Parallel Grid-based Colocation Mining Algorithms on GPUs for Big Spatial Event Data

Arpan Man Sainju, *Student Member, IEEE,* and Danial Aghajarian, *Student Member, IEEE* and Zhe Jiang, *Member, IEEE,* and Sushil Prasad, *Member, IEEE*,

Abstract—Colocation patterns refer to subsets of spatial features whose instances are frequently located together. Mining colocation patterns is important in many applications such as identifying relationships between diseases and environmental factors, but is computationally challenging due to the large number of instances and candidate patterns. Existing algorithms are mostly sequential, and thus can be insufficient for big spatial event data. Recently, parallel colocation mining algorithms have been developed based on the Map-reduce framework, which is economically expensive. Another work proposed a GPU algorithm based on iCPI tree, but assumes that the number of neighbors for each instance is within a small constant, and thus cannot be used when instances are dense and unevenly distributed. To address these limitations, we recently proposed grid-based GPU colocation mining algorithms that include a novel cell-aggregate-based upper bound filter, and two refinement algorithms. In this paper, we provide theoretical analysis of running time. Furthermore using GPU profiling, we identify our recent GPU implementation, GPU-grid-join, as a memory bound problem and to address its bottlenecks, we proposes GPU-grid-join+, an optimized GPU algorithm. Our experimental results on real world data shows that GPU-grid-join+ achieves 4 to 12-fold speedup over GPU-grid-join both running on Nvidia P100 GPU as well as 56 to 126-fold speedup over OpenMP implementation over Intel(R) Xeon(R) CPU with 12 cores. Also for synthetic data, the speedup is in ranges 3 to 7-fold and 9 to 42-fold respectively.

Index Terms—Colocation mining, GPU algorithms, Spatial Big Data.

1 Introduction

THIS paper focuses on mining colocation patterns from big spatial event data (e.g., up to millions of spatial instances). Given a set of spatial instances with their feature types and locations, the colocation mining problem [1] aims to identify subsets of feature types whose instances are frequently located in close spatial proximity. For example, in ecology, species with symbiotic relationships tend to live close with each other in geographic space (e.g., Nile Crocodiles and Egyptian Plover). In public health, diseases (e.g., lung cancer) may co-occur with certain environmental factors (e.g., air pollution). The colocation pattern mining problem is similar to traditional association rule mining [2] in that both problems aim to find subset of features that frequently occur together. However, the main difference is that there are no natural transactions given in continuous space for colocation mining as those in association rule mining (e.g., market basket transaction analysis).

Societal applications: Colocation mining is important in many applications that aim to find associations between different spatial event types and environmental factors. For example, in public safety, law enforcement agencies are interested in finding relationships between different crime event types and potential crime generators [3]. In ecology, scientists analyze common spatial footprints of various species to capture their interactions and spatial distributions [4], [5]. In public health, identifying colocation patterns can help study disease transmission and environmental factors can help reveal the causes [6], [7]. In climate science, colocation patterns help reveal relationships between the occurrence of different climate extreme events. In location based services, colocation patterns help identify travelers that share the same favourite locations to promote effective tour recommendation [8].

Challenges: Mining colocation patterns from big spatial event data poses several computational challenges. First, unlike traditional association rule mining problems, there is no natural transactions given in colocation mining problems (all spatial instances are embedded in continuous space). In order to evaluate if a candidate colocation pattern is prevalent, we need to explicitly generate pattern instances. Second, the number of candidate colocation patterns are exponential to the number of spatial features. Evaluating a large number of candidate patterns can be computationally prohibitive. Third, the number of pattern instances can be enormous when the number of event instances is large and instances are clumpy (i.e., there are many instances within the same spatial neighborhoods). For example, given 100,000 event instances and assume that the clumpiness is 10 (i.e., every event instance has 10 neighboring instances of the same event type), the number of colocation pattern instances of cardinality 5 (i.e., there are five event types in the pattern) can be up to $100,000 \times 10^5 = 10^{10}$. This not only makes pattern instance generation a computationally challenging task, but also creates a memory bottleneck.

Related work: Colocation pattern mining has been studied extensively in the literature. Most of existing colocation mining algorithms are sequential, including early work

- *Arpan Man Sainju and Zhe Jiang were with the Department of Computer Science, University of Alabama, Tuscaloosa, AL, 35487. Danial Aghajarian and Sushil Prasad were with the Department of Computer Science, Georgia State University, Atlanta, GA, 30302.*
- *Arpan and Danial are co-first-authors with equal contributions.*
- *Zhe Jiang (zjiang@cs.ua.edu) is the contact author.*

Manuscript received October xx, 2017; revised August xx, xxxx.

on spatial association rule mining [9], [10] and colocation patterns based on event-centric model [1]. Various computational techniques have been proposed to efficiently identify colocation patterns, including Apriori property, multi-resolution upper bound filter [1], density-based filter [11], grid-based hash-index and plane sweep [12], partial join [13] and joinless approach [14], iCPI tree based colocation mining algorithms [15]. There are also works on identifying regional [16], [17], [18] or zonal [19] colocation patterns, and statistically significant colocation patterns [20], top-K prevalent colocation patterns [21], prevalent patterns without thresholding [22] or on extended spatial objects [23]. These methods do not utilize recent big data computational platforms, and thus can be insufficient when the number of event instances is very large (e.g., several millions). In recent years, Map-reduce framework [24], [25] and GPU platforms [26], [27], [28], [29], [30], [31], [32], [33] have been used for spatial data processing, particularly for parallel spatial join operations, but few studies have been done for colocation mining. [34] proposed map-reduce framework to mine colocations from a large data volume. However, these algorithms need a large number of nodes to scale up, which is economically expensive, and their reducer nodes have a bottleneck of aggregating all instances of the same colocation patterns. Indeed, Map Reduce based algorithms can be complementary to GPU algorithms. One work proposes a GPU based parallel colocation mining algorithm [35] using iCPI tree [36], [37], [38] and the joinless approach, but this method assumes that the number of neighbors for each instance is within a small constant (e.g., 32), and thus can be inefficient when instances are dense and unevenly distributed. There are also extensive works on parallel association rule mining algorithms in the literature [39], [40], [41], [42], [43], but these methods cannot be directly used for colocation mining due to the lack of natural transactions being given.

To address limitations of related work, we recently proposed parallel grid-based colocation mining algorithms on GPUs [44]. Our algorithms include a novel cell-aggregate-based upper bound filter and two parallel implementation of refinement algorithms. Proposed cell-aggregate-based filter computes upper bounds of the interest measure of colocation patterns based on aggregated counts of event instances in neighborhood cells. The upper bound can be computed in parallel on GPU without generating pattern instances, and is also insensitive to pattern *clumpiness* (the average number of overlaying colocation instances for a given colocation instance) compared with the existing multi-resolution filter.

This journal paper extends our recent work [44] with further GPU optimization, as well as detailed theoretical analysis and experimental evaluations. We make the following additional contributions in the journal paper:

- We conducted profiling and found out that the problem is memory bound for GPU architecture. Then based on detected bottlenecks, we proposed GPU optimizations including reducing unnecessary memory transfer and using batch memory transfer to replace frequent small memory transfers. We also redesigned preprocessing and filter kernels.
- We conducted experimental evaluations. Results on real data show that our optimized GPU algorithm (GPU-grid-join+) achieves 4 to 12-fold speedup over our conference version (GPU-grid-join) on Nvidia P100 GPU, as well as 56 to 126-fold speedup over OpenMP implementation on Intel(R) Xeon(R) CPU with 12 cores. On synthetic data, the speedup is in ranges of 3 to 7-fold and 9 to 42-fold respectively.
- We analyzed the theoretical time complexity of our GPU colocation mining algorithms.

Scope and outline: We focus on spatial colocation patterns defined by the event-centric models [1]. Other colocation definitions such as Voronoi diagram based are beyond our scope. We also assume the underlying space is Euclidean space. In addition, we only focus on issues related to computational efficiency of parallel colocation mining algorithms on GPUs. Parallel algorithms on other platforms such as Map Reduce framework can be considered as complementary to our work.

The outline of the paper is as follows. Section 2 reviews basic concepts and the definition of the colocation mining problem. Section 3 introduces our recent GPU colocation algorithms in [44], including the upper bound filter and two refinement algorithms, as well as further GPU optimization proposed in this paper. Section 4 analyzes the theoretical properties of algorithms, including correctness and completeness, as well as time complexity. Section 5 evaluates proposed methods. Section 6 provides some discussions. Section 7 concludes the paper with potential future research directions.

