Parallel Primitives based Spatial Join of Geospatial Data on GPGPUs

Jianting Zhang

Department of Computer Science, the City College of the City University of New York
email: jzhang@cs.ccny.cuny.edu

Abstract

Modern GPU architectures closely resemble supercomputers. Commodity GPUs that have already been equipped with personal and cluster computers can be used to boost the performance of spatial databases and GIS. In this study, we report our preliminary work on designing and implementing a spatial join algorithm on GPUs by using generic parallel primitives that have been well understood and efficiently implemented in many parallel libraries. In addition to help understand the inherent data parallelisms in spatial join operations, our experiments have shown that the reference implementation, which represents a tradeoff between code efficiency and code complexity, is able to achieve a 6.7X speedup when compared to an optimized CPU serial implementation. The result is encouraging in the sense that native implementation of spatial joins directly on top of GPU accelerators can potentially achieve much higher speedups for spatial joins which are fundamental to spatial databases and vector GIS. We also believe that the implementations of parallel spatial algorithms on top of generic parallel primitives can be an important first step towards designing and developing high-performance spatial-specific parallel primitives to make it easier to build parallel spatial databases and GIS.

Key words: Spatial join, GPU, High-performance, Parallel Primitives

1 Introduction

Spatial joins are fundamental in Spatial Databases (SDB) and Geographical Information System (GIS). Given two geospatial datasets (which can be points, polylines and polygons), a spatial join finds all pairs of objects satisfying a given spatial relationship between the objects, such as within, intersect and nearest neighbor. Spatial joins on CPUs have been extensively studied over the past few decades (Jacox and Samet 2007) given their practical importance. However, while research in parallel spatial joins can be dated back to 1990s (Hoel and Samet 1994), it was not until General Computing on Graphics Processing Units (GPGPU) technologies[1] on commodity hardware become available in recent years that using parallel spatial join processing to speed up SDB and GIS performance starts to be practical, both technologically and economically. As argued in (Clematis et al 2003), despite significant research on parallel geospatial processing, research before 2000 has very little impact on real practices at large due to quite a few factors, especially limited accesses to parallel hardware. On the other hand, the current GPU architectures closely resemble supercomputers as both implement the primary Parallel Random Access Machine (PRAM) characteristic of utilizing a very large number of threads with uniform memory latency (Hong et al 2011). As such, we believe research on efficient spatial joins on GPGPUs is timely and can potentially have a large impact on the geospatial computing community.

It is well known that spatial joins have two phases, i.e., filter and refinement (Jacox and Samet 2007, Shekhar and Chawla 2003). The optional filter phase relies on various spatial indexing data structures to filter out a large portion of candidate pairs to be joined while the refinement phase computes spatial relationships among filtered candidate pairs. A few GPGPU based spatial indexing data structures have been proposed in the past few years (e.g., Zhou et al. 2008, Zhang et al. 2010, Hou et al. 2011, Zhou et al 2011, Luo et al 2011) and can be applied in the filter phase. In this study, we will be focusing on the refinement phase. Since many spatial operations that are involved in the refinement phase, such as calculating distances and point-in-poly tests, are more computing intensive than testing the spatial relationships based on Minimum Bounding Boxes (MBB) in the filter phase, it is more desirable to use GPUs to speed up the refinement phase.

While computing efficiency is the driving motivation for GPGPU based parallel spatial joins, we believe it is also important to understand the inherent parallelisms in spatial joins so that the proposed parallel spatial join algorithms can sustain across different GPU architecture generations and interoperate with multi-core CPUs which increasingly have more GPU features as the numbers of CPU cores increases. As such, instead of directly providing a CUDA based implementation, our prototypical implementation adopts a parallel primitive based approach and the

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

implementation is based on Thrust library[2] that comes with CUDA distributions since version 4.0[3]. The implementation can serve as a baseline to compare with both a serial CPU implementation and a native implementation that uses CUDA directly which is under development. It is well-known that parallel primitives based implementations represent tradeoffs between coding complexity and code efficiency. Spatial join on multi-dimensional geographical data on top of generic parallel primitives for one-dimensional vectors may not be the most efficient ones. Nevertheless, the prototypical implementation can serve as a starting point for developing efficient spatial-specific parallel primitives to make it easier to build parallel SDB and GIS.

