GIS creates *threads* so that memory buffers and partial results can be shared by similar queries. This plays a vital role in the performance of our system.

6.2 GPU Algorithm: Volume

The volume of a watertight 3D triangular mesh is computed on Lumic-GIS using the principle of the divergence theorem [11]. That theorem states that the dot product of a point and its respective normal for all faces are sufficient to evaluate the mesh volume.

As these attributes do not depend on information from another faces of the same mesh, this problem can be said to be *embarrassingly parallel*. The theorem is defined as

[2]Both full results and partial results (through server-side cursor) are supported.
[3]Queues of device work. The device schedules work from the streams when GPU resources are free.

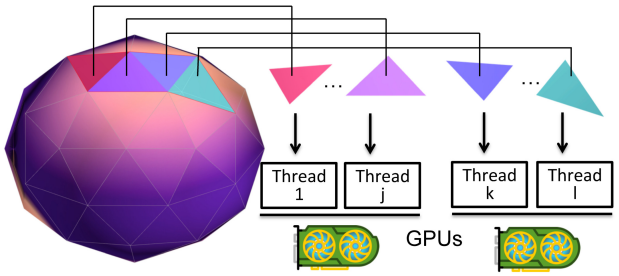

Figure 4. Evaluation of the volume of a 3D triangle meshes using GPUs. Each GPU thread computes the area and the normal of each triangle, according to the divergence theorem. The results are combined to compute the mesh volume. Other spatial operators in Lumic-GIS use a similar approach in that each GPU thread performs some basic computation on a solid face.

$$\iiint_V (\nabla \cdot \mathbf{F}) dV = \oiint_S (\mathbf{F} \cdot \mathbf{n}) dS, \tag{1}$$

where V denotes a volume in $\mathbb{R}^3$, S is the boundary of V ($S = \partial V$), $\mathbf{F}$ is a continuously differentiable vector field (flux) defined on a neighborhood of V, and $\mathbf{n}$ is the unit normal pointing outward S.

Let P be a polyhedron representing a database geometry given by a set of triangles A_i, $i = 0, \ldots, N-1$ with vertices (u_i, v_i, w_i). As all the faces are counter clockwise on A_i, the outer unit normal n to P on each face A_i is $n_i = \hat{n}_i \div |\hat{n}_i|$ where $\hat{n} = (v_i - u_i) \times (w_i - u_i)$. Therefore, if we assume a flux $F = \vec{p}/3$ where $\vec{p}$ is a point on the surface of P, then the volume (V) of P can be evaluated as

$$\begin{aligned} V &= \int_P 1 dP = \int_P (\nabla \cdot \vec{p}/3) dP = \frac{1}{3} \int_{\partial P} (\vec{p} \cdot n) d(\partial P) \\ &= \frac{1}{3} \sum_{i=0}^{N-1} \int_{A_i} (u_i \cdot n_i) dA_i = \frac{1}{6} \sum_{i=0}^{N-1} u_i \cdot \hat{n}_i, \end{aligned} \tag{2}$$

where we exploit the fact that the area of A_i is $|\hat{n}|/2$, and $\vec{p} \cdot n_i$ is constant over each A_i. Figure 4 presents the strategy to accelerate the volume calculation of 3D polyhedral meshes. Differently from a sequential CPU execution, GPU threads calculate $u_i \cdot \hat{n}_i$ for each face A_i in parallel. Depending on the number of faces of the solid, its volume can be computed in a single GPU parallel execution. The implementations of distance and intersection operators follow a similar approach as each GPU thread performs a basic computation on the solid's face.

6.3 GPU Algorithm: 3D Distance

The current implementation of 3D distance supports the following geometries: (i) line segment/line segment, (ii) line segment/polyhedral surface, and (iii) point/polyhedral surface. The following explanation is related to the distance between a line segment and a polyhedral surface; the other distance variations employ an analogous approach. We begin by findind the distance between each triangular face of the polyhedron and the line segment. Similarly to the volume calculation, the distance of each face and the line is calculated in a parallel GPU thread. Then, the minimum distance is returned to the user.

We employ the approach suggested by [9, 30] to evaluate the distance between a triangle and a line segment. Assume a line

segment with end points P_0 and P_1 with parametric representation $L(t) = P_0 + t\vec{d}$, $0 \leq t \leq 1$ where $\vec{d} = P_0 - P_1$. Let a triangle with vertices V_0, V_1, V_2 be represented as $T(u, v) = V_0 + u\vec{e_0} + v\vec{e_1}$ where $\vec{e_0} = V_1 - V_0$, $\vec{e_1} = V_2 - V_0$, $0 \leq u, v \leq 1$, and $u + v \leq 1$. The minimum distance is computed by locating the values u, v, t so that $T(u, v)$ is the triangle point closest to $L(t)$. We find u, v, t that minimizes the squared-distance $Q(u, v, t)$ (Equation 3) from T to L to find the minimum distance from L to each mesh triangle A_i.

$$Q(u, v, t) = |T(u, v) - L(t)|^2 \tag{3}$$

6.4 GPU Algorithm: 3D Intersection

The intersection supports the same geometries and utilizes the same face decomposition approach as the 3D distance. Whenever polyhedral meshes are involved in the computation, the operator decomposes the intersection evaluation of each polyhedral face in a GPU thread. We employ a less computationally-intensive evaluation for intersection when compared to the distance operator.

