Parallel Selectivity Estimation for Optimizing Multidimensional Spatial Join Processing on GPUs

Jianting Zhang
Dept. of Computer Science
City College of New York
New York City, NY, 10031
jzhang@cs.ccny.cuny.edu

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

Le Gruenwald
School of Computer Science
University of Oklahoma
Norman, OK 73071
ggruenwald@ou.edu

ABSTRACT

Managing large-scale data is typically memory intensive. The current generation of GPUs has much lower memory capacity than CPUs which is often a limiting factor in processing large data. It is desirable to reduce memory footprint in spatially joining large-scale datasets through query optimization. In this study, we present a technique of selectivity estimation for optimizing spatial join processing on GPUs. By seamlessly integrating multi-dimensional cumulative histograms and the summed-area-table algorithm, our technique can be efficiently realized on GPUs with good portability. Our experiments on spatially joining two sets of Minimum Bounding Boxes (MBBs) derived from real point and polygon data, each with about one million MBBs, have shown that computing the total numbers of MBB pairs at four grid levels took only about 3/4 second. By using the best grid resolution, our technique saves 38.4% memory for the spatial join. When histograms are materialized, it only took a few tens of milliseconds to search for the best grid level for the spatial join.

1. INTRODUCTION

Spatial data volumes are fast increasing due to advances of locating, sensing and simulation techniques. For example, although navigation devices (e.g. GPS, cellular and WIFI network-based, and, their combinations) embedded in smartphones (nearly 500 million sold in 2011 [1]) have already generated large volumes of location and trajectory data, the next generation of consumer electronics, such as Google Glasses, are likely to generate even larger volumes of location-dependent multimedia data. Objects identified from high-resolution satellite imagery and medical imagery, when represented as vectors of geometric coordinates, can also be considered as spatial data. In addition, large-scale climate, astronomical and molecular simulations are likely to produce even larger spatial datasets. Very often different spatial datasets need to be joined to derive new information and knowledge to support decision making. For example, GPS traces can be better interpreted when aligned with urban infrastructures, such as road networks and Point of Interests (POIs), through spatial joins. As spatial datasets are getting increasingly larger, techniques for high-performance spatial join processing on commodity and inexpensive parallel hardware become crucial in addressing the "BigData" challenge.

Spatial joins can be considered as extensions of relational theta joins [2] where spatial relationships, such as distance and topology, are involved in joining criteria [3]. While considerable research on join processing for both relational and spatial data have been reported, including those targeted for parallel computing platforms [2,3], there is little research on spatial join optimization on GPUs. Compared with multi-core CPUs, the current generations of GPUs typically have limited memory capacity, which frequently becomes a constraining factor for parallel spatial joins on large-scale spatial datasets. In addition, different from multi-core CPUs that are designed to support coarse-grained task-level parallelisms, fine-grained data parallelisms are crucial in achieving hardware potentials on GPUs. As such, many existing spatial join techniques that are either sequential in nature or rely on coarse-grained parallelisms cannot be efficiently applied to GPUs. The combined technical challenges in minimizing memory footprints and maximizing data parallelisms has motivated us to develop novel spatial join techniques on GPUs. In our previous studies, we have explored several GPU-based techniques for parallel spatial join processing, such as distance based point-to-polyline join [4], trajectory similarity join [5], and topology based point-in-polygon-test spatial join [6]. Our techniques adopt the classical two-phase spatial join framework, i.e., a filtering phase to pair MBBs followed by a refinement phase to evaluate the spatial relationships of geometric objects inside the MBBs [3]. While the refinement phase typically involves more floating point computation and is desirable to utilize GPUs for speeding up [4-6], we believe it is more technically challenging in improving the efficiency of the filtering phase on GPUs under stricter resource constraints, e.g., GPU memory capacity. Compared with the refinement phase that can relatively easily utilize batch processing to reduce resource requirements in a single batch, it is more difficult to explore a similar strategy for filtering as global information is typically required in the phase. Spatial filtering techniques that minimize memory consumption are thus preferred from an implementation and application perspective.

While spatial indexing techniques, such as pre-built quad-trees and R-Trees [7], have been frequently used to speed up spatial joins in classic computing models that are designed for serial algorithms, uniprocessors and disk-resident systems,

their suitability on GPUs for spatial joins needs to be reevaluated. First of all, very often their hierarchical tree structures and irregular memory access patterns incur significant performance penalty on parallel hardware, especially GPUs. Second, the complex data structures are expensive to construct and maintain (on both CPUs and GPUs) and difficult to manipulate (especially on GPUs). In this study, we aim at utilizing multi-dimensional histograms as light-weighted indices that are parallelization friendly to facilitate spatial join processing on GPUs. Different from heavy-weighted spatial index structures that are associated with data items for direct query processing, these histograms contain only essential statistical information to guide the choice of optimal/suitable parameters for spatial joins under resource constraints, with or without using spatial indices. Our technique is based on Cumulative Histogram (CD) in 2D space [8] to count the numbers of Minimum Bounding Boxes (MBBs) that intersect with cells in uniformly tessellated grids, compute the possible numbers of pairs in the grid cells, and choose an optimal grid level that satisfies GPU memory footprint budget. While CDs have been utilized in previous studies (e.g. [8]), we believe we are the first to take advantage of data parallelisms in constructing and utilizing CDs for parallel selectivity estimation and guide the optimization of spatial joins on GPUs. We provide a simple design and implementation that can be presented as a chain of sort-reduce-scatter-transform parallel primitives [18, 20]. These parallel primitives are well-supported across multiple parallel hardware platforms, including Nvidia and AMD GPUs.

The rest of the paper is arranged as the following. Section 2 introduces the background, motivation and related work. Section 3 presents the details of the parallel selectivity estimation technique. Section 4 presents the experiments and results. Finally, Section 5 is the conclusions and future work directions.