Our preliminary results show that, the reference implementation is able to achieve a 6.7X speedup compared with an optimized serial CPU implementation when joining pickup locations of 122,043 taxi trip records (points) with 43,252 MapPluto tax lots[4] (polygons) that have 293,335 vertices in the New York City (NYC) area. We expect significant higher speedups can be achieved when the spatial join algorithm is implemented in CUDA and provided as a parallel geospatial computing primitive. In addition, the reference implementation is more than 256X faster than joining the same two datasets using the state-of-the-art open source geospatial packages for indexing (libspatialindex[5]) and distance computation (GDAL/OGR[6]). The result clearly shows the potential of the performance gains in evolving existing SDB and GIS software that are optimized for traditional hardware architectures to modern hardware architectures. The features we have exploited in this study include simple array-based cache conscious data structures (v.s. sophisticated pointer-based data structures using dynamic memory allocations), in-memory processing (v.s. disk-resident) and parallel accelerations (v.s. using standalone uni-processors).

The rest of the paper is arranged as the following. Section 2 formulates the spatial join problem and discusses related works. Section 3 presents the design of the proposed spatial join algorithm using parallel primitives. Section 4 provides implementation details and experiment results including the comparisons with a baseline serial CPU implementation and an open source implementation using existing GIS software stack. Finally section 5 is the conclusion and future work directions.

2 Problem Formulation and Related Work

Given two vector geospatial datasets, which can be point, poloyline or polygon data types, spatially joining the two datasets can be performed when neither of the datasets, either of the datasets or both of the datasets are indexed (Jacox and Samet 2007). While traditionally indexing geospatial data is considered expensive and spatial join techniques have been developed for non-indexed spatial data, recent works have shown that spatial indexing can be efficiently performed on GPUs (Zhou et al. 2008, Zhang et al. 2010, Hou et al. 2011, Zhou et al 2011, Luo et al 2011). Furthermore, many geospatial data can be considered static relative to their update cycles and spatial indexing can be done offline. As such, we assume both datasets are indexed and their spatial indices can be used to filter out a large portion of candidate pairs to be joined. The filtered pairs are provided as a vector of (fid, tid) pairs where fid is the identifier of a basic unit of the joining dataset (DF) and tid is the identifier of a basic unit of the dataset to be joined (DT). Here the basic unit can be a tree node if tree indexing approaches are used to index the geospatial datasets or a cell of a grid if a grid-file is used for indexing. We refer to (Gaede and Gunther 1998, Samet 2005) for more details on spatial indexing.

In this study we consider joining a point dataset that is indexed by a quadtree and a polygon dataset that is indexed by an R-tree although the proposed approach can be generalized to many other spatial join scenarios. The filter phase can be implemented by querying the R-Tree using the expanded bounding boxes of quadtree nodes, i.e., (x1-D, y1-D, x2+D, Y2+D). A quadtree node is paired with an R-tree entry if its expanded bounding box intersects with the bounding box of the entry in a leaf R-tree node. Furthermore, we limit our discussion to distance based nearest-neighbor spatial join, i.e., for each point in DF, compute the minimum distance from the point to polygons in DT. Here the distance from a point to a polygon is canonically defined as the minimum distance between the point and all line segments of the rings (including both the outer ring and the inner rings of polygons with holes) of the polygon. While classic quadtree indexing on point data put each point in a tree node, we have observed that for large-scale high-resolution point data, very often points that are close to each other have similar data access patterns and it is beneficial to group them together in spatial joins. For example, there are half a million taxi trip records in

[2] http://code.google.com/p/thrust/
[3] http://developer.nvidia.com/cuda-toolkit-40
[4] http://www.nyc.gov/html/dcp/html/bytes/applbyte.shtml
[5] http://libspatialindex.github.com/
[6] http://www.gdal.org/

NYC and there can be thousands of taxi pickup locations roughly at the same point locations. When joining these points and the nearby tax lot polygons (to associate the taxi trips with land use categories that are used as proxies of trip purposes), they will be paired with the same R-Tree entries representing the respective polygons. As such, we assume a leaf quadtree node can hold a set of points. The mechanism can also be viewed as indexing a collection of points (which is supported by OGC Simple Feature Specification as GeometryCollection[7]) using a single quadtree node. The refinement phase requires compute pair-wise distances between the points under a quadtree node and the segments of a polygon indexed by an R-tree. This needs to be done for all the (fid, tid) pairs resulted from the filter phase. We further assume that the MBBs of both the quadtree nodes and the R-Tree nodes are provided as vectors and can be accessed by using the fids and tids, respectively. The outputs of the pair-wise distance computation are a vector of triples of (fid+pid, tid+sid, distance) where pid is the identifier of a point in the quadtree node referred by fid and sid is the identifier of a segment of the polygon referred by tid. To find the shortest distance between a point and a polygon, the binary minimum function should be applied to all distances under a same combination of fid+pid+tid. We next discuss the layouts of the inputs so that we can concentrate on parallel primitives based spatial join algorithm to be presented in Section 3.