2 PROBLEM STATEMENT

2.1 Basic Concepts

This subsection reviews some basic concepts based on which the colocation mining problem can be defined. More details on the concepts are in [1].

Spatial feature and instances: A *spatial feature* is a categorical attribute such as a crime event type (e.g., assault, drunk driving). For each spatial feature, there can be multiple *feature instances* at the same or different point locations (e.g., multiple instances of the same crime type "assault"). In the example of Figure 1(a), there are three spatial features (A, B and C). For spatial feature A, there are three instances (A_1, A_2, and A_3). Two feature instances are *spatial neighbors* if their spatial distance is smaller than a threshold. Two or more instances form a *clique* if every pair of instances are spatial neighbors.

Spatial colocation pattern: If the set of instances in a clique are from different feature types, then this set of instances is called a *colocation (pattern) instance*, and the corresponding set of features is a *colocation pattern*. The *cardinality* or *size* of a colocation pattern is the number of features involved. For example, in Figure 1(a), (A_1, B_1, C_1) is an instance of colocation pattern (A, B, C) with a size or cardinality of 3. If we put all the instances of a colocation pattern as different rows of a table, the table is called an *instance table*. For example, in Figure 1(b), the instance table of colocation pattern (A, B) has three row instances, as shown in the third table of the bottom panel. A spatial colocation pattern is

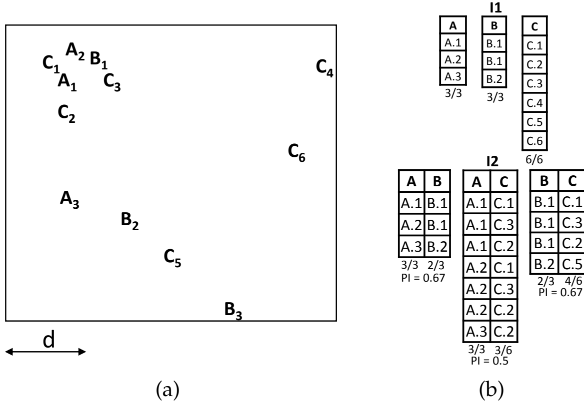


Fig. 1. A problem example with inputs and outputs. (a) Input spatial features and instances; (b) Candidate and prevalent colocation patterns, instance tables

prevalent (significant) if its feature instances are *frequently* located within the same neighborhood cliques. In order to quantify the prevalence or frequency, an interestingness measure called participation index has been proposed [1].

The *participation ratio* of a spatial feature within a candidate colocation pattern is the ratio of the number of unique feature instances that participate in colocation instances to the total number of feature instances. For example, in Figure 1, the participation ratio of B in candidate colocation pattern (A, B) is $\frac{2}{3}$ since only B_1 and B_2 participate into colocation instances ((A_1, B_1), (A_3, B_2)). The *participation index* (PI) of a candidate colocation pattern is the minimum of participation ratios among all member features. For example, the participation index of the candidate colocation pattern (A, B) in Figure 1 is the minimum of $\frac{3}{3}$ and $\frac{2}{3}$, and is thus $\frac{2}{3}$. We use "candidate colocation patterns" to refer to those whose participation index values are undecided.

2.2 Problem Definition

We now introduce the formal definition of colocation mining problem [1].

Given:
- A set of spatial features and their instances
- Spatial neighborhood distance threshold
- Minimum threshold of participation index: θ

Find:
- All colocation patterns whose participation index are above or equal to θ

Objective:
- Minimize computational time cost

Constraint:
- Spatial neighborhood relationships are defined in Euclidean space

Figure 1 illustrates a problem example. The input data (Figure 1(a)) contains 12 instances of 3 different spatial features A, B, and C. The neighborhood distance threshold is d. The prevalence threshold is 0.6. The output prevalent colocation patterns include (A, B) (participation index 0.67) and (B, C) (participation index 0.67). Colocation patterns with a cardinality of 1, e.g., (A), (B), (C), are trivial, since they only have one feature and are always prevalent. Pattern (A, C) is not prevalent, because its participation index is only 0.5 (only half instances of C participates in the pattern).

3 Proposed Approach

This section introduces our proposed GPU colocation mining algorithms. We first introduce algorithms proposed in our conference paper [44]. Section 3.1 overviews the entire algorithm structure. Section 3.2 and Section 3.3 introduce the parallel implementations of our upper bound filter and two refinement algorithms respectively. Further GPU optimization beyond the conference version is introduced in Section 3.4.

3.1 Algorithm Overview

There are several challenges in the design and implementation of GPU parallel colocation mining algorithms. First, the number of row instances of a candidate colocation pattern can grow significantly with pattern cardinality when instances are dense, creating both intensive computational load and memory bottleneck. Second, memory coalesce is also a challenge due to the fact that the number of row instances produced by each kernel thread cannot be predetermined, and thus it is non-trivial to map a kernel thread index to the corresponding row numbers of the output instance table in GPU global memory.

Recently, we proposed GPU implementations [44]. The overall algorithm framework is shown in Algorithm 1, which generates and evaluates candidate colocation patterns by iterations over pattern cardinality (similar to sequential the algorithm in [1]). Specifically, the instance table of a size $k+1$ pattern is generated by join operations over instance tables of size k patterns. Our implementation is heterogeneous with both CPU codes and GPU codes (kernel function calls), and it consists of three major components: **preprocessing**, **filtering**, and **refinement**.

The **preprocessing phase** (steps 1 to 5) is done in CPU sequentially. It generates size 2 candidate patterns, creates a grid index over instances (cell to instance index), and counts the number of instances under each feature type in each cell ($CountMap$). After the computation, the grid index and $CountMap$ are transfered from host memory to device memory.

The **filtering phase** (steps 7 to 8) is done in GPU mostly. For each candidate pattern c with a cardinality of $k+1$ (in the $k+1$th iteration), the filter kernels (ParallelCellAggregateFilter) will be launched, before which the candidate pattern c is transferred from host to device. After kernel execution, an upper bound of participation index (PI) of c will be returned by the kernel from device to host.

The **refinement phase** (steps 9 to 19) is executed only if the upper bound of PI of candidate pattern c is above the minimum threshold. It generates the instance table of c and computes the exact PI through GPU refinement kernels (either grid-join based or prefix-join based). Since each kernel thread is used to generate a number (unknown before execution) of rows in the output instance table of c, there is a memory coalesce issue between threads. We address the issue with two rounds execution and a slot count array for kernel threads (more details in Section 3.3).

Algorithm 1 Parallel-Colocation-Miner

Input: A set of spatial features F
Input: Instances of each spatial features I_{f2F}
Input: Neighborhood distance threshold d
Input: Minimum prevalence threshold θ
Output: All prevalent colocation patterns P

1: Initialize $P \leftarrow \emptyset$, $k \leftarrow 1$, $C_k \leftarrow F$, $P_k \leftarrow F$
2: Initialize $C_{k+1} \leftarrow$ APRIORIGEN(P_k, $k+1$), $P_{k+1} \leftarrow \emptyset$
3: Overlay a regular grid with cell size $d \times d$ (total N cells)
4: Create cell index: instances of each feature in each cell
5: Compute $CountMap$ in one round scanning
6: **while** $|C_{k+1}| > 0$ **do**
7:   **for each** $c \in C_{k+1}$ **do**
8:     $Upperbound$ =PARALLELCELLAGGREGATEFILTER($CountMap$, $PCountMap$, c)
9:     **if** $Upperbound \geq \theta$ **then**
10:       **if** Prefix Join Refinement **then**
11:         $PI \leftarrow$ PREFIXJOINBASEDREFINEKERNEL(I,c,$Slots$,R_1)
12:         **if** $PI \geq \theta$ **then**
13:           $P_{k+1} = P_{k+1} \cup c$
14:           $I_c \leftarrow$ PREFIXJOINBASEDREFINEKERNEL(I,c,$Slots$,R_2)
15:       **else if** Grid Search Refinement **then**
16:         $PI \leftarrow$ GRIDBASEDREFINEKERNEL(I, c, $Slots$, R_1)
17:         **if** $PI \geq \theta$ **then**
18:           $P_{k+1} = P_{k+1} \cup c$
19:           $I_c \leftarrow$ GRIDBASEDREFINEKERNEL(I, c, $Slots$, R_2)
20:   $P \leftarrow P \cup P_{k+1}$
21:   $k \leftarrow k+1$; $C_k \leftarrow C_{k+1}$; $P_k \leftarrow P_{k+1}$
22:   $C_{k+1} \leftarrow$ APRIORIGEN(P_k, $k+1$)
23:   $P_{k+1} \leftarrow \emptyset$
24: **return** P