2. BACKGROUND, MOTIVATION AND RELATED WORK

Given two spatial datasets each with a geometric attribute the_geom, i.e., T1(id, the_geom) and T2(id, the_geom), the basic form of spatial join processing can be expressed as the following SQL statement:

```
SELECT * from T1, T2
WHERE ST_OP (T1.the_geom, T2 the_geom)
```

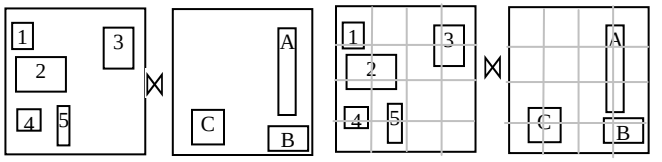
Here the geometric attributes in T1 and T2 can be any of the geometric types (e.g., point, polygon and polyline) and ST_OP can be any of the spatial relationships (e.g., intersect, within) defined by the Open Geospatial Consortium (OGC) Simple Feature Specification (SFS) [10] which has been the cornerstone of virtually all commercial (e.g., Oracle Spatial and Microsoft SQL Server Spatial) and open source (PostgreSQL/PostGIS) spatial databases. More complex queries may also involve additional attributes in T1 and T2, additional operators (e.g., count, sum) and additional clauses (e.g., *group by*, *having* and *order by*). Similar to theta joins in relational queries, spatial joins can be conceptually formulated as Cartesian products followed by evaluating spatial relationships between two geometric objects based on some well-established principles (e.g., nearest neighbor) and/or computational geometry algorithms (e.g., point-in-polygon test). Assuming the cardinality of T1 and T2 are n1 and n2, respectively, similar to processing relational joins, indices can be constructed to reduce the complexity from O(n1*n2) to O(n1) or O(n2) provided that a good spatial filtering strategy is available so that a spatial object in T1 will only be paired with a limited number of spatial objects in T2. As argued in [3], spatial joins are distinguished from relational joins due to the fact that spatial data are inherently multi-dimensional data that exhibits several unique features, e.g., lacking ordering that preserves proximity (which makes sort-merge join largely inapplicable), unsuitable for grouping due to having extents (which makes equijoin inapplicable), and, requiring complex geometric computation (which is typically much more expensive than arithmetic operations).

Hundreds of indexing structures have been developed in the past few decades to index and query spatial data [7]. In addition, we refer to the excellent survey paper [3] for a comprehensive review on spatial join techniques, including several parallel spatial join techniques on traditional cluster computing environments. We also refer to several recent works on spatial join processing on MapReduce/Hadoop clusters [11,12] with demonstrated scalability at the expenses of singe node efficiency. Despite that shared-memory systems are getting increasingly popular and affordable in both personal and cluster computing settings and typically are easy to program, almost all existing parallel spatial join techniques are designed for shared-nothing architectures. As GPUs that are capable of general computing typically have large numbers of processors (10^2-10^3), much higher bandwidth (~100 GB/s vs. ~100 MB/s) and more floating point computing power (by design), an alternative to cluster computing (including Hadoop clusters running MapReduce jobs) in solving moderate sized spatial join problems on single GPU devices becomes promising. We note that as GPUs are typically used as accelerators in computing nodes, it is quite possible to integrate the two sets of techniques to solve larger scale spatial join problems when needed.

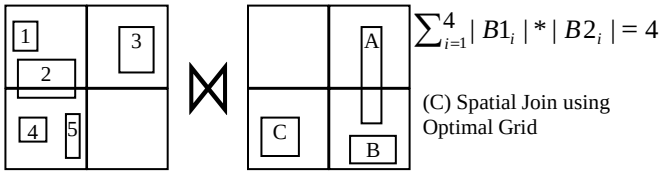
As detailed in [3], spatial joins can be performed on two spatial datasets that both, one or none of them have indices. Although using pre-built indices typically can significantly boost the performance of spatial joins on CPUs, we argue that traditional tree-based hierarchical indices are less effective to be ported to GPUs for parallel execution directly. While more and more programming language constructs are increasingly available for GPU computing, e.g., recursions, pointers and dynamic memory allocations/deallocations, which makes porting serial code to GPUs easier, they can bring significant performance penalty if applied inappropriately. Naïve GPU implementations can perform even worse than serial CPU implementations. Furthermore, we strongly believe that identifying the inherent parallelisms in spatial joins, which are likely to sustain several hardware generations, is more important than optimizing a particular design for a specific hardware platform. As such, instead of porting existing popular tree-based spatial indexing techniques to GPUs to speed up spatial join processing, *our focus in this research is to develop novel parallel techniques that can make full use of GPU hardware computing power under memory capacity constraints*. The research is in parallel with several existing research efforts on indexing and querying multi-dimensional spatial data on GPUs [13,14,15].

Selectivity estimation is considered a vital component in query optimization in both relational databases and spatial databases. Given a set of query items in T1, selectivity estimation techniques estimate the numbers of items in T2 that are likely to be joined with each of the query items. Fast and accurate selectivity estimations can help database query optimizers to choose better query plans under resource constraints. A slightly different problem that provides summary information of query results (e.g., counts) for a single query or a set of queries without actually querying database records essentially require a same set of techniques as selectivity estimation. These typically can be achieved by maintaining data structures of essential statistics. Several techniques on selectivity estimation for spatial joins and query processing have been reported (see Appendix 2 for a brief review and a list of related publications), but none of them has been researched in parallel computing settings.