For instance, for the line segment and polyhedral surface intersection, each GPU thread intersects the line segment with the plane containing the given triangular face, and then determines whether or not the intersection point is within the triangle [9]. In this case, we represent the triangle with vertices V_0, V_1, and V_2 as $T(u, v, w) = uV_0 + vV_1 + wV_2$ where $w = 1 - u - v$, $0 \leq u, v \leq 1$, $u + v \leq 1$. The triple (u, v, w) is known as barycentric coordinates of T [30]. Assume also a line segment with points P_0 and P_1 and parametric representation $L(t) = P_0 + t\vec{d}$, $0 \leq t \leq 1$ where $\vec{d} = P_0 - P_1$. Then the values for t, u, and v should be

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(\vec{d} \times e_1) \cdot (e_0)} \begin{bmatrix} ((P - V_0) \times e_0) \cdot e_1 \\ (\vec{d} \times e_1) \cdot (P - V_0) \\ ((P - V_0) \times e_0) \cdot \vec{d} \end{bmatrix}, \tag{4}$$

where $e_0 = V_1 - V_0$ and $e_1 = V_2 - V_0$. If t, u, and v respect the aforementioned contraints, then L intersects T.

6.5 Precision Concerns

Although the 3D intersection equations are correct from the mathematical perspective, their execution on computers requires the utilization of floating point numbers, which may lead to approximation issues for very big or small numbers. Therefore, to check t, u, and v constraints from Equation 4, Lumic-GIS utilizes an ϵ epsilon constant to overcome precision pitfalls. For instance, to compare a given number a to 1, we use $|a - 1| < \epsilon$ instead of $a == 1$.

If we adopt a very small ϵ value, we may have some false negatives intersection results. On the other hand, for relatively large ϵ, false positives will occur more often. Figure 5 presents a possible precision problem in intersection calculation. Depending on the value of ϵ, if it is too low, the intersection may return false as the intersection occurs right on the borders of mesh triangles. This value is configurable; it is up to the user to decide which potential false conclusion has a greater impact on her application.

Lumic-GIS utilizes 3D intersection to check whether or not a point P is within an object B. We employ the crossing number method [10] which counts the number of intersections between an infinite ray starting from P and the surface of B. If the number of intersections is odd then P is inside B, otherwise, P is outside B. Sometimes the crossing number method fails to detect points inside or outside closed surfaces as infinite rays pass close to mesh triangles edges and the number of intersections is wrongly evaluated (e.g., Figure 5).

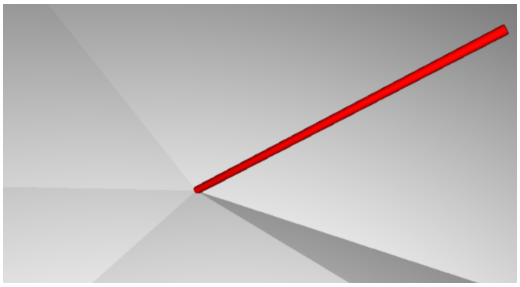

Figure 5. Possible intersection problem. For intersections tests very close to the borders of a geometry, an ϵ constant is required to overcome potential precision issues.

To overcome this issue, we extract two opposite rays R_1 and R_2 from P and count the number of intersections with B. There should be an odd number of intersections for both R_1 and R_2 with B to assume P is inside B. As the probability of having R_1 and R_2 passing through triangles edges like in Figure 5 is very low, we reduce drastically the chances of having misleading inclusion conclusions.

7 Performance Evaluation

7.1 Runtime Environment

Computing environment: all experiments were conducted on a bare metal machine with 32 Intel E5-2620 v4 processors, 768 GB of memory, 1.2 TB of SSD storage, and two NVIDIA Tesla V100 GPU cards. The software stack was based on the most recent development snapshots of PostgreSQL (version 12 beta1) and PostGIS (3.0.0 alpha1). The GPU kernels were built with the CUDA compiler version 10.0.130 and the host code with GCC 7.3.1. The PostgreSQL server and Lumic-GIS were hosted on that same machine.