3.2 Cell-Aggregate-Based Upper Bound Filter

To introduce proposed cell aggregate based filter, we define a key concept of **quadruplet**. A quadruplet of a cell is a set of four cells, including the cell itself as well as its neighbors on the right, bottom, and right bottom. For a cell that is located on the right and bottom boundary of the grid, not all four cells exist and its quadruplet is defined empty (these cells will still be covered by other quadruplets). For example, in Figure 2, the quadruplet of cell 0 includes cells $(0, 1, 4, 5)$, while the quadruplet of cell 15 is an empty set. For each a quadruplet, our filter computes the aggregated count of instances for every feature in the candidate pattern. If the aggregated count for any feature is zero, then there cannot exist colocation instances in the quadruplet. Otherwise, we pretend that all these feature instances participate into colocation pattern instances. This tends to overestimate the participating instances of a colocation pattern (an "upper bound"), but avoids expensive spatial join operations. Compared with the existing multi-resolution filter [1], which computes the upper bound based on generating coarse scale instance tables, our cell aggregate based filter has two advantages: first, it is embarrassingly parallelizable and can leverage the large number of GPU cores; second, its performance does not rely on the assumption that pattern instances are clumpy into a small number of cells, which is required by the existing multi-resolution filter.

Algorithm 2 ParallelCellAggregateFilter

Input: $CountMap$, count of feature instances
Input: $PCountMap$, count of participating feature instances
Input: c, candidate colocation pattern
Output: $upperBound$, upper bound of participation index

1: **for each** cell i **do in parallel**
2:   $QuadrupletCells$ = GETQUADRUPLET(cell i)
3:   $QuadrupletCount[] \leftarrow 0$
4:   **for each** feature $f \in c$ **do**
5:     **for each** cell $j \in QuadrupletCells$ **do**
6:       $QuadrupletCount[f] \leftarrow QuadrupletCount[f] + CountMap[j][f]$
7:     **if** $QuadrupletCount[f] == 0$ **then**
8:       **finish** the parallel thread for cell i
9:   **for each** feature $f \in c$ **do**
10:     **for each** cell $j \in QuadrupletCells$ **do**
11:       $PCountMap[j][f] \leftarrow CountMap[j][f]$
12: **for each** feature $f \in c$ **do**
13:   $PR[f]$ = PARALLELSUM($PCountMap[\,][f]$)/$|I_f|$
14: $upperBound$ = MIN(PR)
15: **return** $upperBound$

The GPU implementation details are shown in Algorithm 2 and Figure 2. The $CountMap$ (count of instances under each feature in each cell) and cell to instance index are already transferred to GPU global memory during preprocessing. We launch a filter kernel call for each candidate pattern c (whose value is transferred from host to device during the call). Each kernel thread is assigned to one quadruplet. For example, in Figure 2(b), thread 0 is assigned to the quadruplet of cells 0, 1, 4, and 5. The thread computes the aggregated counts of instances under each feature in c in the quadruplet, and saves results to GPU local memory ($QuadrupletCount$). In this case, thread 0 finds 2 instances of A and 1 instance of B. Since none of the feature count is zero, the thread copies the four $CountMap$ elements for the quadruplet into $PCountMap$, an array in GPU global memory that records the instance count for each feature in every cell that potentially participates in the candidate colocation pattern. After all threads finish execution, we can compute the participating index based on $PCountMap$ (the sum is done in parallel through the thrust library), returning an upper bound from device to host. We used one dimensional GPU blocks with 1024 threads in each block.

3.3 Refinement Algorithms

As shown in Algorithm 1, we have two options for refinement algorithms, one generating colocation instances by grid-based spatial join (geometric approach), and another generating colocation instances by prefix-based relational table join (combinatoric approach).

Grid-based approach: The grid-based approach generates an instance table of a size $k+1$ pattern by doing a spatial join between the instance table of a size k pattern and the instances of the k+1th (last) feature. A grid index (i.e., cell to

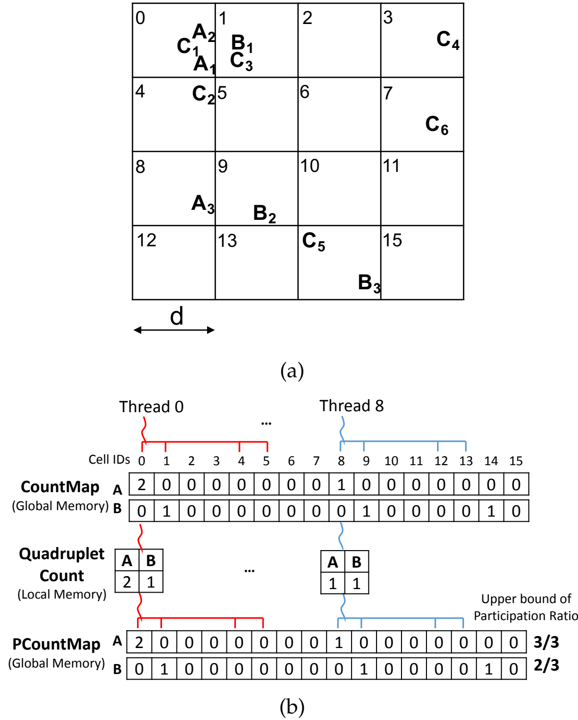


Fig. 2. Grid-aggregate based Upper Bound Filter: (a) A regular grid (b) An execution trace of upper bound filter

Algorithm 3 $GridBasedRefineKernel$

Input: I, instance tables of size k colocation patterns
Input: c, candidate size $k+1$ colocation pattern
Input: $Status$, value R_1 for PI calculation and slot counting; value R_2 for instance table generation
Input: $Slots$, it records output in round R_1, provides input in round R_2
Output: PI if $Status = R_1$; or I_c if $Status = R_2$

1: Initialize $I_c \leftarrow \emptyset$
2: Find instance table I_{c_k} for k-prefix-pattern $c[1:k]$
3: **for each** row instance $I_{c_k}[i] \in I_{c_k}$ **do in parallel**
4: get $neighborhood$ cells of first feature instance in $I_{c_k}[i]$
5: **for each** cell i in $neighborhood$ **do**
6: **for each** instance ins of feature $c[k+1]$ in cell i **do**
7: **if** ins is neighbor of all instances in $I_{c_k}[i]$ **then**
8: Create row instance of c as $< I_{c_k}[i], ins >$
9: **if** $Status == R_1$ **then**
10: Update BitMap, Update $Slots$
11: **else if** $Status == R_2$ **then**
12: Insert $< I_{c_k}[i], ins >$ into I_c
13: **if** $Status == R_1$ **then**
14: **for each** feature $f \in c$ **do**
15: $PR[f] \leftarrow$ PARALLELSUM$(BitMap[\,][f])/|I_f|$
16: PI = MIN$(PR[\,])$
17: **return** PI, keep slot counts
18: **else if** $Status = R_2$ **then**
19: **return** I_c

instance index) is used to prune out irrelevant instances of the last feature type. Algorithm 3 and Figure 3 provide the GPU implementation details. A kernel is launched for each size $k+1$ candidate pattern c (e.g., ABC in this example). Input size k instance table (e.g., AB in this example) is transferred from host to device (GPU global memory) before each kernel launch. Each kernel thread is assigned to one row in the size k instance table (e.g., (A_1, B_1) for thread 0, (A_2, B_1) for thread 1). The thread 0 will first identifies which cell the first feature's instance is located in (in this case, A_1 is in cell 0). Then it will search all the $k+1$th feature's (C's in this case) instances within the neighborhood cells (e.g., cells 0, 1, 4, and 5) through "**cell to instance index**" (**grid index**). The index (in GPU global memory) is illustrated in Figure 3, in which the start and end positions for instances under each feature in each cell are stored in an array. For instance, the first two elements record the start and end positions of A's instances in cell 0; the third and fourth elements record the start and end positions of B's instances in cell 0 (they are NULL pointers since there is no instance of B in the cell). Through the index, thread 0 can identify C_1, C_3 and C_2 consecutively in the neighborhood cells. The same is for thread 1. Thus, thread 0 and thread 1 are naturally coalesced due to the fact that their first instances (A_1 and A_2) are in the same cell (thus they search the same neighborhood cells). For each C instances, thread 0 checks if the instance is a spatial neighbor for both A_1 and B_1 (i.e., they form a colocation instance of ABC). If so, an output colocation instance of ABC is produced.