Among these techniques, we are particularly interested in works that are based on regularly spaced multi-dimensional histograms. This is because when the histogram bins use the same configuration for both datasets involved in a spatial join, the total number of pairs to be processed in the refinement phase in each bin is $s=\Sigma|B1_i|*|B2_i|$ where $|B1_i|$ and $|B2_i|$ are the numbers of geometric objects that intersect with spatial extents of the common bins in T1 and T2, respectively. However, as shown in the example in Fig. 1, bin sizes of such 2D histograms (typically the same as grids that used for indexing/querying) play a very important role in determining the memory footprint of spatial filtering. As our GPU based spatial join framework (Appendix 1) requires pairing all the MBRs from both datasets, very often a large number of pairs, need to be output before unique pairs can be computed and used in the refinement phase. Although the number of unique pairs might be small, the number of intermediate pairs can be too large to be fit in GPU memory. On the other hand, if a small cell size is chosen, while $|B1_i|$ and $|B2_i|$ are likely to be smaller, *N* usually grows quadratically which may also incur large numbers of intermediate pairs. For the example shown in Fig. 1, the grid at the bottom (Fig. 1C) is most memory efficient where the number of candidate pairs (4) is significantly smaller than using a coarser grid (Fig. 1A) or finer grid (Fig. 1B). This is also the primary motivation of our proposed technique.



(A) Spatial Join using Coarser Grid (B) Spatial Join using Finer Grid



(C) Spatial Join using Optimal Grid

Fig. 1 Illustration of Choosing Optimal Grid Level in Minimizing Memory Footprint for Spatial Filtering

By observing that generating a 1D cumulative histogram is equivalent to performing a scan (prefix-sum [18]) on the original regular histogram, and generating a 2D cumulative histogram can be realized using two scans on the original 2D regular histogram using both row-major order and transposed row-major order (to be detailed in Section 3), we have developed a simple yet effective parallel approach to derive $|B1_i|$ and $|B2_i|$ counts for all bins from the two sets of MBBs in a spatial join and use them to choose the appropriate grid level under GPU memory constraints. The technique will be presented in details in Section 3.

As more recent GPUs increasingly support unified memory addressing on both CPUs and GPUs, it becomes possible to use CPU memory and/or external memories to virtually increase GPU memory to avoid GPU memory capacity limit. While this may be promising for certain applications, such as certain relational joins as reported in [16,17] where slow data movement across GPU/CPU/disk boundaries can be hidden by computing, it may not be efficient for spatial filtering based on our current framework for spatial join processing (Appendix 1). This is because sort/search/unique primitives in spatial filtering are largely bounded by memory bandwidth and do not exhibit blocked memory access patterns unless specially designed.

3. PARALLEL SELECTIVITY ESTIMATION ON GPUs

The parallel selectivity estimation algorithm presented in this section is an important component in our GPU based parallel spatial join framework using regularly spaced grid file structure [7,14] for spatial filtering (see Appendix 1 for details). The algorithm aims at estimating the total numbers of pairs of MBBs that will be generated during the spatial filtering phase at multiple grid levels and guide the query optimizer to choose an optimal grid level that minimizes GPU memory footprint during spatial filtering.

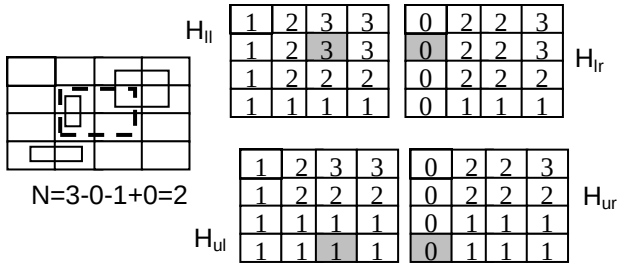


Fig. 2 Illustration of Spatial Cumulative Histogramming and Selectivity Estimation for a Single Query Window

Four 2D cumulative histograms (H_{ll}, H_{lr}, H_{ul} and H_{ur}) are required to store the numbers of MBBs whose lower-left, lower-right, upper-left and upper-right corners fall within grid cells [8]. Assuming the bin values of a regular histogram are m_i, then the corresponding bin values of a cumulative histogram are $m_k = \sum_{i=1}^{k} m_i$. Given a query window $W(x_1,y_1,x_2,y_2)$ where x_1, y_1, x_2 and y_2 are given as grid coordinates, the number of MBBs that intersect with W can be calculated as $N= H_{ll}(x_2,y_2) - H_{lr}(x_1-1,y_2) - H_{ul}(x_2,y_1-1) + H_{ur}(x_1-1,y_1-1)$ (we refer to [8] for the derivation). As an example shown in Fig. 2, the number of MBBs that intersect with the query window shown as a dashed

rectangle is N=2 by accessing the four highlighted bins in the four cumulative histograms. While the approach allows arbitrarily shaped rectangles, we set x1=x2 and y1=y2 so that N will be the number of MBBs that intersect with the grid cell (histogram bin) at (x1,y1). It is easy to see that the computation has perfect data parallelism on GPUs.

To derive each of the four cumulative histograms for all grid cells, we first re-use the design and implementation of point aggregation approach in our previous work [4,19] to compute regular 2D histograms (B) where corner points are transformed to cell identifiers by applying a *transform* parallel primitive. A *reduce (by key)* parallel primitive is then applied to count the number of corner points that fall within each grid cell/bin. As the grids/histograms might be sparse, the counts need to be scattered to the respective histogram bins by applying a *scatter* parallel primitive. The second step in computing a cumulative histogram is to apply the parallel summed-area-table algorithm outlined in [9] that includes four sub-steps: applying an *(inclusive) scan* primitive on row-majored grid cells (assuming the result is B'), *transpose* B' to B'', applying the same *(inclusive) scan* primitive on B'' to derive B''' before finally *transpose* B''' back to the original row-major order to derive H. The complete algorithm is provided in Fig. 3. Note that we use upper case variables to represent vectors and lower case variables to represent scalars. In case we need to refer to individual vector elements, we put indices as superscripts on the corresponding vectors. In the presentation, parallel primitives that are used in the algorithms are both bolded and italicized. As these parallel primitives are either directly supported by CUDA SDK (through its Thrust library [20]) or have been widely implemented on GPUs (e.g., *transpose*), we will not further explain the implementations of these functions and we refer to the interested readers to the Thrust library documentation. To better illustrate algorithms *compute_counts* and *gen_sat*, an example using the same data as in Fig. 2 is shown in Fig. 4. The example calculates H_{ll} using algorithm *gen_sat* and calculates the N grid in Step 2 of algorithm *compute_counts* after all the four cumulative histograms have been derived.