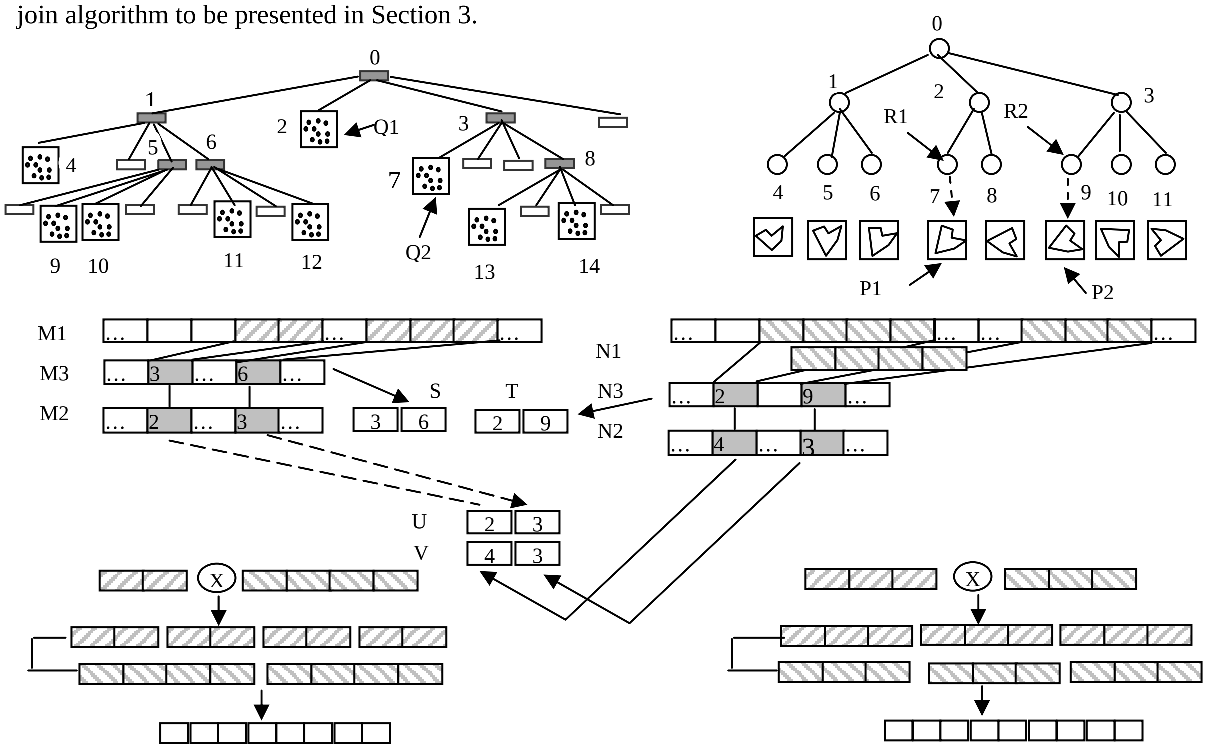

Fig. 1 Illustration of Spatial Join on Point (Indexed by a Quadtree) and Polygon (indexed by an R-Tree) and Storage Layout for Pair-Wise Distance Computation

As shown in the middle of Fig. 1, each input dataset is organized as three vectors with the first vector (M1) records the coordinates of points, the second vector (M2) records the numbers of points in the quadtree quadrants and the third vector (M3) records the starting positions of the points in all quadrants correspond to quadtree leaf nodes. Similarly another set of three vectors are used to record the similar information for polygons, i.e., N1 for the segments of polygon rings, N2 for the numbers of polygon segments and N3 for the starting positions of polygon segments in N1. While a vector of (fid, tid) pair can be used intuitively to represent the inputs of the refinement phase, the information stored in M2/M3 and N2/M3 correspond to the fids and tids are copied into four vectors, i.e., U, V, S, T, for fast and convenient data accesses. Here U stores the numbers of points in the quadrants correspond to the fids in the (fid, tid) pairs, V stores the numbers of segments of polygons correspond to the tids in the (fid,tid) pairs, S stores the starting positions of the quadrants in M1 and T stores the starting positions of the polygon segments in N1. As the expanded MBBs of quadtree quadrants many intersect with MBBs of multiple polygons as

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

indicated in the (fid,tid) pairs after the filter phase, both U/S and V/T can contain multiple copies of M2/M3 or N2/N3 elements, but the combinations of (U,V) and (S,T) should be unique in a spatial join.