Two round execution and slot counts: One issue in generating instances of size $k+1$ patterns is that each thread may generate an unknown number of rows in output instance table, and thus it is hard to pre-estimate the size of output instance table and pre-determine which table rows each thread should write to. We address the issue by running the kernel in two rounds: in the first round R_1, the size k instance table is transferred from host to device. Each kernel thread counts the number of output instances it generates, and stores the result into its corresponding element in a slot count array called $Slots$ in global memory. A bitmap (one dimensional array in GPU global memory) is also generated as a byproduct that can be used to compute the PI. If the PI is above the prevalence threshold, we launch the kernel function in the second round R_2, and allocate GPU global memory of output instance table based on the total slot counts. Each kernel thread writes output instances in corresponding rows determined by slot counts. For example, thread 0 generates 3 row instances ($(A_1B_1C_1)$, $(A_1B_1C_3)$, $(A_1B_1C_2)$), so the second kernel thread has to start writing instances from the $4th$ row. For each candidate pattern, a fix amount of global memory is allocated for the slot count array. Output size $k+1$ instance table is transferred from device to host due to limited device memory for next kernel launch.

Prefix-based Join based refinement: Another option is to generate size $k+1$ instance table by a combinatorics approach. For example, when generating instance table of pattern (A, B, C), we can join rows in instance tables of (A, B) and (A, C). The join condition is that the first k instances from the two tables should be the same, and the last instances from two tables should be spatial neighbors.

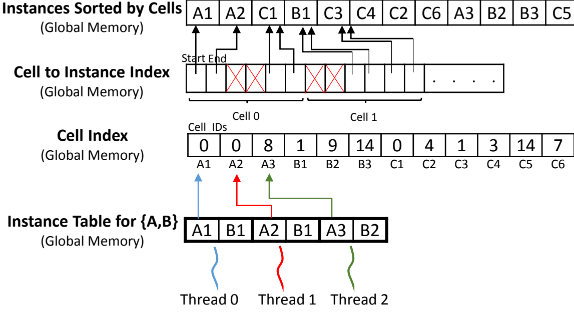


Fig. 3. Illustrative execution trace for grid-based refinement

Algorithm 4 $PrefixJoinBasedRefineKernel$

Input: I, instance tables of size k colocation patterns
Input: c, candidate size $k+1$ colocation pattern
Input: $Slots$, it records output in round R_1, provides input in round R_2
Input: $Status$, value R_1 for PI calculation and slot counting; value R_2 for instance table generation
Output: PI if $Status = R_1$; or I_c if $Status = R_2$

1: Find instance table I_{c_1} for k-prefix patterns $c[1:k]$
2: Find instance table I_{c_2} for k-prefix patterns $c[1:(k-1), k+1]$
3: **for each** row instance $I_{c_1}[i]$ in I_{c_1} **do in parallel**
4:   **for each** row instance $I_{c_2}[j]$ in I_{c_2} starting with $I_{c_2}[i][1]$ **do**
5:     **if** $I_{c_1}[i]$ and $I_{c_2}[j]$ forms an instance of c **then**
6:       Create new instance by merging $I_{c_1}[i]$ and $I_{c_2}[j]$
7:       **if** $Status == R_1$ **then**
8:         Update BitMap, Update $Slots$
9:       **else if** $Status == R_2$ **then**
10:         Insert $< I_{c_k}[i], ins >$ into I_c
11: **if** $Status == R_1$ **then**
12:   **for each** feature $f \in c$ **do**
13:     $PR[f] \leftarrow$ PARALLELSUM$(BitMap[\,][f])/|I_f|$
14:   PI = MIN$(PR[\,])$
15:   **return** PI, keep slot counts
16: **else if** $Status = R_2$ **then**
17:   **return** I_c

For example, when joining a row (A_1, B_1) with another row (A_1, C_1), we check that the first instance is the same (A_1), and the last instances B_1 and C_1 are spatial neighbors. So these two rows are joined to form a new row instance (A_1, B_1, C_1). In sequential implementations [1], the join process can be done efficiently through sort-merge join. However, for GPU algorithm, sort merge is difficult due to the order, dependency and multi-attribute keys. We choose to use prefix join instead. A **prefix-based index** is built on the second table based on instances of the first spatial feature in the GPU global memory. In the example of Figure 4, which uses the same input data as Figure 2(a), the first two elements of the prefix index record the start and end row ids of all colocation instances of AB that start with A_1. Similarly, the third and fourth elements record the start and end row ids of instances that start with A_2. Similar to grid-based refinement, the prefix-join based refinement uses two-round execution with slot counts. Each kernel thread

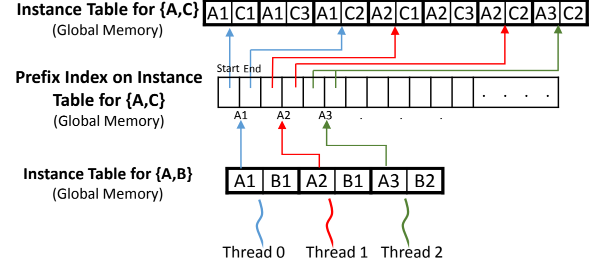


Fig. 4. Illustrative execution trace for prefix-join-based refinement

is allocated to one row in the first size k instance table I_{c_1} (e.g., kernel thread 0 is allocated to (A_1, B_1), while thread 1 is allocated to (A_2, B_1)). Each kernel thread then scans all relevant rows in the second size k instance table I_{c_2} (AC in this case) with the same prefix instance. Specifically, thread 0 will scan relevant row instances of AB starting from prefix A_1, while thread 1 will scan relevant row instances starting from prefix A_2. The memory transfer between host and device is similar to grid-based approach, except that there are two size k instance tables being transferred from host to device to do the join. We used one dimensional GPU blocks with 1024 threads in each block for both approaches.

3.4 Further GPU Optimization

GPU Profiling and Bottleneck Analysis: After profiling our recent GPU implementations [44], we found that the problem is a memory-bound with a significant amount of memory allocation and transfer. The bottleneck is due to the limited amount of available device memory, as well as the significant growth of the size of instance tables with pattern cardinality when event instances are spatially dense. The memory transfer between host and device in our conference GPU implementations is summarized in Table 1. Based on the identified memory bottleneck, we further optimize our recent GPU implementation with refined memory management, including reducing unnecessary memory transfer between host and device, replacing a large number of small memory transfer into a small number of batch memory transfer, and reducing excessive use of pinned memory. We also design a new kernel function for the preprocessing, and redesign the filtering kernel (e.g., utilizing GPU shared memory, reducing kernel calls). GPU optimization procedures in different phases are summarized in Table 2.

TABLE 1
Memory Data Structures Transfered Between Host and Device

Phases	Host to Device	Device to Host
Preprocessing	* Cell to Instance Index * CountMap	N/A
Filter	* Candidate Pattern c	Upper Bound of PI
Refine	**Round R1**: * Candidate Pattern c (size k+1) * Instance Table of size k **Round R2**: *Slot Counts	**Round R1**: * Participation Index (PI) of c * Slot Counts **Round R2**: *Instance Table of c

1. **Reducing unnecessary host-to-device and device-to-host memory transfers**: In the preprocessing phase, our

TABLE 2
Summary of GPU Optimization in different algorithm phases

	1. Reducing Unnecessary memory transfer	2. Batch Memory transfer/allocation	3. Elimination of pinned memory	4.Adding New Kernel Function	5. Redesigning Kernel Functions
Preprocessing	✔		✔	✔	
Filter		✔	✔		✔
Refine		✔	✔		

recent implementation conducts preprocessing in CPU and transfer results (CountMap and Cell to Instance Index) from host to device. Preprocessing may take up a significant ratio of total time cost. Thus, we design a new kernel function for preprocessing (details are in item 4 below), so results were directly generated in GPU memory without the need of memory transfer.

2. **Replacing frequent small memory transfers with batch memory transfers or allocation**: In the filtering phase, our recent implementation launches the filtering kernels for each candidate pattern c in the iteration. We need to transfer the candidate pattern c from host to device before kernel launch, and transfer upper bound of PI back from device to host after kernel launch. Since the number of candidate patterns in each iteration (cardinality) is combinatorial, there are frequent small memory transfers. To address this, we redesign the filtering kernel (details are in item 5 below) so that upper bounds of all candidate patterns in each iteration are computed all together in only two kernel launches. Thus, the candidate patterns and their upper bounds of PI are transferred all together once (batch mode) between host and device. In the refinement phase, our recent implementation allocates a fix amount of global memory for slot count arrays for each candidate pattern. We reduce that by allocating a bigger global memory buffer only once, and manage the buffer to write slot count arrays for different candidate patterns contiguously. This uses more memory but reduce the number of memory allocations.