Database setup: PostGIS spatial functions had their cost set to 100,000 to force parallel plans to be generated. The number of maximum parallel workers and the limit on parallel workers per gather were adapted according to the configuration being tested. We also configured the cost of non-sequentially fetched disk pages to 1 and improved the cache size to 50 GB.

Dataset: we generated a synthetic dataset [4] featuring 7,846 lines (representing complete drill holes) and 228,772 smaller line segments reflecting different intervals of these drill holes. The dataset also includes 71 polyhedral surfaces (geological shapes) and 78,051 blocks (block model). Because the number of patches (faces) of the geological shapes used as input to our benchmarks is variable, performance gains are non-linear as the number of geometries processed grows.

Queries: to evaluate the scalability of our GPU-based accelerator, the shapes were separated in groups of 10, 20, 30, 40, 50, 60, and 71 (all). Next, we ran spatial queries frequently performed by geologists: filtering drill holes based on their distance to profitable areas

[4] http://lucasvr.gobolinux.org/publications/2019-SIGSPATIAL-Dataset.zip

of a mine and other objects, and retrieving information from drill holes that intersect with certain objects of interest (such as geological shapes). All spatial functions are entered as *WHERE* clauses so that they can be properly forwarded through Postgres-FDW to the GPU accelerator. The queries are:

1. **3D Distance (lines × shapes):** executes the `ST_3DDistance` predicate over the geological shapes and all 7,846 drill holes;
2. **3D Distance (lines × lines):** computes the distance between the 7,846 drill holes and the 228,772 drill hole intervals;
3. **3D Distance (shapes × shapes):** runs `ST_3DDistance` over selected geological shapes;
4. **3D Intersection (shapes × lines):** computes the intersection between the same set of objects as above.

Speedup observed with the volume operator is the same as presented in our previous paper [27].

Measurement: we present the query time reported by the `psql` utility that ships with the PostgreSQL distribution. That measurement includes the time to retrieve geometries from the backing storage, the kernel execution times, and the time to assemble and send the reply back to the application – reflecting the total times that users have to wait when querying their data.

7.2 3D Distance (lines × shapes)

Unlike an intersection evaluation, processing of 3D distance queries does not benefit from spatial indexes: every pair of geometries needs to be processed and their distance have to be evaluated against any query filters before the results are sent back to the application. In this scenario, the difference between an off-the-shelf GIS system and a high-performance spatial engine is evident, as Figure 6 shows.

The largest set computed in this experiment involves the evaluation of the distance over 557,066 pairs of geometries. That is a relatively small number; geologists often run queries that result in several million computations. Yet, we observe that PostGIS, in its simplest configuration, takes 2h20min to compute and assemble the results. In the best case scenario, PostGIS is able to process the heaviest query in 51 minutes. Our GPU-accelerated platform, on the other hand, processes the same query in 11 seconds in its simplest configuration and in 7 seconds when two GPU cards are used. A graphical depiction of the query times with different GPU configurations on Lumic-GIS is shown in Figure 7.

On the executions with PostGIS, it is possible to see that the performance with 8 CPUs is close to that of 4 GPUs. This happens because **PostgreSQL planned and launched 5 workers only**, even though the platform's 32 processors were idle at the time of the query planning. PostgreSQL arrived at the same plan even when requested to allocate 32 workers.

With Lumic-GIS, one can notice an erratic pattern with the configuration based on 2 GPUs and 32 Streams: query times are not consistent, as the high standard deviation markers show. The reason for this behavior is that this configuration leads to the use of 64 OpenMP threads, which exceeds the number of processors available on that platform. The performance instability comes from the operating system's task scheduling overhead. The best configuration comes with 2 GPUs and 16 Streams, which maps to an optimal number of 32 OpenMP threads and that leads to an **improvement of more than 1300 times over PostGIS**.

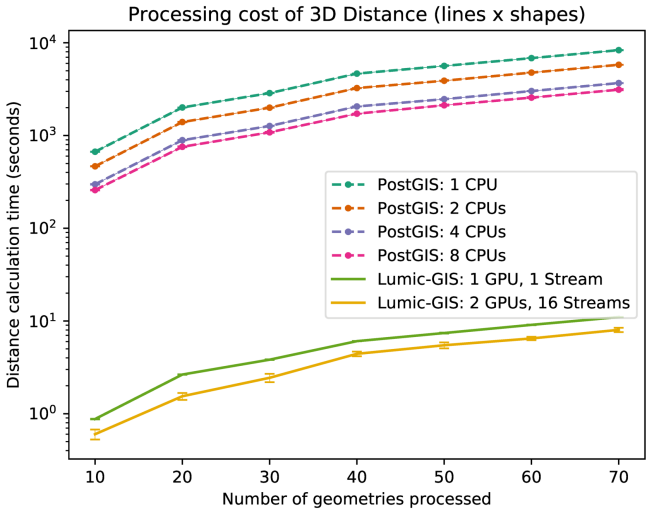

Figure 6. 3D distance query times with different number of geological shapes, shown in logarithmic scale. Our GPU-based acceleration engine improves over PostGIS by more than 1300×.