In the example shown in Fig. 1, Q1 is paired with R1 and Q2 is paired with R2. There are 2 points in Q1 and 3 points in Q2. Their starting positions in the point coordinate array (M1) are 3 and 6, respectively. There are 4 segments in the polygon indexed by R1 and 3 segments in the polygon indexed by R2 and their starting positions in the polygon segment array (N1) are 2 and 9, respectively. Since the number of pairs after the filter phase is 2, there are 2 elements in U, S, V and T whose values are [2, 3], [3, 6], [4, 3] and [2, 9], respectively. As shown at the bottom of Fig. 1, there will be 2*4=8 distance computations for the first (fid, tid) pair and 3*3=9 distance computations for the second (fid, tid) pair and there will be 8+9=17 elements in the resulting distance vector. While it is straightforward to provide a serial implementation on CPUs by using three loops (i.e., the first loop for all (fid, tid) pairs, the second loop for all points in a quadrant and the third loop for all polygon segments), as we shall see in Section 3, it becomes non-trivial using a parallel computing model as each parallel processing unit (thread in GPGPU) needs to know exactly where to retrieve the data and where to output the results. As such, significant coordination efforts are required which is the core part of the primitives based spatial join algorithm.

Before we present the parallel algorithm in Section 3, we would like to briefly discuss our work with related works on parallel relational join on both multi-core CPUs (Blanas et al 2011) and GPUs (He et al 2008, Kaldewey 2012) which have attracted considerable research interests recently. We note two major differences between relational joins and spatial joins. First, as many spatial indexing approaches (such R-Trees) allow tree node to have overlapped MBBs, filter based on such spatial indexing structures can result in duplicated tree nodes and subsequently requires non-trivial combinations in the refinement phase. Second, spatial operations are usually more computing intensive than equijoin and most types of theta-joins which make GPGPU accelerations more desirable. Our work is closely related to the original works on parallel primitives based spatial joins and geospatial operations nearly 20 years ago (Hoel and Hamet 1994). However, while their works targeted for then supercomputers (Thinking Machine[8]) which were only accessible to a few very selective groups, our work targets at the commodity GPUs that are affordable by every researcher now days. In addition, while their works focused on the filter phase, our research focuses on the refinement phase.

Although it is straightforward to adopt a task-based parallelization schema at the (fid, tid) pair level on CPUs, it might not be a good choice for GPGPUs for two reasons. First, unlike the current generation of CPUs that have a few a dozen parallel processing cores with each is capable lunching 1-2 threads on a single computing node, GPUs typically have hundreds processing cores and support simultaneously launching hundreds of thousands threads. Task-level parallelization many not provide enough parallelism on GPUs for such spatial joins when the number of pairs is less than the number of processing units (blocks or threads) on GPGPUs. Second, the computing loads among the pairs can be highly imbalanced and the task-level parallelization can be too coarse to fully utilize GPGPU computing powers. The skewness can be more severe on GPGPUs than on multi-core CPUs as more processing units on GPGPUs are prone to be underutilized under a same degree of skewness. Furthermore, we are more interested in developing data parallel algorithms that can handle data skewness in an embedded manner than simply passing the burdens of handling skewness to middleware, operating systems or hardware built-in schedule modules which usually do not understand the inherent parallelism among data well.

3 Algorithm Design using Parallel Primitives

While computing efficiency is often the primary motivation for GPGPU applications including spatial joins, we believe it is also important to understand the inherent parallelisms in spatial joins from a research perspective. Parallel primitives that have been implemented in quite a few parallel libraries such as Thrust and CUDPP[9] facilitate quick prototypical implementations. The high-level implementations usually are easy to understand and have better portability although they may not be the most efficient ones that are achievable on parallel hardware. In this study, as an exploratory research effort to understand the parallelisms in spatial joins and their performance on modern GPGPUs, we have decided to adopt a parallel primitive based approach. More specifically, our parallel spatial join algorithm is built on top of the Thrust library that becomes part of Nvidia CUDA SDK after version 4.0. While it is beyond the scope of this paper to introduce the details of the parallel primitives, we refer to the Thrust project website for more technical details on the key primitives, including

[8] http://en.wikipedia.org/wiki/Thinking_Machines_Corporation
[9] http://code.google.com/p/cudpp/

transform, scan, reduce, gather, scatter and their variants. A brief explanation of the primitives with examples is provided in our technical report[10].

The overall structure of the algorithm is provided in Fig. 2 and an illustrative example is shown in Fig. 3. While the algorithm looks complex, each of the 25 steps corresponds to a single line of code to invoke a parallel primitive which makes the implementation really simple. The majority of the design (steps 1-21) is devoted to compute the positions of points and segments in their respective storage arrays (M1 and N1) for parallel processing units (CUDA threads in this case) so that distance computation can be performed in parallel (steps 22-25). While it is straightforward to perform a two-level loop in a serial program as the two looping variables can be increased sequentially, it becomes non-trivial using a parallel computing model. A combination of scatter, scan and gather is required to emulate the sequence of $\sum_{k=0}^{K-1}\sum_{i=0}^{U[k]}\sum_{j=0}^{V[k]} i*V[k]+j$ where k loops through all the K (fid, tid) pairs, i loops through all points and j loops through all segments within a (fid, tid) pair. We next try to explain how the serial loops can be realized using primitives on parallel machines.