Algorithm selectivity_estimation
Inputs: MBB sets *M1* and *M2*
Outputs: optimal grid resolution r_o

For each candidate resolution r_k
Step 1 call compute_counts(*M1*, r_k, *N1*)
Step 2 call compute_counts(*M2*, r_k, *N2*)
Step 3: *N3*←***transform(****N1*,*N2*) where $N3^i=N1^i*N2^i$
Step 4: s_k=***reduce(****N3*)
Step 5: if s_k exceeds memory budget then break
Return r_o that corresponds to the smallest s_k

Algorithm compute_counts
Inputs: MBB set *M* and resolution *r*

Outputs: grid *N* representing the numbers of MBBs intersect with each grid cell
Step 1 For each of *H* in {H_{ll}, H_{lr}, H_{ul} and H_{ur}}
Step 1.1 *V*←***Transform(****M*) where V^i={lower-left, lower-right, upper-right, upper-right} corner coordinates of M^i
Step 1.2 Call point_aggregation(*V*,*B*,*r*)
Step 1.3 Call gen_sat(*B*,*H*)
Step 2 ***Transform*** on H_{ll}, H_{lr}, H_{ul} and H_{ur} and put the result in *N* where $N^i = H_{ll}(x_2,y_2) - H_{lr}(x_1-1,y_2) - H_{ul}(x_2,y_1-1) + H_{ur}(x_1-1,y_1-1)$ and $x_1=x_2=i\%c$ and $y_1=y_2=i/c$ ($c=2^k$)

Algorithm gen_sat
Inputs: Grid *B*
Outputs: summed area table *H*

Step 1 *H*←***inclusive_scan_by_key(****B*) using row identifiers as keys
Step 2: *H*←***transpose***(*H*)
Step 3: *H*←***inclusive_scan_by_key***(*H*) using row identifiers as keys
Step 4: *H*←***transpose***(*H*)

Algorithm point_aggregation
Inputs: Point set *V* in the form of (*x*,*y*) pairs and grid resolution *r*;
Coordinate system origin x_0 and y_0 (global variables)
Outputs: Grid *B* representing the numbers of corner points

Step 1 ***transform*** *V* to generate cell identifiers $C^i=(y-y_0)/r*COL+(x-x_0)/r$
Step 2 ***sort*** *C*
Step 3***: reduce*** **C** ***(by key)***, count the numbers of keys (*K*,*D*)←reduce_by_key(*C*)
Step 4: **scatter** *D* to *B* according to *K* where $(row,col)^i \leftarrow K^i$

Fig. 3 Algorithms for Parallel Selectivity Estimation

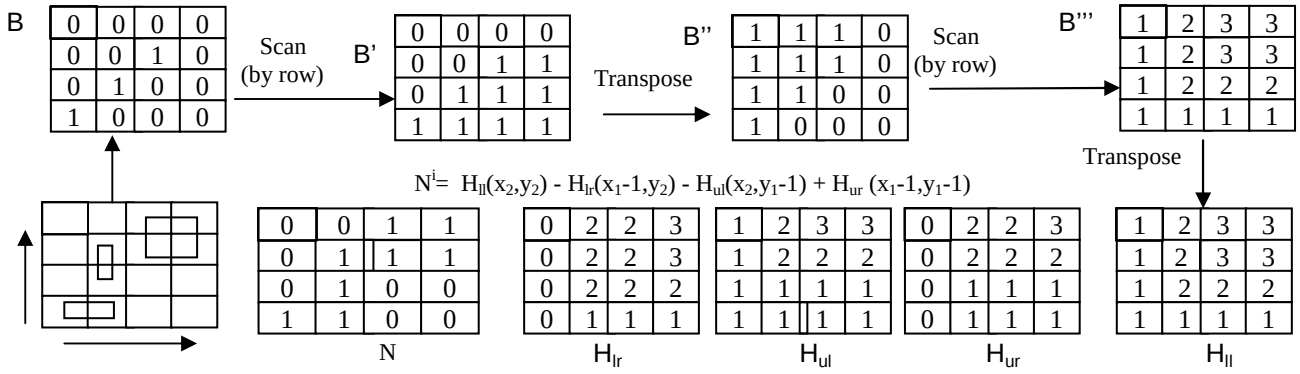


Fig. 4 An example to illustrate algorithms compute_counts and gen_sat

We next provide a brief time and space analysis of the proposed technique. For algorithm *point_aggregation*, assume the number of grid cells is Ng and the number of points is Np, then the time complexity is O(Np) for Step1, O(Np) for Step 2 using radix sort (which is the case for our GPU implementation), max(O(Np),O(Ng)) for Step 3, and O(Ng) for Step 4. As such, the total time complexity of algorithm *point_aggregation* is O(Np)+O(Ng). The time complexity for the algorithm *gen_sat* is O(Ng) for all the four steps. As the time complexity for Step 1.1 and Step 2 of algorithm *compute_counts* are O(Np) and O(Ng), respectively, the total time complexity of the algorithm is 4*(O(Np)+O(Np)+O(Ng)+4*O(Ng))+O(Ng)=O(Np)+O(Ng). Assuming that the numbers of MBBs in T1 and T2 are |M1|=U1 and |M2|=U2 and the number of grid cells (histogram bins) at the grid level r_k is Wk, by substituting Np with U1/U2 and substituting Ng with Wk, the time complexity of the first two steps in algorithm *selectivity_estimation* is