3. **Reducing the use of pinned memory**: In all phases, our recently GPU implementation used pinned memory on the host. We used pinned memory to reduce the cost of memory transfer. However, since pinned memory is shared between operating system and applications and it is a limited resource. Therefore, excessive allocation of this memory (e.g., the big instance tables up to several gigabytes) may lead to degrading the performance of the system. We optimized pinned memory usages by replacing unnecessary cudaMallocHost allocations with system malloc API.

4. **Adding new kernel function for preprocessing**: The grid processing is an embarrassingly parallel problem and if it is efficiently implemented over GPU, it can be achieved at least an order of magnitude speedup over the CPU implementation. In our recent GPU versions, the preprocessing phase of applying uniform grid over data is handled over CPU. Although the preprocessing time is not large, by designing a kernel to do the computation over GPU we are able to gain significant speedup for this part over our recent versions. Each thread handles one feature instance to calculate which cell that instance belongs to. Therefore, total number of threads is equal to total number of feature instances. Since the input vector (x and y coordinate of feature instances) is linear, we use a linear kernel configuration to provide coalesce memory access in each block.

5. **Redesigning filtering kernel function**: As discussed earlier, the number of kernel function calls in the filtering phase of our recent implementations is combinatorial to the number of features. There are two kernel launches for each candidate pattern (one launch to compute $PCountMap$, and another launch to compute the upper bound of PI). For example, if there are 13 features, there will be 156 and 572 kernel calls for filtering all the degree-2 and degree-3 patterns respectively. The number of kernel calls can potentially go as high as 3432 for these features. To address these issues, we redesigned the GPU-based filter algorithm such that only two kernel calls are needed to filter out all the uninteresting patterns of the same cardinality (in the same iteration). In particular, we assign each three dimensional GPU block to count the number of feature instances of a given candidate pattern in one quadruplet with X-dimension size of block defined by the number of features and Y and Z-dimensions form a 2×2 strucure for processing each cell in quadruplet. Then, by using GPU shared memory, the first kernel aggregates the results within each GPU block (quadruplet) and the second kernel simply eliminates all the patterns that are below the given threshold. In the first kernel, each thread counts the number of instances of one feature within each GPU block. Then, each GPU block determines if all the features presented in the pattern have non-zero counts in the quadruplet using shared memory signaling. Therefore, the kernel configuration is a function of grid structure, the number of features and the number of candidate patterns for any given degree. In the second kernel, the algorithm uses a reduction tree for each candidate pattern in one 1-dimensional GPU block to find out if the minimum feature instance count passes the threshold. The block size of this kernel (only x-dimension) is a function of the number of features.

In the above, we discussed our further GPU optimization proposed in this paper based on refined memory management and kernel redesign. We acknowledge that there is still space of further improvement, such as redesigning refinement kernels, which could be done in future work.

4 THEORETICAL ANALYSIS

This section conducts theoretical analysis on proposed algorithms.

4.1 Correctness and Completeness

Lemma 1. *The participation index of a colocation pattern in the cell-aggregate-based filter is an upper bound of the true participation index value.*

Proof. The proof is based on the following fact. We create an upper bound to the true number of neighboring points in

neighboring cells (quadruplet) by assuming that all pairs of points of neighboring cells are within the distance threshold, which coincides with the cell size. Of course, some of them will not, but it is impossible for points not within neighboring cells to be neighboring with respect to the distance threshold. □

Theorem 1. *The cell aggregate based upper bound filter is correct and complete.*

Proof. The proof is based on Lemma 1. The algorithm is complete (it does not mistakenly prune out any prevalent pattern) due to the upper bound property. The algorithm is correct since it computes the exact participation index of a candidate pattern if it passes the upper bound filter. □

4.2 Computational Cost Analysis

Theorem 2. *The time complexity of the cell aggregate based upper bound filter is $O(|c| \cdot N)$, where $|c|$ is pattern size, N is the number of cells.*

Proof. For each cell, the algorithm needs to scan the number of instances for each feature of c in the quadruplet. Thus, the total time complexity is $O(|c| \cdot N)$. □

Theorem 3. *The time complexity of the grid-based refinement is $O(|I_{c_k}| \cdot k \cdot n)$, where c_k is a size k prefix pattern of candidate pattern c, $|I_{c_k}|$ is the number of instances for pattern c_k, n is the number of instances for the last feature in c in the neighboring cells.*

Proof. For each pattern instance of c_k ($I_{c_k}[i]$), the algorithm needs to scan all instances of the last feature type in c in the neighboring cells. For each of the n instances, the algorithm computes its distance to each feature instance in $I_{c_k}[i]$. Thus, the total cost is $O(|I_{c_k}| \cdot k \cdot n)$. □

Theorem 4. *The time complexity of prefix-join based refinement is $O(|I_{c_1}| \cdot |I_{c_2}| \cdot k/|I_{c[1]}|)$, where $k+1$ is the size of pattern c, $|I_{c_1}|$ and $|I_{c_2}|$ are the number of instances for two size k sub-patterns that are used to generate instances of c, $|I_{c[1]}|$ is the number of unique instances of the first feature type in c.*

Proof. For each instance in I_{c_1} ($I_{c_1}[i]$), the algorithm scans all instances of I_{c_2} that share the same first instance with $I_{c_1}[i]$ (i.e., instances in the prefix hash bucket). During the scanning, the algorithm conducts join operations, which costs $O(k)$. Thus, the total cost is $O(|I_{c_1}| \cdot |I_{c_2}| \cdot k/|I_{c[1]}|)$. Here we assume that instances of I_{c_2} are evenly distributed in the hash buckets based on their first feature instances. □

Is the colocation mining problem memory-bound or compute-bound? As mentioned earlier, our implementation is heterogeneous over GPU and CPU. Thus, there are many memory transfers between device and host that adds up to the memory operations delay. Moreover, GPU kernels that are launched for preprocessing, filter and refinement phases are memory bound. In other words, the arithmetic intensity of these kernels, or the ratio of arithmetic operations to memory operations (compute-to-memory ratio) is low, below the balanced compute-to-memory ratio of our GPU hardware.

5 Evaluation

The goals of our evaluation are to:
• Compare the speedup of GPU implementations with CPU multi-core (OpenMP) implementation.
• Compare the speedup of our optimized journal GPU implementations with conference versions.
• Compare the time costs of different algorithm components in different parallel implementations.
• Evaluate of the effect of data sizes and prevalence thresholds on speedups.

Experiment Setup: We performed GPU experiments on Bridges cluster located at Pittsburgh Supercomputing Center (PSC). We used one *NVIDIA Tesla P100* GPU on a GPU Shared Memory node. Nvidia $P100$ GPU has 16 *GB* of the main memory and it provides $3,584$ *CUDA cores* operating at 1480 MHz base clock that provides 5.3 *TFLOPS* of double precision floating point calculations. Furthermore, we ran CPU sequential and multi-core (OpenMP) experiments on Dell Precision Tower 7910 equipped with Intel(R) Xeon(R) CPU E5-2687w v4 @ 3.00GHz, 64GB main memory, and Ubuntu operating system. Algorithms were implemented in C++ and CUDA and compiled using g++ (without optimization flags) and nvcc compilers. For each experiment, we measure the average time cost of five runs. We compared the following candidate colocation algorithm implementations.

CPU Sequential: this is the baseline by Huang et al. [1] (multi-resolution filter, grid-based instance table generation for size $k = 2$, and sort-merge based instance table generation for size $k > 2$).
CPU Multi-core (OpenMP): We implemented our recent parallelization idea of grid-join based refinement in [44] in multi-core CPU (OpenMP omp compiler directive for multi-threading, through "-fopenmp" flag in g++). We used the grid-join based in OpenMP due to its superior performance over prefix-join in previous results.
GPU-prefix-join: This is our conference paper GPU version with cell aggregate filter and prefix join based refinement.
GPU-grid-join: This is our conference paper GPU version with cell aggregate filter and grid-based join based refinement.
GPU-grid-join+: This is the implementation with further GPU optimization proposed in this paper.

Dataset description: We used Seattle Police Department 911 Incident Response [45] dataset with 1.48 million 911 incident instances with longitude, latitude and event description recorded from 2010 to 2017. We extracted the incident reports in 2012 within a partition of minimum longitude 122 26 18.24″ W, minimum latitude 47 26′ 43.44″ N and maximum longitude 122 12′ 43.2″ W, maximum latitude 47 40′ 21″ N as real dataset. It contains 165,458 911 incident records and 13 crime types as shown in the Table 3.