Inputs: Vectors M1, N1, U, V, S, T (as defined in Section 2)
Outputs: Pair-wise distance vector among all points and segments across the K (fid,tid) pairs

1 **Transform** on **U** and **V** to compute the numbers of distance calculation pairs in all (fid,tid) pairs and store the result as vector **X1** using the **multiplies** binary function.
2 **Scan** (**exclusive**) on **X1** to compute the boundary positions of all (fid, tid) pairs and store the result as vector **X2**.
3 **Reduce** on **X1** to compute the total number of distance calculations for all (fid, tid) pairs and store the result as scalar **tot_pairs.**
4 **Scatter** the sequence of (0..**K**-1) to **X3** using **X2** as the map
5 **Scan** (**exclusive**) on **X3** and store the result in **X4** using the **maximal** binary function
6 **Gather** on **T** using **X4** as the map and store the results to **X5**
7 Gather on **S** using **X4** as the map and store the results to **X6**
8 **Scan** (**exclusive**) on **U** and store the result in **X7**
9 **Reduce** on U and compute the number of query points as scalar **tot_points**.
10 **Scatter** the sequence of (0..**K**-1) to **X8** using **X7** as the map.
11 **Scan** (**exclusive**) on **X8** and store the result in **X9** using the **maximal** binary function
12 **Gather** on **V** using **X9** as the map
13 **Scan** (**exclusive**) on **X10** and store the result in **X11**
14 **Scatter** **X11** using itself as the map and store the result in **X12**.
15 **Scan** (exclusive) on **X12** and store the result in **X13** using the **maximal** binary function
16 **Transform** on a sequence of (0.. **tot_pairs**-1) and **X13** using the **minus** binary function and store the results in **X14**.
17 **Transform** **X5** and **X14** using the **plus** binary function and store the results in **X15**
18 **Scatter** a sequence of (0..**tot_points**-1) to **X16** using **X11** as the map
19 **Scatter** **X7** to **X17** using **X4** as the map.
20 **Transform** on **X16** and **X17** using the **subtract** binary function and store the results in X18
21 **Transform** **X6** and **X18** using the **plus** binary function and store the results in X19
22 **Gather** on **M1** using **X19** as the map and store the results in X20
23 **Gather** on **N1** using **X15** as the map and store the results in X21
24 **Transform** on **X20** and **X21** using a **user-defined point-to-line distance** function and store the results in **X22**
25 **Reduce (by key)** on **X22** using the **minimum** binary function and store the results to the **output** vector.

Fig. 2 Algorithm Design Using Parallel Primitives

Steps 1-5 builds a template vector to mark the boundaries of (fid, tid) pairs (results stored in X4). Steps 6-7 broadcast the position of the first point in the quadrant corresponds to the fid of the kth (fid, tid) pair to all U[k]*V[k] pairs within the boundary of the pair for all the K pairs (results stored in X5). Steps 8-13 calculate the starting positions of all the points in all the K (fid, tid) pairs. After step 13, vector X11 actually stores $\sum_{k=0}^{P-1}\sum_{i=0}^{U[k]} i*V[k]$ for all possible combinations of k and i. Steps 14-17 compute the positions of segments in the N1 vector to be paired with all points for each (fid, tid) pair. Steps 14-16 essentially generate a sequence of 0..V[k]-1 that is duplicated U[k] times within each of the K (fid, pid) pairs to emulate the inner loop. Similarly steps 18-21 compute the

[10] http://geoteci.engr.ccny.cuny.edu/primquad/primquad_draft.pdf (pages 4-5).

positions of points in the M1 vector to be paired with all segments for each (fid, tid) pair. Steps 18-20 generate a sequence of 0…U[k]-1 for each (fid, tid) pair with each element in the sequence duplicated V[k] times to emulate the outer loop. Despite the differences among steps 14-17 and 18-21, both of the procedures require broadcast the loop numbers at all levels so that the global information can be used to compute the correct point/segment positions in their respective storage arrays. We note that several of the Thrust built-in binary functions, including plus, minus, multiplies, minimum and maximum, have been used in the design. However, these binary functions can be easily implemented if the underlying parallel libraries do not support them natively.