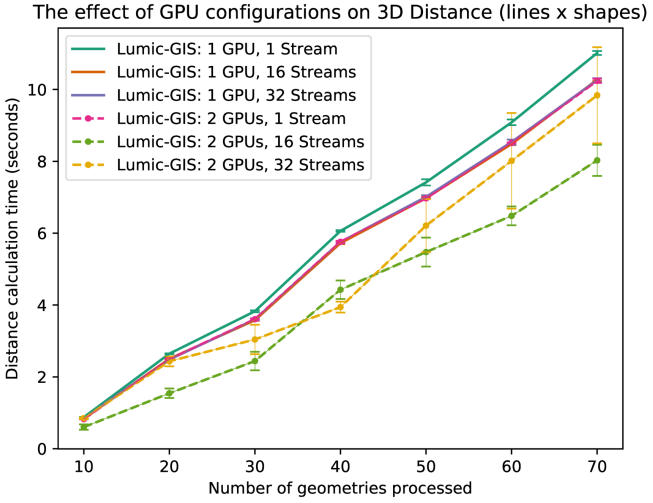

Figure 7. Impact of different GPU settings on 3D distance query times, shown in linear scale.

7.3 3D Distance (lines × lines)

Querying the distance between two line segments is often performed as a way to identify drill holes that are next to a drilled segment rich in minerals of interest. Moreover, because line geometries are less expensive to process than a collection of triangular meshes, some software resort to geometry simplification, collapsing triangles into lines for faster processing through approximation [28].

We picked the most interesting Lumic-GIS configurations from the previous test for this one. The results, presented in Table 1, show a lack of variation in execution times with PostGIS. The reason, observed in the query plan, is that PostgreSQL uses parallel workers to scan the drill holes tables, and joins the results with a call to `ST_3DDistance` – which runs on a single processor, despite being the dominant element in the query. The best recorded speedup of our GPU-based accelerator over PostGIS is of 61×.

Configuration	Query Time (sec)
PostGIS, 8 CPUs	718.90 ± 0.26
PostGIS, 4 CPUs	721.24 ± 2.24
PostGIS, 2 CPUs	721.31 ± 1.50
PostGIS, 1 CPU	719.14 ± 0.39
Lumic-GIS, 1 GPU / 1 Stream	13.91 ± 0.19
Lumic-GIS, 1 GPU / 32 Streams	13.56 ± 0.44
Lumic-GIS, 2 GPUs / 16 Streams	11.83 ± 0.23

Table 1. 3D Distance between pairs of lines

#Triangles		Query time		
Shape 1	Shape 2	PostGIS	Lumic-GIS	Speedup
1,936	2,270	9.2	0.05	159×
10,447	10,656	232 ± 1	0.45	515×
18,596	21,785	852 ± 20	1.44 ± 0.01	591×
49,052	51,095	5,212 ± 82	8.80 ± 0.24	592×
150,571	230,681	72,187 ± 451	123 ± 0.24	582×

Table 2. 3D Distance between two different geological shapes

7.4 3D Distance (shapes × shapes)

For this test, we hand-picked pairs of geometries with different numbers of triangles to explore the cost of processing their distance in the 3D domain. Since PostGIS' algorithms are not parallel (it is PostgreSQL that potentially spawns several workers to handle different geometries from the input), executing this query with 1, 2, 4, or 32 processors does not change its processing time. We compare PostGIS with an instance of Lumic-GIS running on a single GPU card and with 32 CUDA threads. The results are shown on Table 2.

Even though our dataset features geometries holding as many as 230,000 triangles, it is unfeasible to compute the distance between large shapes on PostGIS. As observed in the last row of the results table, processing the two largest geometries of our dataset takes more than **20 hours** on PostGIS, versus **2 minutes** on Lumic-GIS.

7.5 3D Intersection (lines × shapes)

When evaluating the intersection between a pair of geometries, most GIS systems employ a two-phase algorithm. In the first, they check if the minimum bounding box of the two geometries intersect. If they do, then the second phase performs a full intersection computation on the actual geometries. Both PostGIS and Lumic-GIS employ this technique, which lets them to work with significantly smaller geometry sets.

The query times of the best configurations of Lumic-GIS and PostGIS are shown in Figure 8. Here, we see the effect of PostgreSQL's function *inlining* decisions (discussed in section 4): when invoking `ST_3DIntersects`, a SQL wrapper around two spatial operations, **PostgreSQL generates query plans that involve just a single worker**. Consequently, a single CPU is used – which explains the similarity between the two PostGIS instances shown.