The synthetic data is generated similarly to [1]. We first chose a study area size of 10000×10000, a neighborhood distance threshold (also the size of a grid cell) of 10, a maximal pattern cardinality of 5, and the number of maximal colocation patterns as 2. The total number of features was 12 (5×2 plus 2 additional noise features). We then generated

TABLE 3
Crime Types in Real Dataset

Crime Type	Number of Instances
Auto Theft	3776
Accident Investigation	12600
Burglary	5058
Car Prowl	6126
Disturbance	27621
Residential Burglary	7499
Liquor Violation	14003
Narcotics	5055
Theft	6160
Shoplifting	4977
Suspicious Person	32565
Traffic Violation	35365
Trespassing	4653

a *number of instances* for each maximal colocation pattern. Their locations were randomly distributed to different cells according to the clumpiness (i.e., the number of overlaying colocation instances within the same neighborhood, higher clumpiness means larger instance tables). In our experiments, we varied the number of instances and clumpiness to test sensitivity.

5.1 Results on synthetic data

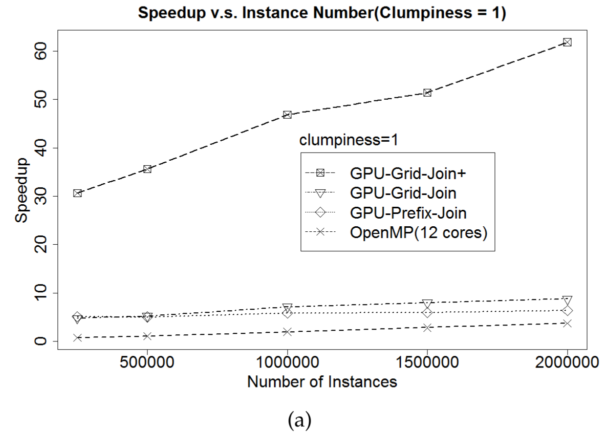
5.1.1 *Effect of the number of instances*

We conducted this experiment with two different parameter settings. For both settings, the minimum participation index threshold was 0.5. In the first setting, we set the clumpiness to 1 (very low clumpiness), and varied the number of feature instances as 250,000, 500,000, 1,000,000, 1,500,000 and 2,000,000. Results are summarized in Figure 5(a). With the number of event instances increasing, the speedups of all parallel methods increase due to increasing computational load. We find that our recent GPU implementations (GPU-Grid-Join and GPU-Prefix-Join) outperform OpenMP parallelization. Among all parallel methods, our journal method (GPU-Grid-Join+) persistently outperforms the others (30 to 60 versus below 10), due to our proposed further GPU optimizations.

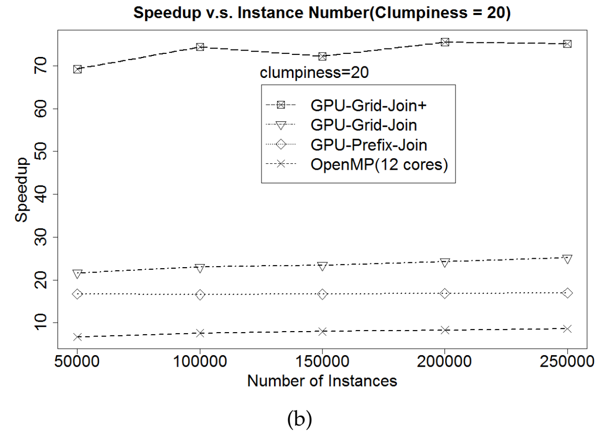
In the second setting, we set the clumpiness value as 20, and varied the number of feature instances as 50,000, 100,000, 150,000, 200,000, and 250,000. The number of feature instances were set smaller in this setting due to the fact that given the same number of feature instances, a higher clumpiness value results in far more colocation pattern instances but we only have limited memory. The results are summarized in Figure 5(b). With the number of instances increasing, the speedups of different parallel methods remain relatively stable. The reason is that on highly clumpy data, the main computational bottleneck is the significantly growing intermediate instance table size, instead of the number of event instances. Our journal method persistently outperforms the other methods (over 70 versus less than 30).

5.1.2 *Effect of Clumpiness*

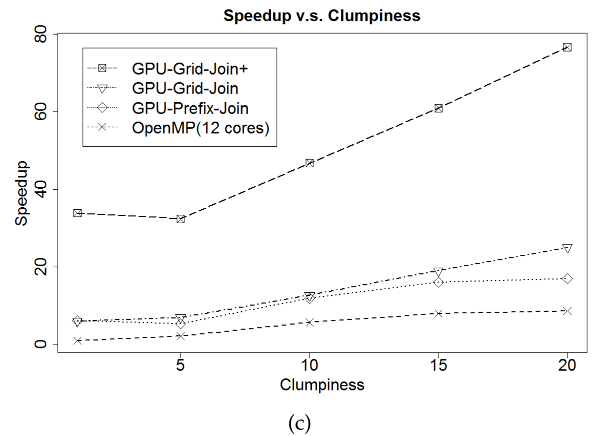
We set the number of instances to 250k, and the prevalence threshold to 0.5. We varied the clumpiness value as 1, 5, 10, 15, and 20. The results are summarized in Figure 5(c).



(a)



(b)



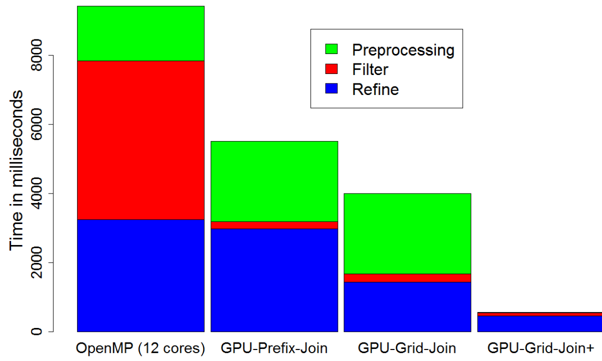
(c)

Fig. 5. Comparison of parallel methods on synthetic datasets: (a) effect of the number of instances with clumpiness as 1 (b) effect of the number of instances with clumpiness as 20 (c) effect of clumpiness with the number of instances as $250,000$

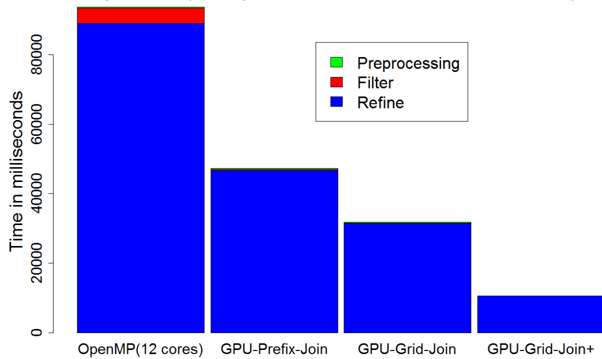
With the clumpiness increasing, the speedups of all parallel method increase, due to the increasing computational load. The speedup of our journal method increases faster and is persistently higher than the others, primarily due to improved host and device memory management under increasing intermediate data sizes.

5.1.3 *Comparison on Preprocessing, Filtering and Refinement Phases*

We also compared the computational time costs of preprocessing, filtering and refinement phases of all parallel

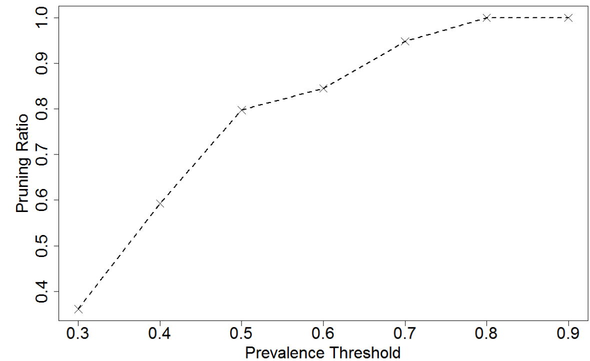


(a)

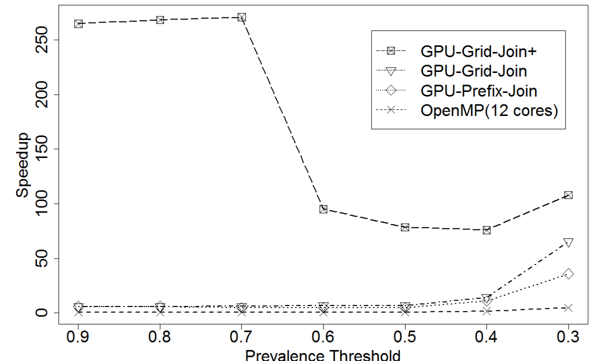


(b)