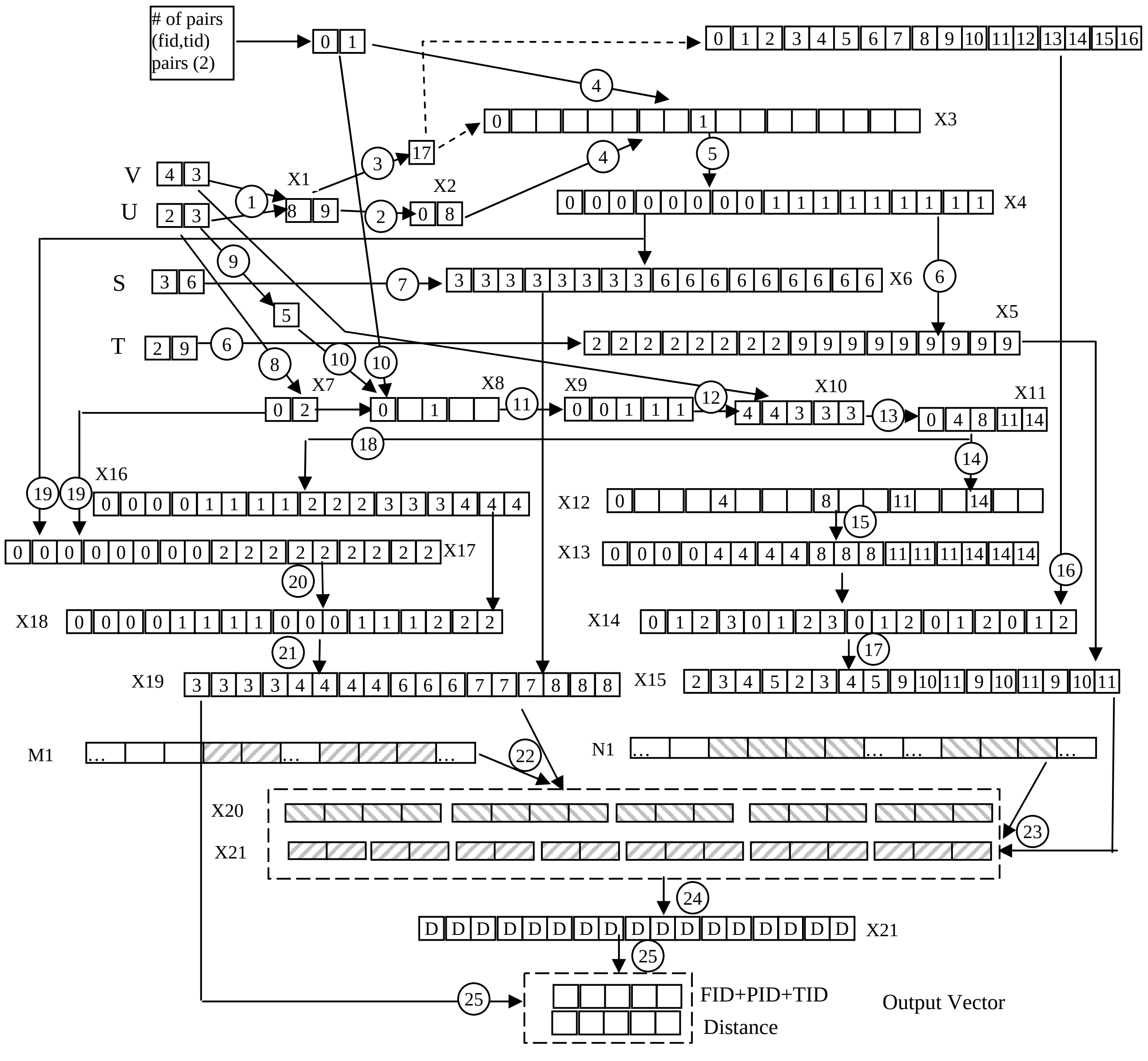

Fig. 3 Illustration of the Proposed Parallel Spatial Join Algorithm Using an Example

The key advantage of the parallel design is that, after all required positions are computed, calculating pair-wise distances across all the K (fid, tid) pairs becomes embarrassingly parallelizable and can be easily handled by virtually any parallel libraries. A major disadvantage is that, the positions need to be explicitly computed and stored. While the computation mostly requires additions and multiplications which are very fast on modern GPUs, accessing GPU global memory is very expensive (although most of the memory accesses are coalesced). The overall performance may depend on the relative weights of the computation part and memory access part. When more complex distance computation functions are used, the performance gains can be significant. On the other hand, when there are not sufficient computation workloads, the performance of the parallel design and implementation can be even worse than a straightforward serial implementation on CPUs. A related disadvantage is that the design has a large memory footprint requirement. Although many of the vectors used in the algorithm can be reused in a real implementation[11], explicitly storing the position vectors will limit the number of (fid, tid) pairs that can be processed on a single GPU device. However, as the design can virtually scale up linearly with the numbers of GPU devices, it is possible to run the implementation in a large cluster computer with fast network connection to address the memory limitation problem.