$$O(U1)+O(Wk)+O(U2)+O(Wk)=O(U1)+O(U2)+O(Wk).$$

As both Step 3 and Step 4 in algorithm *selectivity_estimation* have a time complexity of O(Wk) and Step 5 has a time complexity of O(1), the total time complexity for grid level r_k is thus

$$O(U1)+O(U2)+O(Wk)+O(Wk)+O(Wk)+O(1) =O(U1)+O(U2)+O(Wk)$$

As k is limited to a small number in practice, the final time complexity of the algorithm is linear with respect to U1, U2 and max(Wk). With respect to space complexity, while conceptually memory storage for M1, M2, N1, N2, N3, H_{ll}, H_{lr}, $H_{ul,}H_{ur,}$ C and K is required at the same time, we note that optimizations to consolidate among N1/N2/N3 and $H_{ll}/H_l/H_{ul}/H_{ur}$ might be possible through GPU kernel fusion/fission [21]. As U1 and U2 are typically in the order of millions and Wk is typically between W_{10}=1024*1024 and W_{13}=8192*8192, the proposed approach can be applied to GPU devices with a few tens of megabytes memory capacity, which can be easily satisfied.

Despite the linear time and space complexity of the proposed approach, we note that the two parallel primitives (*scan* and *transpose*) in the *gen_sat* algorithm are invoked eight times each for the two input MBB sets. Similarly, the four parallel primitives (*transform*, *sort*, *reduce* and *scatter*) in the *point_aggregation* algorithm are invoked four times. As such, the efficiency of the implementations of these parallel primitives is crucial for the overall efficiency of the proposed technique. In our implementation, we implement the *transpose* primitive using CUDA and shared memory is used to further improve the efficiency. For the rest of the parallel primitives, we use those provided by the Thrust library although we believe there are still rooms to further improve their efficiency (e.g., as demonstrated by the MGPU library [22]). The improvements are left for our future work.

4. EXPERIMENTS AND RESULTS

To validate our design and implementation, we use two real datasets in our experiments. The first dataset contains 168 million taxi trip records each with a pick-up and drop-off location. We generate quadrants from the point dataset by setting the maximum number of points in each quadrant to K=1024 points [6]. We use the MBBs of the points in the quadrants. We call the first MBB set as Taxi and the number of MBBs in the set U1=|Taxi|= 990,142. The second dataset to participate in the spatial join is the NYC MapPluto tax lot data [23] with 735,488 polygons and 4,698,986 vertices. We use the MBBs of the polygons as our second MBB set, i.e., U2=|MapPluto|= 735,488. We use four grid levels, i.e., k varies from 10 to 13 and grid size varies from 1024*1024 to 8192*8192 for both selectivity estimation and spatial filtering. All experiments are performed on an Nvidia Quadro 6000 GPU device with 448 cores and 6 GB memory.

Table 1 lists the computed numbers of pairs of MBBs (s_k) and query estimation times (Ts) at the four levels. For grid level 10, there are 86 million pairs. The two data vectors that store the pairs require nearly 700 megabytes of GPU memory (4 bytes for each of the two identifiers in a pair). The spatial filtering module may fail on certain GPU devices due to memory capacity limit. On the other hand, if level k=12 is chosen, the memory footprint will be reduced by more than half which clearly demonstrates the importance of query optimization. Assuming that the chance of picking all grid levels for spatial filtering is the same, then the expected number of estimated pairs is AvgN=$\Sigma N_i/4$. After applying the selectivity estimation algorithm, we are able to pick the grid level that incurs the minimum number of pairs minN=min(N_i). As such, the benefit is (AvgN-minN)/AvgN=34.8%. The total cost of the optimization is simply the total runtimes ΣTs_i=744 ms. In other words, we are able to reduce the memory footprint of the spatial join by 34.8% in 744 ms, which is desirable in many cases.

Table 1 Memory Footprints and Runtimes for Selectivity Estimation and Spatial Filtering at Multiple Grid Levels

Grid Level k	Grid Size	# of Estimated Pairs (N)	Ts (ms)	Tf (ms)
13	8192*8192	78,328,554	496	1090
12	4096*4096	40,414,590	146	432
11	2048*2048	43,121,125	62	332
10	1024*1024	86,103,593	39	525

For comparison purposes, we also list the spatial filtering runtimes (Tf) among the two datasets (without using the selectivity estimation module) in the last column of Table 1. We can see that Ts is significantly lower than Tf at all grid levels, especially for grids with lower sizes, e.g., 1024*1024 when k=10. The observation that Ts grows superlinearly with grid level can be explained by the fact that Ts is a combination of the cost that is linear with Np (U1/U2) and the cost that is linear with Ng (W_k) but quadratic with grid resolution (2^k), based on the cost model presented in Section 4. In contrast, although the details were skipped in Section 3, Tf is a combination of the cost that is linear with U1/U2 and the cost that is linear with N. Different from cumulative histograms that require using 2D grids in selectivity estimation, vectors representing sparse 2D grids are used in spatial filtering which is more efficient. As the grid size gets higher, selectivity estimation becomes more costly and the performance advantage of selectivity estimation over complete spatial filtering decreases. It is possible that the ΣTs_i can be larger than $Tf_{k.}$ at a certain point. Selectivity estimation overhead could become a significant portion of the end-to-end

spatial filtering runtime when the selectivity estimation module is included. Since we increase grid level (k) gradually, we can include the projected spatial filtering cost into the stop criteria in Step 5 of algorithm *selectivity_estimation* in Fig. 3 in addition to memory budget. Given that using high grid resolution will increase selectivity estimation times superlinearly and may not always be able to improve memory consumption for spatial filtering, we recommend stop increasing k as soon as ΣTs_i reaches a time budget limit (e.g., 100-500 ms).