The difference between the PostGIS and our GPU-based platform is once again noticeable: **Lumic-GIS performs 250 times better**.

7.6 Compression Overhead on PostGIS

As presented in Section 4, PostGIS uses a side-table to store large geometries whose binary representation, in bytes, exceeds a page

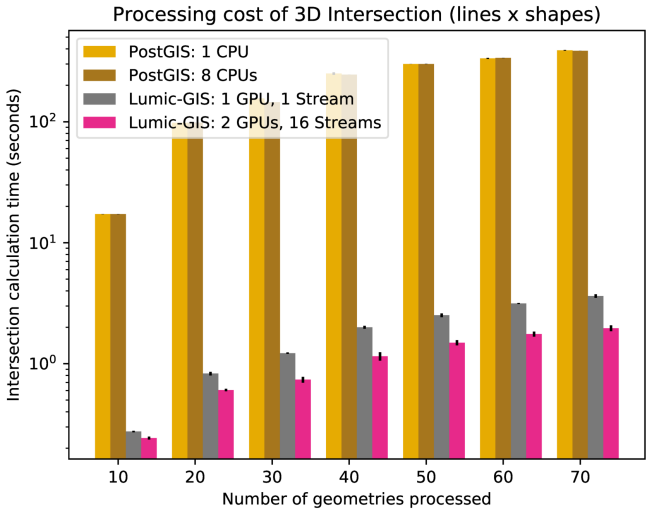

Figure 8. 3D intersection query times, shown in logarithmic scale. The observed speedup is of 250× over PostGIS.

size. Since the binary data is stored in compressed form, there is a run-time overhead to uncompress it prior to the computation of spatial queries with that data.

In this test, we configured PostgreSQL so that large geometries from our dataset are stored *uncompressed* in that area. Next, we ran the 3D distance tests once again (over lines and shapes) and captured the overheads associated with data preparation. The results are shown in Figure 9 and on Table 3.

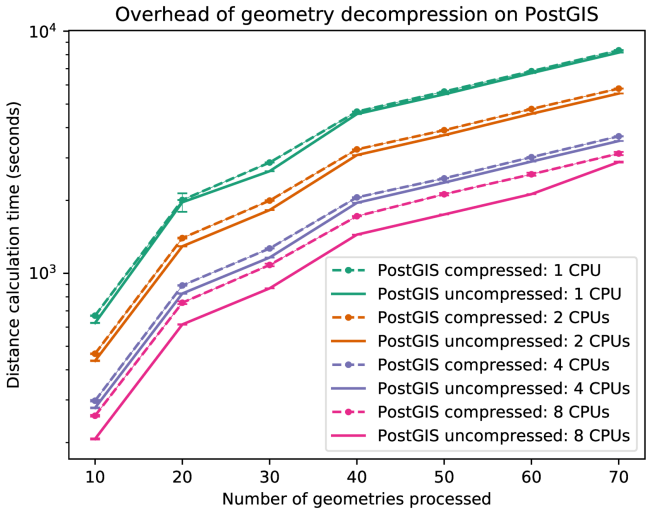

Figure 9. Overhead of geometry decompression of PostGIS, shown in logarithmic scale. In its simplest (1 CPU) and best (8 CPUs) configurations, the maximum decompression penalty is of 8% and 25%, respectively, as measured with the 3D Distance operator (lines × shapes).

Even in the absence of overheads associated with geometry decompression, the query times with PostGIS are significantly larger than what can be achieved with our GPU-based platform. In the best case, when processing 20 geological shapes, Lumic-GIS performs

Shapes	CPUs	Compressed	Uncompressed	Overhead
10	1	668 sec	624 sec	7.1%
20	1	2,009 sec	1,966 sec	2.1%
30	1	2,869 sec	2,642 sec	**8.6%**
40	1	4,656 sec	4,549 sec	2.3%
50	1	5,635 sec	5,490 sec	2.6%
60	1	6,839 sec	6,718 sec	1.8%
71	1	8,339 sec	8,164 sec	2.1%
10	8	258 sec	207 sec	24.6%
20	8	756 sec	615 sec	22.9%
30	8	1,083 sec	867 sec	**24.9%**
40	8	1,722 sec	1,445 sec	19.1%
50	8	2,121 sec	1,750 sec	21.1%
60	8	2,565 sec	2,124 sec	20.7%
71	8	3,125 sec	2,882 sec	8.4%

Table 3. PostgreSQL's compression effects on PostGIS performance. The *Compressed* column holds results for queries that retrieve geometries in a compressed format from the side-table, whereas *Uncompressed* holds results for a setup in which geometries are saved to that table in an uncompressed form.

1270× faster than PostGIS with a single CPU on a decompressed side-table. In its worse performance gains, Lumic-GIS processes 30 geological shapes 690× faster than PostGIS with the same setup.