4 Implementation and Experiments

While our initial research is driven more by algorithmic design, to gain more insights on its performance for practical applications, we have implemented the algorithm on top of the Thrust library. As discussed previously, the implementation is fairly straightforward using Thrust where each step is implemented as a call to the respective Thrust primitive. The implementation is compiled using CUDA SDK 4.0 and the experiments are performed on an Nvidia Quadro 6000 GPU with 448 cores (1.15 GHZ) and 6 GB memory [12]. The host machine is a Dell Precision T5400 with dual Intel E5405 CPUs (2.0 GHZ) and 16 GB memory. For point data, we use a 1% sample of the pickup locations of taxi trips in Manhattan in January 2009 as the point data which has 122, 043 point locations. For polygon data, the MapPluto Tax Lot data in Manhattan is used which has 43,252 polygons and 293,335 vertices. Many of the tax lot polygons have regular shapes. This explains that the average number of vertices per polygon is only around 7. However, a small portion of the polygons have hundreds of vertices which make the polygon data skewed in computation. We have empirically set the maximum number of points in a quadtree quadrant (P) to 100 and set D to 100 feet for expanding bounding boxes in the filter phase.

As a comparison, we have also implemented the refinement phase on CPUs using straightforward loops. Our data storage layout design discussed in Section 2 actually fits CPU hardware architectures very well. Looping through one dimensional arrays is naturally cache friendly. Given that accesses to memory can be hundreds of times slower than accesses to registers and CPU architectures increasingly reply on caching to improve memory access performance, it is important to be cache friendly. Unlike the parallel primitives based GPU implementation that requires computing, writing and reading positions and distances explicitly to GPU memory in order to use the parallel primitives, they can reside in CPU registers which is more efficient. For fair comparison, we have put both the CPU and primitives based GPU implementations in a same program and used –O3 flag to optimize for speed for the CPU code. We note that both implementations use a same filter phase and thus we will only compare the performance of the refinement phase. Our results showed that the end-to-end runtime of the primitives based GPU implementation is 345.75 milliseconds to join the 122, 043 points in 8447 quadrants and 43, 252 polygons. A total of 142,927,001 distance computations for 1,503,640 (fid, tid) pairs has been observed. In contrast, the optimized CPU implementation requires 2,325.84 milliseconds. As such, a 6.7X speedup is achieved for the primitives based GPU implementation.

Both the CPU and GPU based implementation are main-memory based using the storage layout discussed in Section 2. As existing spatial databases and GIS are mostly designed for disk-resident data, to understand how main-memory based systems can improve the performance of geospatial computing, we have implemented the same spatial join task using open source packages. The open source based implementation uses libspaitalindex for R-tree indexing and query processing in the filter phase and GDAL/OGR for distance computation between points and polygons. For each point with a coordinate of (x,y), we first query the R-Tree to extract polygons that are with a square of (x-D, y-D, x+D, y+D) and then compute the minimum distance from the point to the polygons. Our experiments have shown that the end-to-end runtime of the open source implementation is 88,531.18 milliseconds which is 38 times slower than the CPU implementation and 256 times slower than the primitives based GPU implementation. The results clearly demonstrate the efficiency of main-memory based implementations. We believe the performance gap between disk-resident and main-memory based implementations can be attributed to the following factors. First, traditional spatial databases and GIS assume a very limited CPU memory capacity and uses sophisticated data structures and algorithms to accommodate memory limitations. However, given the increasing memory capacities (tens of GBs to TBs) and decreasing prices (~$5/GB retail prices), the assumed limitation is not valid anymore. Second, pointers and dynamic memory allocations have been extensively used in many spatial database and GIS software developments and they may not be cache friendly which is becoming increasingly important in modern hardware architectures including both CPUs and GPUs.

[11] We plan to release the source code after some cleaning in a way similar to a related project on building Binned Min-Max Quadtrees (BMMQ-Tree) on GPGPUs http://134.74.112.65/primquad/primquad.htm.
[12] http://www.nvidia.com/object/product-quadro-6000-us.html

Based on the experiments and analysis, we suggest re-examining the performance bottlenecks in geospatial computing by adopting an integrated hardware and software co-design approach. As computer processors are evolving into a parallel era, it is essential to fully utilize the parallel computing power provided by multi-core CPUs, many-core GPUs and distributed computing nodes (Zhang 2010). Existing Spatial Databases and GIS software need to be adapted to the new hardware architectures to efficiently process large-scale geospatial data and effectively solve real world problems. As a first step towards the adaptation, understanding the inherent parallelisms in major geospatial computing algorithms and designing their parallel implementations on top of well-understood and well-supported parallel primitives can play an important role. They can lay solid foundation in developing spatial-specific parallel primitives that are both high-performance and easy to use for geospatial computing applications.