By observing that N1 and N2 histograms (grids) that are used to compute $_{Sk}$ are only related to their respective MBB sets and do not depend on each other, a viable solution to improve selectivity performance is to use prebuilt N1 and N2 grids to compute N3 and s_k. As such, algorithm *selectivity_estimation* needs only to compute N3 on-the-fly (Step 3) before reducing N3 to calculate s_k (Step 4). These two steps are extremely fast on GPUs as they are embarrassingly parallelizable and can make full use of GPU hardware resources. Our experiments have shown that the runtimes vary from about 3 ms for the 1024*1024 grid size (k=10) to about 10 ms for the 8192*8192 grid size (k=13) which is fast enough for most spatial datasets. However, as the prebuilt N1 and N3 grids need to be streamed from hard drives to CPU memories and then to GPU memories, I/O overheads can be significant. For an 8192*8192 integer grid, the storage requirement is about 256 megabytes which may require 3-5 seconds to read from disks to CPU memory and 50-200 ms to transfer from CPU memory to GPU memory. Note these I/O overheads do not exist when N1 and N2 are computed on-the-fly on GPUs. Fortunately, as many real world spatial datasets are highly clustered, N1 and N2 grids are likely to be sparse and many data compression techniques can be applied to reduce storage overheads and data transfer times. Furthermore, as approximate s_k values are sufficient for finding the optimal grid level (k) in most cases, lossy compression techniques such as wavelet based ones, are acceptable. In addition, as CPU memory capacity limit can reach hundreds of gigabytes on commodity workstations in an economically sound way, these grids can be pre-loaded to CPU memory to avoid excessive disk I/O latency. While it is beyond the scope of this study to evaluate these data compression and buffer management techniques, we will explore this direction in our future work.

5. CONCLUSION AND FUTURE WORK

In this study, we have provided a parallel selectivity estimation technique to reduce memory footprint in spatial join processing on GPUs where memory capacity is typically a limiting factor in processing large-scale data. Experiments on joining the two MBB sets with nearly a million MBBs each have shown that our technique is able to reduce memory footprint by 38.4% in about 750 milliseconds when histograms are computed on-demand at multiple scales. When histograms are materialized, it only takes a few tens of milliseconds to search the best grid level for spatial join across multiple grid levels. The design is both simple and portable by utilizing well researched parallel primitives. GPU implementation is efficient and the proposed technique is effective in query optimization on large-scale spatial joins.

For future work, first, we would like to identify the performance bottleneck in the parallel primitives based implementation, re-implement the relevant parallel primitives and fine tune parameters to improve the overall performance. Second, we plan to systematically investigate whether to include larger grid sizes (e.g., 4096*4096 and up) in our selectivity estimation algorithm as they both provide memory footprint reduction potentials and incur high computing and memory overheads. Finally, we plan to compare our technique with alternative ones, e.g., using pinned CPU memory, in reducing GPU memory constraints in processing large-scale data.

6. REFERENCES

[1] http://en.wikipedia.org/wiki/Smartphone.

[2] Mishra, P, and Margaret, E. H (1992). Join processing in relational databases. ACM Computing Surveys. 24(1) 63-113.

[3] Jacox, E. H. and Samet, H. (2007). Spatial join techniques. ACM Transaction on Database System 32(1).

[4] Zhang, J., You, S. and Gruenwald, L (2012). High-performance online spatial and temporal aggregations on multi-core CPUs and many-core GPUs. Proceedings of ACM DOLAP workshop.

[5] Zhang, J., You, S. and Gruenwald, L (2012). U2STRA: High-Performance Data Management of Ubiquitous Urban Sensing Trajectories on GPGPUs. Proceedings of ACM City Data Management Workshop (CDMW).

[6] Zhang, J. and You, S. (2012). Speeding up Large-Scale Point-in-Polygon Test Based Spatial Join on GPUs. Proceedings of ACM BigSpatial Workshop.

[7] Samet, H. (2005). Foundations of Multidimensional and Metric Data Structures Morgan Kaufmann.

[8] Jin, J., An, N. and Sivasubramaniam, A (2000). Analyzing range queries on spatial data. Proceedings of IEEE ICDE.

[9] Hensley, J., Scheuermann, T., et al (2005). Fast Summed-Area Table Generation and its Applications. Computer Graphics Forum 24(3) 547-555.

[10] http://www.opengeospatial.org/standards/sfs

[11] 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.

[12] Aji, A., Wang, F. and Saltz, J.H (2012). Towards building a high performance spatial query system for large scale medical imaging data. Proceedings of ACM-GIS. 309-318

[13] Luo, L., Wong, M. D. F., et al. (2011). Parallel implementation of R-trees on the GPU. Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC).

[14] Yang, K., He, B., Fang, R., Lu, M., Govindaraju, N., Luo, Q., Sander, P. and Shi, J. (2007). In-memory grid files on graphics processors. Proceedings of ACM DaMoN Workshop.

[15] Beier, F., Kilias, T., and Sattler, K-U (2012). GiST scan acceleration using coprocessors. Proceedings of DaMoN.

[16] Pirk, H., Manegold, S. and Kersten, M. (2011). Accelerating Foreign-Key Joins using Asymmetric Memory Channels. Proceedings of ADMS.

[17] Kaldewey, T., Lohman, G., Mueller, R. and Volk, P. (2012). GPU join processing revisited. Proceedings of ACM DaMoN Workshop.

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

[19] Zhang, J. and You, S. (2012). CudaGIS: Report on the Design and Realization of a Massive Data Parallel GIS on GPUs. Proceedings of ACM IWGS Workshop.