7.7 Double-precision versus Single-precision

The engine of Lumic-GIS can be configured to work with single-precision IEEE floating points as opposed to double-precision ones. Despite known issues with the use of fewer bits for geometric computation [29] (e.g., rounding errors, false positives and negatives, imprecise numerical results), there are several applications for which a tradeoff between speed and accuracy is acceptable, as is being demonstrated by the field of approximate computing since the last decade [6, 14].

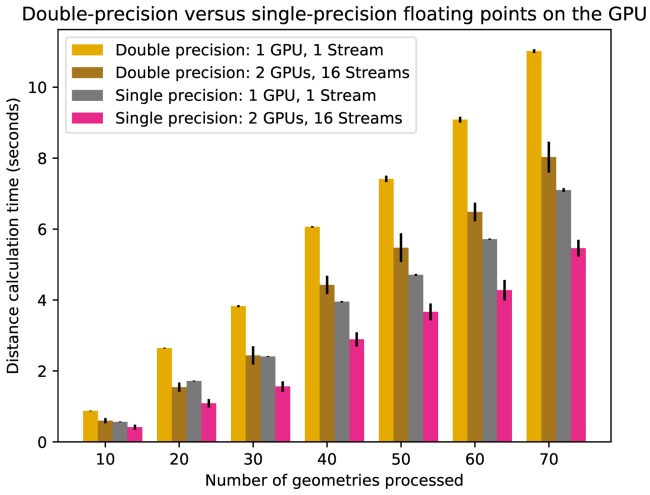

Figure 10. Performance improvements of single-precision floating points over double-precision on Lumic-GIS (shown in linear scale).

In this test we evaluate the difference in query times between the two floating point modes by benchmarking the 3D distance operation over lines and shapes. The results, shown in Figure 10, indicate processing times up to 59% faster when single-precision floats are used.

8 Related Work

Researchers have been looking into GPU acceleration for spatial query processing since hardware began supporting programmable vertex and fragment shaders. One of the first publications in this area comes from [33], who uses OpenGL to compute hardware-assisted intersection and distance tests of 2D geometries. The authors explore point, line, and polygon rasterization properties of OpenGL to guarantee accuracy in their hardware acceleration techniques. Especially, the strategy for testing the intersection of two 2D polygons is quite simple: polygons A and B are rendered with different colors C_1 and C_2 and, next, the framebuffer is searched for overlapping pixels with color $C_1 + C_2$. If such pixels exist, the two polygons intersect each other. Despite the slow times associated with framebuffer scanning, the authors achieved speedup of up to 4.8 times for intersection joins and up to 5.9 times for within-distance joins using a single NVIDIA GeForce 4 card.

In [17], the authors present Hadoop-GIS 3D, a 3D spatial query processing system built on top of Hadoop and programmed using the MapReduce programming model. Geometries are stored on Hadoop's Distributed File System (HDFS) and spatially partitioned into cuboids, which are then used for parallel computation of spatial queries. Spatial indexes are able to cover the entire geometry (called *global subspace indexes* and stored on HDFS as regular files that are close in size to the HDFS block size) and to operate at a finer grain, in which each cuboid has its own in-memory index that the platform generates on the fly. Another index supported by Hadoop-GIS 3D enables indexing of complex geometries that demand accurate geometrical computations. A data structure based on an R^*-tree stores the minimum bounding-box of each geometry and is used to quickly prune cuboids that do not satisfy 3D spatial join query conditions. The authors present a linear speedup on that kind of operation in a cluster comprising five nodes with 124 cores. However, there is no baseline comparison with another GIS platforms; even though PostGIS can be configured to use the same spatial library for 3D geometric computation as Hadoop-GIS 3D [22], the only speedups reported are of the author's own platform against itself under different configurations.

In [18], a revamped version of Hadoop-GIS 3D named iSPEED is presented. To achieve low latency, iSPEED maintains geometry data in memory and uses a progressive compression for each 3D object to have a minimal impact on memory footprint. The original geometry data remains stored on HDFS as in the previous generation of the platform. This new platform incorporates support for representing 3D data with multiple levels of detail so that users can choose between accuracy or fast query results. Performance results are compared against their previous work (Hadoop-GIS 3D), which shows an improvement of up to 6.6 times for 3D spatial join queries. The authors report a negligible cost of reading and parsing indexes stored in main memory and an impact of 2% to decompress geometries on-the-fly.

Many other studies on the acceleration of spatial operations are found in the 2D domain. Aghajarian and Prasad [1] use GPU hardware to improve the performance of grid-based spatial join algorithms. The idea they propose is to use non-uniform grids

to remove segments of a geometry that do not satisfy the join predicate. They do so by applying a minimal bounding box around the extent of each geometry and, in case of overlapping bounding boxes, partitioning their intersection area into equal-sized cells that are used for finer-grained filtering. They report a speedup of 225% over its predecessor GCMF [2] on spatial joins using a single-node, dual GPU system and more than 800-fold speedup over a naïve sequential implementation.