5 Conclusion, Discussion and Future Work

We have designed a parallel spatial join algorithm that is suitable to implement on parallel machines including GPGPUs. Our prototypical implementation using the Thrust parallel library has demonstrated a 6.7X speedup over an optimized CPU serial implementation. The result is encouraging in the sense that native implementations of spatial joins directly on top of GPU accelerators can potentially achieve much higher speedups for spatial joins which are fundamental to spatial databases and vector GIS.

From a methodological perspective, the serial CPU implementation and the parallel primitives based GPU implementation represent two extremes with respect to efficiency and scalability. The serial implementation is efficient but not scalable for parallel execution while the primitives based implementation is scalable but not very efficient due to memory access overheads in storing and retrieving positions and intermediate results. We believe some hybrid approaches can potentially achieve both high efficiency and scalability at the levels that are appropriate for applications. For example, for polygon data that are not extremely skewed, it might be more beneficial to use two-levels of parallelisms on CUDA-enabled GPUs, i.e., (fid, tid) pairs at the computing block level and pair-wise distance computation at the thread level. In this case, both points and polygon segments correspond to a (fid, tid) pair can be loaded to GPU shared memory by all the threads in a computing block collaboratively so that no direct global memory accesses are needed during distance computation. Computing within a GPU computing block can be much similar to the CPU serial implementation with respect the two-level loop for the pair-wise distance computation within a (fid, tid) pair.

For future work, first, we would like to first implement the hybrid idea to explore design tradeoffs and potential performance gains. Second, we plan to implement a few indexing algorithms on GPGPUs for the filter phase so that we can perform spatial joins completely on GPUs. Third, while the framework of the spatial join algorithm discussed in this paper is generic, our current implementation is limited to joining points with polygons. We plan to make the implementation more generic to accommodate spatial joins of other data types for both parallel primitives based and hybrid designs.

References

1. Blanas, S., Li, Y. and Patel, J., 2011. Design and evaluation of main memory hash join algorithms for multi-core CPUs. Proceedings of ACM SIGMOD Conference.
2. Clematis, A., Mineter, M. et al., 2003. High performance computing with geographical data. Parallel Computing 29(10): 1275-1279.
3. Gaede V. and Gunther O., 1998. Multidimensional access methods. ACM Computing Surveys 30(2), 170-231.
4. He, B., Yang, K., et al. 2008. Relational joins on graphics processors. Proceedings of ACM SIGMOD conference.
5. Hoel, E. G. and Samet, H., 1994. Performance of Data-Parallel Spatial Operations. Proceedings of VLDB Conference.
6. Hong, S., Kim, S. K., et al., 2011. Accelerating CUDA graph algorithms at maximum warp. Proceedings of the 16th ACM symposium on Principles and practice of parallel programming.
7. Hou, Q., Sun, X., et al., 2011. Memory-Scalable GPU Spatial Hierarchy Construction. IEEE Transactions on Visualization and Computer Graphics 17(4), 466-474.
8. Jacox, E. H. and Samet, H., 2007. Spatial join techniques. ACM Transaction on Database System 32(1).
9. Kaldewey, T., Lohman, G., et al. 2012. GPU Join Processing Revisited. ACM DAMON Workshop.
10. Luo, L., Wong, M. D. F., et al., 2011. Parallel implementation of R-trees on the GPU. Proceedings of the 17th Asia and South Pacific Design Automation Conference (ASP-DAC).
11. Samet, H., 2005. Foundations of Multidimensional and Metric Data Structures Morgan Kaufmann.
12. Shekhar, S. and Chawla S., 2003. Spatial Databases: A Tour Prentice Hall.
13. Zhang, J. Towards Personal High-Performance Geospatial Computing (HPC-G): Perspectives and a Case Study. Proceedings of ACM HPDGIS workshop.
14. Zhang, J., You, S. and Gruenwald, L., 2010. Indexing large-scale raster geospatial data using massively parallel GPGPU computing. Proceedings of ACMGIS.
15. Zhou, K., Hou, Q., et al., 2008. Real-Time KD-Tree Construction on Graphics Hardware. ACM Trans. on Graphics 27(5).
16. Zhou, K., Gong, M., et al., 2011. Data-Parallel Octrees for Surface Reconstruction. IEEE Transactions on Visualization and Computer Graphics 17(5), 669-681.