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

[21] Wu, H., Diamos, G. F., et al (2012).Optimizing Data Warehousing Applications for GPUs Using Kernel Fusion/Fission. Proceedings of IPDPS Workshops, 2433-2442

[22] http://www.moderngpu.com/

[23] http://www.nyc.gov/html/dcp/html/bytes/applbyte.shtml

Appendix 1: A parallel Spatial join framework on GPUs

Spatial data is rich in data types and different spatial data types may allow different spatial operations, for example, distance calculation between points and polylines and point-in-polygon tests among points and polygons. We refer to the OGC SFS [10] for more details on spatial data modeling. While most of the existing spatial databases adopt Object-Relational data models for spatial data to extend relational databases functionality to spatial data, extensive dynamic memory allocations to construct spatial objects in memory is not cache friendly and can significantly degrade the performance. To boost the performance of the in-memory data structures for complex and read-only spatial data, we have designed an array-based physical data layout scheme [19]. For complex spatial objects such as polylines and polygons, in addition to their vertex arrays, auxiliary index arrays are also created. Point/vertex arrays and index arrays can be efficiently streamed among disks, CPU memories and GPU memories. While we are still actively experimenting the performance of R-Tree and Quadtree based spatial indexing and query processing techniques [19], in this study, we assume spatial index structures are available for neither datasets involved in a spatial join and we thus resort to a simple grid file based approach for spatial filtering (middle of Fig. A1).

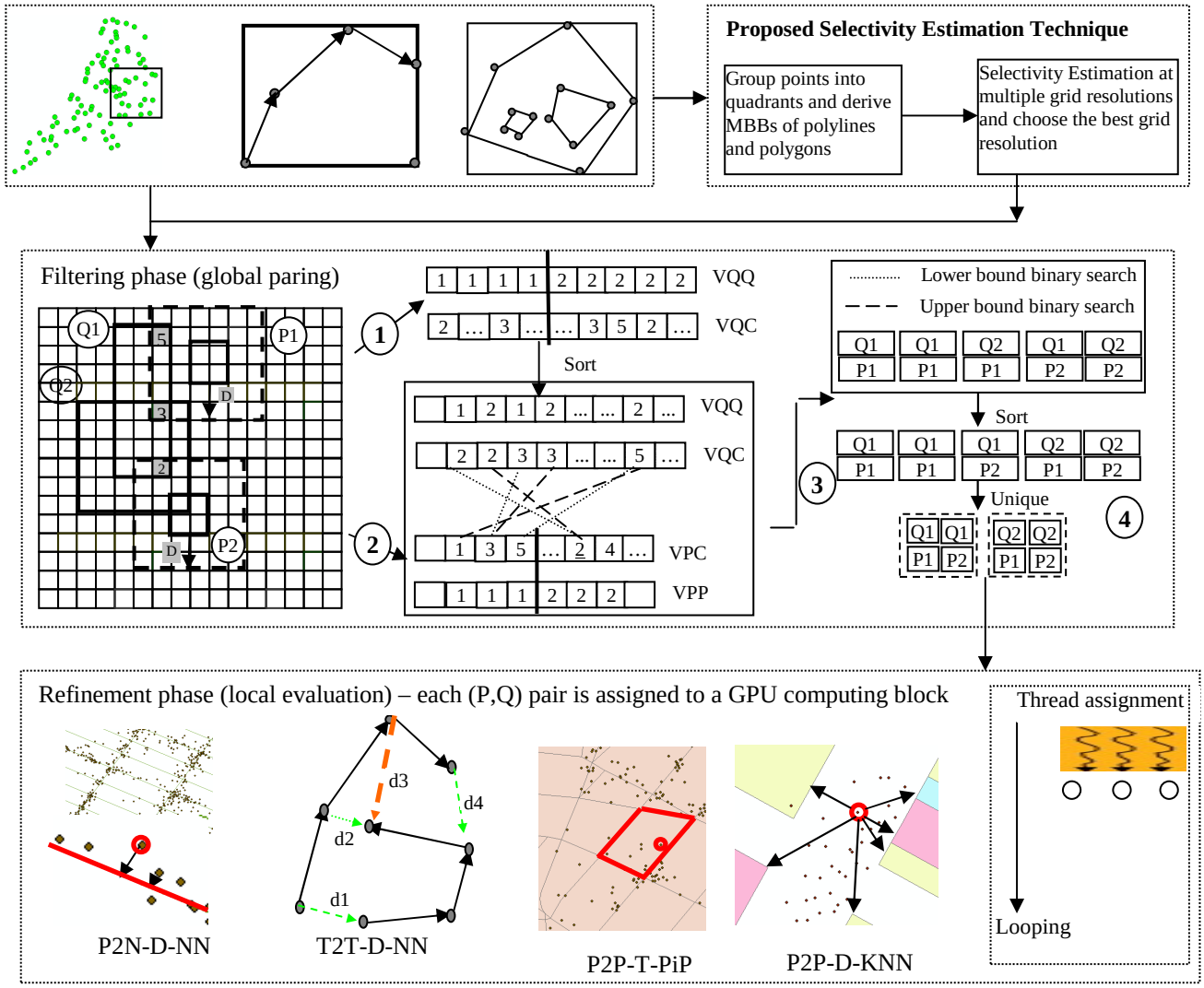


Fig. A1 A Framework of Parallel Spatial Join Processing on GPU

While points can be easily grouped into grid cells by chaining sorting and reduction (using grid cell identifiers as keys), MBBs of polylines and polygons may intersect with multiple grid cells. After both geometric objects are aligned with one or more grid cells, generating (P, Q) pairs can be transformed into a binary search problem. For each grid cell in the VPC vector, which stores the one-to-many mappings between the MBB of a geometric object in T1 to the grid cells that the MBB intersects, we search the cell in the VQC vector, which stores the one-to-many mappings between the MBB of a geometric object in T2 to the grid cells that the MBB intersects (as illustrated in the center of Fig. A1). The matched objects in T1 and T2 will be paired for subsequent refinement. Clearly, for MBB pairs that cover multiple grid cells, the (P, Q) pairs will be duplicated and need to be removed to avoid redundant spatial refinements.