Fig. 6. Comparison of methods in their preprocessing, filter and refinement time costs on synthetic data: (a) when clumpiness is 1 and the number of instances is $2,000,000$ (b) when clumpiness is 20 and the number of instances is $250,000$



(a)



(b)

Fig. 7. Results on real world dataset: (a) effect of prevalence threshold on pruning ratio (b) speedup of different parallel methods on varying prevalence thresholds

methods in the results of Figure 5(a-b). Results are shown in Figure 6. Figure 6(a) shows results on input data with clumpiness as 1 and the number of instances as 2 million. In the results, our recent GPU implementations have slightly higher preprocessing cost than OpenMP (due to the extra memory transfer from host to device after preprocessing), but significantly lower costs in filtering (due to the utilization of GPU cores). Our journal method (GPU-Grid-Join+) significantly reduces the preprocessing cost due to parallelization on GPU, reducing memory transfer, as well as the refinement cost due to batch memory allocation and elimination of improper usage of pinned memory. Figure 6(b) shows the results on the data with clumpiness as 20 and the number of instances as $250,000$. In the results, all methods have tiny preprocessing and filtering costs compared with the refinement costs. The refinement costs in GPU implementation is significantly lower than the OpenMP method. Our journal method (GPU-Grid-Join+) shows the minimum refinement cost due to batch memory allocation and elimination of improper usage of pinned memory.

5.2 Results on real world dataset

5.2.1 Effect of Minimum Participation Index Threshold

We fixed the distance threshold as 10 meters and varied the prevalence thresholds from 0.3 to 0.9 by a step size of 0.1 (we did not chose thresholds lower than 0.3, because there would be too many instance tables exceeding our host memory capacity). Results are summarized in Figure 7. Figure 7(a) shows the effect of prevalence threshold on pruning ratios. With the threshold getting higher, the pruning ratio also gets higher because upper bounds of candidate patterns are more likely to be below the thresholds. Figure 7(b) shows the speedups of different parallel methods under different prevalence thresholds. With the threshold decreasing, the speedups of OpenMP and our recent GPU implementations all slightly increase due to the increasing computational load. However, the speedup of our journal method (GPU-Grid-Join+) with further GPU optimization first stay very high (as high as 250), then decreases, and finally slightly increases. The reason is that our journal method incorporates redesign of the filtering kernel. When the threshold is higher, the filtering cost takes the vast majority of the total time costs, and the effect of GPU optimization on filtering kernel is the most significant. Such effect decreases as the threshold gets lower (filtering phase no longer takes the vast majority of the time cost). Finally, the speedup of the journal method slightly increases due to increasing overall computational load.

5.2.2 Comparison of preprocessing, filter and refinement

The distance threshold was 10 meters, and the prevalent threshold was 0.5. Results are shown in figure 8, which

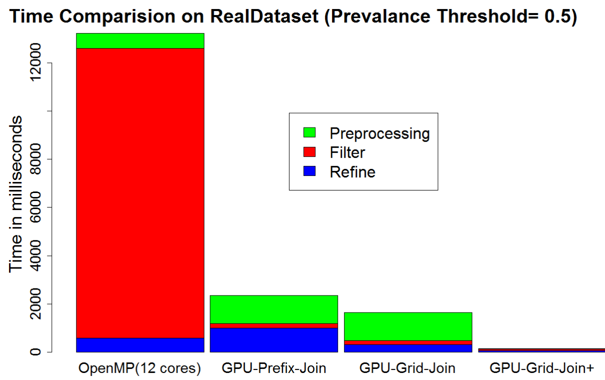


Fig. 8. Comparison of parallel methods on their preprocessing, filter and refinement time costs on real data

looks similar to the results on synthetic data with clumpiness as 1. The OpenMP implementation has a significant cost in the filtering phase, while the GPU implementation has significantly lower filtering costs. Our journal method (GPU-Grid-Join+) has significantly lower preprocessing cost and refinement cost compared with our recent GPU implementations.

5.2.3 Output Crime Colocation Patterns

We reran our colocation mining algorithm on real dataset with prevalence threshold of 0.3 and distance threshold of 10 meters. The colocation patterns generated with highest cardinality (four features) are shown in the Table 4. For example, Car Prowl, Disturbance, Theft, and Suspicious Person are frequently located close together. The reason might be that thefts were often conducted by suspicious persons after car prowl, etc. However, we would like to acknowledge that fully interpreting the output patterns require collaboration with domain experts on crime analysis, which is beyond the scope of this paper.

TABLE 4
Crime Colocation Patterns based on Real Dataset

Colocation Patterns
{Car Prowl, Disturbances, Theft, Suspicious Person}
{Car Prowl, Disturbances, Suspicious Person, Traffic Violation}
{Disturbances, Liquor Violation, Narcotics, Theft}
{Disturbances, Liquor Violation, Suspicious Person, Trespass}
{Disturbances, Liquor Violation, Suspicious Person, Trespass}
{Disturbances, Theft, Suspicious Person, Traffic Violation}
{Disturbances, Theft, Suspicious Person, Trespass}

6 Discussion

The colocation mining problem can potentially be generalized from the Euclidean space to the spatial network space. In the spatial network space, neighborhood definition will be based on network distance, e.g., shortest path distance. The spatial statistics basis will be network K-function.

7 Conclusion and Future Work

This paper investigates GPU colocation mining algorithms. We introduce our recently proposed cell-aggregate-based upper bound filter, which is easier to implement on GPU and less sensitive to data clumpiness compared with the existing multi-resolution filter. We also introduce two recent GPU implementations of refinement algorithms. We proposed further GPU optimization based on identified memory bottleneck, and provide theoretical analysis on the correctness and completeness, as well as time cost. Results on both real world data and synthetic data on various parameter settings show that proposed GPU algorithms optimization significantly improves the speedups, and GPU implementation are more scalable than OpenMP implementation.

In future work, we will explore further GPU optimizations, e.g., redesign the refinement kernels. We will also evaluate methods on different GPU models. We may explore GPU implementations on a cluster of nodes (integrating GPU implementation with the MapReduce frameworks). We may also explore other distance measures and the scenario of streaming input data.

References

[1] Y. Huang, S. Shekhar, and H. Xiong, "Discovering colocation patterns from spatial data sets: a general approach," *IEEE Transactions on Knowledge and data engineering*, vol. 16, no. 12, pp. 1472–1485, 2004.

[2] R. Agrawal, R. Srikant *et al.*, "Fast algorithms for mining association rules," in *Proc. 20th int. conf. very large data bases, VLDB*, vol. 1215, 1994, pp. 487–499.

[3] P. Phillips and I. Lee, "Crime analysis through spatial areal aggregated density patterns," *Geoinformatica*, vol. 15, no. 1, pp. 49–74, 2011.

[4] S. E. Donovan, G. J. Griffiths, R. Homathevi, and L. Winder, "The spatial pattern of soil-dwelling termites in primary and logged forest in sabah, malaysia," *Ecological Entomology*, vol. 32, no. 1, pp. 1–10, 2007.

[5] P. Haase, "Spatial pattern analysis in ecology based on ripley's k-function: Introduction and methods of edge correction," *Journal of vegetation science*, vol. 6, no. 4, pp. 575–582, 1995.

[6] A. Sadilek, H. A. Kautz, and V. Silenzio, "Predicting disease transmission from geo-tagged micro-blog data." in *AAAI*, 2012.

[7] G. M. Vazquez-Prokopec, D. Bisanzio, S. T. Stoddard, V. Paz-Soldan, A. C. Morrison, J. P. Elder, J. Ramirez-Paredes, E. S. Halsey, T. J. Kochel, T. W. Scott *et al.*, "Using gps technology to quantify human mobility, dynamic contacts and infectious disease dynamics in a resource-poor urban environment," *PloS one*, vol. 8, no. 4, p. e58802, 2013.

[8] L.-A. Tang, Y. Zheng, J. Yuan, J. Han, A. Leung, W.-C. Peng, and T. L. Porta, "A framework of traveling companion discovery on trajectory data streams," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 5, no. 1, p. 3, 2013.

[9] K. Koperski and J. Han, "Discovery of spatial association rules in geographic information databases," in *International Symposium on Spatial Databases*. Springer, 1995, pp. 47–66.

[10] Y. Morimoto, "Mining frequent neighboring class sets in spatial databases," in *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2001, pp. 353–358.