A different platform for efficient 2D spatial querying is presented in [25]. Instead of using a traditional approach based on the MapReduce paradigm or on relational databases, the authors propose a solution based on MPI, a high performance computing message-passing interface [35]. Dubbed MPI-Vector-IO, their work makes MPI aware of 2D spatial data types and introduces reduction operators for spatial primitives, effectively allowing the embedding of spatial data types within collective computation and communication. MPI-Vector-IO relies on parallel file systems (such as GPFS and Lustre) to efficiently parse and partition geometry data into cells of a local grid. MPI processes are given a subset of these cell-based tasks to compute the query. The authors report I/O and parsing improvements by one to two orders of magnitude, using up to 1152 CPU cores, over sequential operations on parallel file systems.

9 Conclusions

In several industries like mining, oil & gas, smart cities, and advertisement, the spatial location of entities plays a key role in the decision-making process. Current geospatial databases present solutions to process spatial queries such as distance and intersection for those problems. However, depending on the dataset size, large queries become infeasible as current platforms fail to return responses in a reasonable time. For several industries, it is not acceptable to wait several hours or days to get responses for a single spatial query.

Our work fills that gap by bringing a portable GPU-based platform to solve this issue and reducing the query time by several orders of magnitude (e.g., speedups of 1300×). We made large spatial queries feasible for several industries that had to limit their exploration scope to reduce response waiting time. Our platform is straightforward to use because no other language or skill is necessary than plain SQL which is already a requirement for users of spatial databases.

This paper also presented a synthetic generator of mine datasets to support our system evaluation. That tool, along with the dataset used in this study, has been made publicly available for the scientific community for future and related work.

Acknowledgments

The authors would like to thank Mark A. Smith and the anonymous referees for their detailed review and comments on this paper.