During the refinement phase, (P, Q) pairs will be assigned to computing blocks as shown at the bottom part of Fig. A1. As there will be multiple points/vertexes in both P and Q (here we treat grouped points as a point collection object), we assign one set of points/vertexes to threads in the computing block while looping through all the other sets of points/vertexes to derive results that will be associated with either points/vertexes or the pairs of MBBs assigned to the computing block. This nested-loop style design is very efficient on GPUs as neighboring threads read neighing points/vertexes in one object (assuming P) before a loop begins and write to neighboring positions for outputting results after the loop finishes while they access the same point/vertex in another object (assuming Q) throughout the looping process. The memory access pattern is perfectly coalesced which is critical in GPU computing.

As an example, assuming P contains M points in a grid cell and Q contains the N vertices of a polygon, we can assign M points to threads while looping through the N vertices. Depending on the sizes of points/vertexes in P and Q and the configurations of GPU computing blocks, we may need to reshape the O(M*N) computation to maximize the utilization of GPU hardware. For example, when M is less than the warp size (currently 32 on CUDA enabled GPUs [20]), we can loop through K (>=2) points in the Q polygon simultaneously and reduce the number of the looping steps to ceiling(N/K). Similarly, when M is larger than the number threads in the computing block (assuming T), we may need to loop over the M points in ceiling(M/T) rounds. The parallel designs and implementations of point grouping, MBB rasterization, spatial filtering and several spatial filtering are documented in our previous work [4,5,6].

Appendix 2 A brief review on Multi-Dimensional Histograms for Selectivity Estimation and Data Summation

Multi-dimensional histograms can be built through regular gridding [24,31,32,36,38], non-regular gridding [33,37,39,41], clustering [25,40,43,46], quad-tree [44], R-Tree [40,45] and ECDF-tree [35] constructions. They can be used to estimate selectivity to facilitate overview style browsing (1,8,15), range queries [25,26,28,31,32,35,36,40] and spatial joins [27,28,33,39]. In addition to regular simple statistics, advanced statistics such as topology [32], sketches [34], density [35,37] and wavelet [39,42] may also be used.

[24] Beigel,R. and Tanin, E. (1998). The Geometry of Browsing. Proceedings of LATIN'98: Theoretical Informatics, 331-340.

[25] Acharya,S., Poosala,V. and Ramaswamy,S. (1999). Selectivity estimation in spatial databases. Proceedings of SIGMOD, 13–24.

[26] Aboulnaga, A. and Naughton, J. F. (2000). Accurate estimation of the cost of spatial selections. Proceedings IEEE ICDE, 123-134.

[27] An, N., Yang, Z.-Y., and Sivasubramaniam, A. (2001). Selectivity estimation for spatial joins. Proceedings of IEEE ICDE, 368-375.

[28] Mamoulis, N. and Papadias, D. (2001). Selectivity Estimation of Complex Spatial Queries. Proceedings of SSTD, 155-174.

[29] Wang, M., Vitter, J., et al (2001). Wavelet-Based Cost Estimation for Spatial Queries. Proceedings of SSTD, 175-196.

[30] Choi, Y-J. and Chung, C-W. (2002). Selectivity estimation for spatio-temporal queries to moving objects. Proceedings of ACM SIGMOD, 440-451.

[31] Sun, C., Agrawal, D. and El Abbadi, A. (2002). Exploring spatial datasets with histograms. Proceedings of IEEE ICDE, 93-102

[32] Lin, X, . Liu, Q., et al. (2003). Multiscale histograms: summarizing topological relations in large spatial datasets. Proceedings of VLDB, 814-825.

[33] Belussi,A., Bertino, E. and Nucita A. (2004). Grid based methods for estimating spatial join selectivity. Proceedings of ACM-GIS, 92-100.

[34] Das, A., Gehrke, J., and Riedewald, M. (2004). Approximation techniques for spatial data. Proceedings of SIGMOD, 695-706.

[35] Zhang, D., and Tsotras, V. J. and Gunopulos, D (2004). Efficient aggregation over objects with extent. Proceedings of PODS, 121-132.

[36] Elmongui, H., Mokbel, M. et al. (2005). Spatio-temporal Histograms. Proceedings of SSTD, 19-36.

[37] Gunopulos, D., Kollios, G., et al (2005). Selectivity estimators for multidimensional range queries over real attributes. The VLDB Journal , 137-154.

[38] Sun, C., Bandi, N. et al (2006). Exploring spatial datasets with histograms. Distributed and Parallel Databases 20(1) 57-88.

[39] Sun, J., Tao, Y. et al (2006). Spatio-temporal join selectivity. Information Systems 31(8):793-813.

[40] Eavis,T. and Lopez,A. (2007). rK-Hhist: an R-tree based histogram for multi-dimensional selectivity estimation. Proceedings of CIKM, 475–484.

[41] Luo, J., Zhou, X, et al (2007). Selectivity estimation by batch-query based histogram and parametric method. Proceedings of ADC. 93-102.

[42] Huang, D-S., Heutte, L.and Loog, M (2007). Spatial Selectivity Estimation Using Cumulative Density Wavelet Histogram. Proceedings ICIC (LNAI 4682), 493-504.

[43] Roh, Y. J., Kim, J. H. et al. (2010). Hierarchically organized skew-tolerant histograms for geographic data objects. Proceedings of SIGMOD, 627-638.

[44] Roh, Y-J., Kim, J-H, et al (2011). Efficient construction of histograms for multidimensional data using quad-trees. Decision Support Systems 52(1), 82 -94.

[45] Achakeev, D. and Seeger, B. (2012) A class of R-tree histograms for spatial databases. Proceedings of ACM-GIS. 450-453.

[46] Mai, H., Kim, J. et al (2013). STHist-C: a highly accurate cluster-based histogram for two and three dimensional geographic data points. GeoInformatica 17(2) 325-352.