[11] X. Xiao, X. Xie, Q. Luo, and W.-Y. Ma, "Density based co-location pattern discovery," in *Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems*. ACM, 2008, p. 29.

[12] X. Zhang, N. Mamoulis, D. W. Cheung, and Y. Shou, "Fast mining of spatial collocations," in *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2004, pp. 384–393.

[13] J. S. Yoo, S. Shekhar, J. Smith, and J. P. Kumquat, "A partial join approach for mining co-location patterns," in *Proceedings of the 12th annual ACM international workshop on Geographic information systems*. ACM, 2004, pp. 241–249.

[14] J. S. Yoo and S. Shekhar, “A joinless approach for mining spatial colocation patterns,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 18, no. 10, pp. 1323–1337, 2006.

[15] P. Boinski and M. Zakrzewicz, “Collocation pattern mining in a limited memory environment using materialized icpi-tree,” in *International Conference on Data Warehousing and Knowledge Discovery*. Springer, 2012, pp. 279–290.

[16] P. Mohan, S. Shekhar, J. A. Shine, J. P. Rogers, Z. Jiang, and N. Wayant, “A neighborhood graph based approach to regional co-location pattern discovery: A summary of results,” in *Proceedings of the 19th ACM SIGSPATIAL international conference on advances in geographic information systems*. ACM, 2011, pp. 122–132.

[17] S. Wang, Y. Huang, and X. S. Wang, “Regional co-locations of arbitrary shapes,” in *International Symposium on Spatial and Temporal Databases*. Springer, 2013, pp. 19–37.

[18] B. Liu, L. Chen, C. Liu, C. Zhang, and W. Qiu, “Rcp mining: Towards the summarization of spatial co-location patterns,” in *International Symposium on Spatial and Temporal Databases*. Springer, 2015, pp. 451–469.

[19] M. Celik, J. M. Kang, and S. Shekhar, “Zonal co-location pattern discovery with dynamic parameters,” in *Data Mining, 2007. ICDM 2007. Seventh IEEE International Conference on*. IEEE, 2007, pp. 433–438.

[20] S. Barua and J. Sander, “Statistically significant co-location pattern mining,” 2015.

[21] J. S. Yoo and M. Bow, “Mining top-k closed co-location patterns,” in *Spatial Data Mining and Geographical Knowledge Services (ICSDM), 2011 IEEE International Conference on*. IEEE, 2011, pp. 100–105.

[22] Y. Huang, H. Xiong, S. Shekhar, and J. Pei, “Mining confident co-location rules without a support threshold,” in *Proceedings of the 2003 ACM symposium on Applied computing*. ACM, 2003, pp. 497–501.

[23] H. Xiong, S. Shekhar, Y. Huang, V. Kumar, X. Ma, and J. S. Yoc, “A framework for discovering co-location patterns in data sets with extended spatial objects,” in *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 2004, pp. 78–89.

[24] D. Agarwal and S. K. Prasad, “Lessons learnt from the development of gis application on azure cloud platform,” in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 2012, pp. 352–359.

[25] S. Puri, D. Agarwal, X. He, and S. K. Prasad, “Mapreduce algorithms for gis polygonal overlay processing,” in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE, 2013, pp. 1009–1016.

[26] S. K. Prasad, M. McDermott, S. Puri, D. Shah, D. Aghajarian, S. Shekhar, and X. Zhou, “A vision for gpu-accelerated parallel computation on geo-spatial datasets,” *SIGSPATIAL Special*, vol. 6, no. 3, pp. 19–26, 2015.

[27] S. K. Prasad, D. Aghajarian, M. McDermott, D. Shah, M. Mokbel, S. Puri, S. J. Rey, S. Shekhar, Y. Xe, R. R. Vatsavai *et al.*, “Parallel processing over spatial-temporal datasets from geo, bio, climate and social science communities: A research roadmap,” in *Big Data (BigData Congress), 2017 IEEE International Congress on*. IEEE, 2017, pp. 232–250.

[28] S. K. Prasad, M. McDermott, X. He, and S. Puri, “Gpu-based parallel r-tree construction and querying,” in *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*. IEEE, 2015, pp. 618–627.

[29] D. Aghajarian, S. Puri, and S. Prasad, “Gcmf: an efficient end-to-end spatial join system over large polygonal datasets on gpgpu platform,” in *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 2016, p. 18.

[30] D. Aghajarian and S. K. Prasad, “A spatial join algorithm based on a non-uniform grid technique over gpgpu,” in *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 2017, p. 56.

[31] J. Zhang and S. You, “Speeding up large-scale point-in-polygon test based spatial join on gpus,” in *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*. ACM, 2012, pp. 23–32.

[32] S. You, J. Zhang, and L. Gruenwald, “Parallel spatial query processing on gpus using r-trees,” in *Proceedings of the 2nd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*. ACM, 2013, pp. 23–31.

[33] J. Zhang, S. You, and L. Gruenwald, “Large-scale spatial data processing on gpus and gpu-accelerated clusters,” *SIGSPATIAL Special*, vol. 6, no. 3, pp. 27–34, 2015.

[34] J. S. Yoo, D. Boulware, and D. Kimmey, “A parallel spatial co-location mining algorithm based on mapreduce,” in *Big Data (BigData Congress), 2014 IEEE International Congress on*. IEEE, 2014, pp. 25–31.

[35] W. Andrzejewski and P. Boinski, “Gpu-accelerated collocation pattern discovery,” in *East European Conference on Advances in Databases and Information Systems*. Springer, 2013, pp. 302–315.

[36] ——, “A parallel algorithm for building icpi-trees,” in *East European Conference on Advances in Databases and Information Systems*. Springer, 2014, pp. 276–289.

[37] J. S. Yoo and D. Boulware, “Incremental and parallel spatial association mining,” in *Big Data (Big Data), 2014 IEEE International Conference on*. IEEE, 2014, pp. 75–76.

[38] W. Andrzejewski and P. Boinski, “Parallel gpu-based plane-sweep algorithm for construction of icpi-trees,” *Journal of Database Management (JDM)*, vol. 26, no. 3, pp. 1–20, 2015.

[39] M. J. Zaki, “Parallel and distributed association mining: A survey,” *IEEE concurrency*, vol. 7, no. 4, pp. 14–25, 1999.

[40] L. Zhou, Z. Zhong, J. Chang, J. Li, J. Z. Huang, and S. Feng, “Balanced parallel fp-growth with mapreduce,” in *Information Computing and Telecommunications (YC-ICT), 2010 IEEE Youth Conference on*. IEEE, 2010, pp. 243–246.

[41] N. Li, L. Zeng, Q. He, and Z. Shi, “Parallel implementation of apriori algorithm based on mapreduce,” in *Software Engineering, Artificial Intelligence, Networking and Parallel & Distributed Computing (SNPD), 2012 13th ACIS International Conference on*. IEEE, 2012, pp. 236–241.

[42] M.-Y. Lin, P.-Y. Lee, and S.-C. Hsueh, “Apriori-based frequent itemset mining algorithms on mapreduce,” in *Proceedings of the 6th international conference on ubiquitous information management and communication*. ACM, 2012, p. 76.

[43] X. Y. Yang, Z. Liu, and Y. Fu, “Mapreduce as a programming model for association rules algorithm on hadoop,” in *Information Sciences and Interaction Sciences (ICIS), 2010 3rd International Conference on*. IEEE, 2010, pp. 99–102.

[44] A. M. Sainju and Z. Jiang, “Grid-based colocation mining algorithms on gpu for big spatial event data: A summary of results,” in *International Symposium on Spatial and Temporal Databases*. Springer, 2017, pp. 263–280.

[45] City of Seattle, Department of Information Technology, Seattle Police Department, “Seattle Police Department 911 Incident Response,” https://data.seattle.gov/Public-Safety/Seattle-Police-Department-911-Incident-Response/3k2p-39jp.

Arpan Man Sainju is currently a Ph.D. student in the department of Computer Science at the University of Alabama, Tuscaloosa. He received his B.E. degree in computer engineering from Tribhuvan University, Nepal. He has worked on spatial big data algorithms on GPU. His research interests include data mining, spatial database, big data analytics, parallel computing and deep learning.

Danial Aghajarian currently is a Phd student in computer science department at Georgia State University. He received his bachelor's and master's degrees in computer engineering from Isfahan University of Technology (IUT) and Iran University of Science and Technology (IUST) respectively. His current research interests include performance evaluation and modeling of parallel/distributed computing systems, parallel algorithm design over Geo-Spatial data over heterogeneous distributed computing platforms equipped with GPUs.