References

[1] Danial Aghajarian and Sushil K. Prasad. 2017. A Spatial Join Algorithm Based on a Non-uniform Grid Technique over GPGPU. In *SIGSPATIAL '17*. ACM, New York, NY, USA, Article 56, 4 pages.
[2] Danial Aghajarian, Satish Puri, and Sushil Prasad. 2016. GCMF: An Efficient End-to-end Spatial Join System over Large Polygonal Datasets on GPGPU Platform. In *SIGSPATIAL '16*. ACM, New York, NY, USA, Article 18, 10 pages.
[3] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. 2009. The Design of OpenMP Tasks. *IEEE Transactions on Parallel and Distributed Systems* 20, 3 (March 2009), 404–418.
[4] Peter Baumann, Dimitar Misev, Vlad Merticariu, Bang Pham Huu, and Brennan Bell. 2018. Rasdaman: Spatio-temporal Datacubes on Steroids. In *SIGSPATIAL '18*. ACM, New York, NY, USA, 604–607.
[5] M. J. Carey et al. 1995. Towards heterogeneous multimedia information systems: the Garlic approach. In *RIDE-DOM'95*. 124–131.
[6] C. Chen, J. Choi, K. Gopalakrishnan, V. Srinivasan, and S. Venkataramani. 2018. Exploiting approximate computing for deep learning acceleration. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 821–826.
[7] Thomas Cokelaer. 2014–. FITTER. https://github.com/cokelaer/fitter
[8] Anusuya Datta. 2016. *Where is the money in geospatial industry?* https://geospatialworld.net/article/where-is-the-money (visited on Sep 8, 2019).
[9] D.H. Eberly. 2007. *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*. Taylor & Francis.
[10] J.D. Foley, F.D. Van, A. Van Dam, S.K. Feiner, J.F. Hughes, J. Hughes, and E. Angel. 1996. *Computer Graphics: Principles and Practice*. Addison-Wesley.
[11] C. Gold. 2018. *Spatial Context: An Introduction to Fundamental Computer Algorithms for Spatial Analysis*. CRC Press.
[12] Nathan Hilliard, John Salis, and Hala ELAarag. 2017. Algorithms for Procedural Dungeon Generation. *J. Comput. Sci. Coll.* 33, 1 (Oct. 2017), 166–174.
[13] M Howson and EJ Sides. 1986. Borehole desurvey calculation. *Computers & Geosciences* 12, 1 (1986), 97–104.
[14] Mohsen Imani, Max Masich, Daniel Peroni, Pushen Wang, and Tajana Rosing. 2018. CANNA: Neural Network Acceleration Using Configurable Approximation on GPGPU. In *ASPDAC '18*. IEEE Press, Piscataway, NJ, USA, 682–689.
[15] Eric Jones, Travis Oliphant, Pearu Peterson, et al. 2001–. SciPy: Open source scientific tools for Python. http://www.scipy.org/
[16] Vanja Josifovski, Peter Schwarz, Laura Haas, and Eileen Lin. 2002. Garlic: A New Flavor of Federated Query Processing for DB2. In *SIGMOD '02*. ACM, New York, NY, USA, 524–532.
[17] Yanhui Liang, Hoang Vo, Ablimit Aji, Jun Kong, and Fusheng Wang. 2016. Scalable 3D Spatial Queries for Analytical Pathology Imaging with MapReduce. In *SIGSPATIAL '16*. ACM, New York, NY, USA, Article 52, 4 pages.
[18] Yanhui Liang, Hoang Vo, Jun Kong, and Fusheng Wang. 2017. iSPEED: An Efficient In-Memory Based Spatial Query System for Large-Scale 3D Data with Complex Structures. In *SIGSPATIAL '17*. ACM, New York, NY, USA, 17:1–17:10.
[19] MariaDB. 2013. *MariaDB CONNECT Storage Engine*. Technical Report.
[20] Jim Melton, Jan Eike Michels, Vanja Josifovski, Krishna Kulkarni, and Peter Schwarz. 2002. SQL/MED: A Status Report. *SIGMOD Rec.* 31, 3 (Sept. 2002), 9.
[21] Thanh Nguyen. 2009. Indexing PostGIS databases and spatial Query performance evaluations. 5 (09 2009), 1–9.
[22] Oslandia. [n.d.]. SFCGAL. http://www.sfcgal.org (visited on May 22, 2019).
[23] Neelabh Pant. 2015. *Performance comparison of spatial indexing structures for different query types*. The University of Texas at Arlington.
[24] Paulo Elias Carneiro Pereira et al. 2017. Geological modeling by an indicator kriging approach applied to a limestone deposit in Indiara city - Goiás. *REM - International Engineering Journal* 70 (09 2017), 331 – 337.
[25] Satish Puri, Anmol Paudel, and Sushil K. Prasad. 2018. MPI-Vector-IO: Parallel I/O and Partitioning for Geospatial Vector Data. In *ICPP 2018*. ACM, New York, NY, USA, Article 13, 11 pages.
[26] Paul Ramsey. [n.d.]. Patching Plain PostgreSQL for Parallel PostGIS Plans. https://carto.com/blog/postgres-parallel (visited on May 22, 2019).
[27] Lucas C. Villa Real and Bruno Silva. 2018. Full Speed Ahead: 3D Spatial Database Acceleration with GPUs. In *ADMS 2018*. 11–15.
[28] Jarek Rossignac. 1997. Simplification and Compression of 3D Scenes. In *Eurographics 1997 - Tutorials*. Eurographics Association.
[29] Stefan Schirra. 2000. *Robustness and Precision Issues in Geometric Computation*. 597–632. https://doi.org/10.1016/B978-044482537-7/50015-2
[30] P. Schneider and D.H. Eberly. 2002. *Geometric Tools for Computer Graphics*. Elsevier Science.
[31] ScyllaDB. 2018. *Applying Control Theory to Create a Self-Optimizing Database: The Technology Underpinnings of ScyllaDB's Autonomous NoSQL Database*. Technical Report. https://scylladb.com/resources/whitepapers (visited on May 22, 2019).
[32] Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman. 2011. The Architecture of SciDB. In *SSDBM'11*. Springer-Verlag, Berlin, Heidelberg, 1–16.
[33] Chengyu Sun, Divyakant Agrawal, and Amr El Abbadi. 2003. Hardware Acceleration for Spatial Selections and Joins. In *SIGMOD '03*. New York, NY, USA.
[34] PostgreSQL Development Team. [n.d.]. PostgreSQL Documentation - Database Physical Storage - TOAST. https://www.postgresql.org/docs/11/storage-toast.html (visited on May 22, 2019).
[35] CORPORATE The MPI Forum. 1993. MPI: A Message Passing Interface. In *Supercomputing '93*. ACM, New York, NY, USA, 878–883.
[36] J.T. Vanderplas, A.J. Connolly, Ž. Ivezić, and A. Gray. 2012. Introduction to astroML: Machine learning for astrophysics. In *Conference on Intelligent Data Understanding (CIDU)*. 47–54.
[37] Ramon Antonio Rodriges Zalipynis. 2018. ChronosDB: Distributed, File Based, Geospatial Array DBMS. *VLDB* 11, 10 (June 2018), 1247–1261.
[38] S. Zhang, J. Han, Z. Liu, K. Wang, and S. Feng. 2009. Spatial Queries Evaluation with MapReduce. In *2009 Eighth International Conference on Grid and Cooperative Computing*. 287–292.