Grid-Based Colocation Mining Algorithms on GPU for Big Spatial Event Data: A Summary of Results

Arpan Man Sainju and Zhe Jiang(✉)

Department of Computer Science, The University of Alabama,
Tuscaloosa, AL 35487, USA
asainju@crimson.ua.edu, zjiang@cs.ua.edu

Abstract. This paper investigates the colocation pattern mining problem for big spatial event data. Colocation patterns refer to subsets of spatial features whose instances are frequently located together. The problem is important in many applications such as analyzing relationships of crimes or disease with various environmental factors, but is computationally challenging due to a large number of instances, the potentially exponential number of candidate patterns, and high computational cost in generating pattern instances. Existing colocation mining algorithms (e.g., Apriori algorithm, multi-resolution filter, partial join and joinless approaches) 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. 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. Another work proposes a parallel colocation mining algorithm on GPU based on the iCPI tree and the joinless approach, but assumes that the number of neighbors for each instance is within a small constant, and thus may be inefficient when instances are dense and unevenly distributed. To address these limitations, we propose a grid-based GPU colocation mining algorithm that includes a novel cell aggregate based upper bound filter, and two refinement algorithms. We prove the correctness and completeness of proposed GPU algorithms. Preliminary results on both real world data and synthetic data show that proposed GPU algorithms are promising with over 30 times speedup on up to millions of instances.

1 Introduction

Given a set of spatial features and their instances, co-location mining aims to find subsets of features whose instances are frequently located together. Examples of colocation patterns include symbiotic relationships between species such as Nile Crocodiles and Egyptian Plover, as well as environmental factors and disease events (e.g., air pollution and lung cancer).

Societal applications: Colocation mining is important in many applications that aim to find associations between different spatial events or factors.

M. Gertz et al. (Eds.): SSTD 2017, LNCS 10411, pp. 263–280, 2017.
DOI: 10.1007/978-3-319-64367-0_14

For example, in public safety, law enforcement agencies are interested in finding relationships between different crime event types and potential crime generators. In ecology, scientists analyze common spatial footprints of various species to capture their interactions and spatial distributions. In public health, identifying relationships between human disease and potential environmental causes is an important problem. In climate science, colocation patterns help reveal relationships between the occurrence of different climate extreme events. In location based service, colocation patterns help identify travelers that share the same favourite locations to promote effective tour recommendation.

Challenges: Mining colocation patterns from big spatial event data poses several computational challenges. First, in order to evaluate if a candidate colocation pattern is prevalent, we need to generate its instances. This is computationally expensive due to checking spatial neighborhood relationships between different instances, particularly when the number of instances is large and instances are clumpy (e.g., many instances are within the same spatial neighborhoods). 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. Finally, the distribution of event instances in the space may be uneven, making it hard to design parallel data structure and algorithms.

Related work: Colocation pattern mining has been studied extensively in the literature, including early work on spatial association rule mining [1,2] and colocation patterns based on event-centric model [3]. Various algorithms have been proposed to efficiently identify colocation patterns, including Apriori generator and multi-resolution upper bound filter [3], partial join [4] and joinless approach [5], iCPI tree based colocation mining algorithms [6]. There are also works on identifying regional [7–9] or zonal [10] colocation patterns, and statistically significant colocation patterns [11–13], top-K prevalent colocation patterns [14] or prevalent patterns without thresholding [15]. Existing algorithms are mostly sequential, and can be insufficient when the number of event instances is very large (e.g., several millions). Recently, parallel colocation mining algorithms have been proposed based on the Map-reduce framework [16] to handle 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. Another work proposes a GPU based parallel colocation mining algorithm [17] using iCPI tree [18–20] 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.

To address limitations of related work, we propose GPU colocation mining algorithms based on a grid index, including a cell-aggregate-based upper bound filter and two refinement algorithms. Proposed cell-aggregate-based filter is easier to implement on GPU 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. We use a GPU platform due to its better energy efficiency and pricing compared to Map-reduce based clouds.

Contributions: We make the following contributions in the paper: (1) We designed and implemented parallel colocation mining algorithms on GPU, including a novel cell-aggregate based upper bound filter that is easier to implement on GPU and also insensitive to pattern clumpiness (i.e., number of colocation instances within the same neighborhood), two parallel refinement algorithms based on prefix-based HashJoin and grid search; (2) We provided theoretical analysis of the correctness and completeness of proposed algorithms; (3) We conducted extensive experimental evaluations on both real world and synthetic data with various parameter settings. Preliminary results show that proposed GPU algorithms are promising with over 30 times speedup on up to millions of instances.

Scope and outline: We focus on spatial colocation patterns defined by the event-centric models [3]. Other colocation definitions such as Voronoi diagram based are beyond our scope. We also assume the underlying space is Euclidean space. Also, in this paper, we are only concerned with the comparison of computational performance of various colocation mining algorithms.

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 proposed GPU colocation pattern mining algorithms, and analyzes the theoretical properties of algorithm correctness and completeness. Section 4 summarizes our experimental evaluation of proposed algorithms on both synthetic datasets and a real world dataset. Section 5 discusses memory bottleneck issues in our approach. Section 6 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 [3].

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 Fig. 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 Fig. 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 Fig. 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 *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 [3].

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 Fig. 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 Fig. 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 [3].

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 provides a problem example. The input data contains 12 instances of 3 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 mining is similar to association rule mining in market basket analysis [21], but is different in that there are no given "transactions" in continuous space. Generation colocation instance tables ("transactions") is the most computationally intensive part.

3 Proposed Approach

This section introduces our proposed GPU colocation mining algorithm. We start with the overview of algorithm structure, and then describe the main part for parallel algorithms implemented in GPU. We prove the correctness and completeness of proposed algorithm.

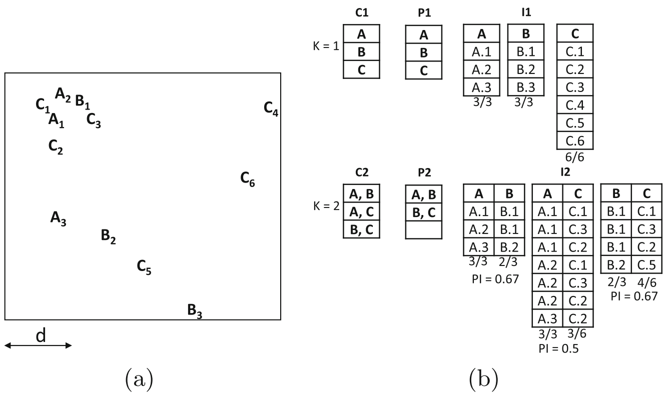

C1 A2 B1 A1 C3 C2 C4 C6 A3 B2 C5 B3
d
(a)
C1 P1 I1
K = 1
A B C
A.1 A.2 A.3 3/3
B.1 B.2 B.3 3/3
C.1 C.2 C.3 C.4 C.5 C.6 6/6
C2 P2 I2
K = 2
A, B A, C B, C
A.1 B.1, A.2 B.1, A.3 B.2 3/3 2/3 PI = 0.67
A.1 C.1, A.1 C.3, A.1 C.2, A.2 C.1, A.2 C.3, A.2 C.2, A.3 C.2 3/3 3/6 PI = 0.5
B.1 C.1, B.1 C.3, B.1 C.2, B.2 C.5 2/3 4/6 PI = 0.67
(b)

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

3.1 Algorithm Overview

The overall structure of our algorithm is similar to the one proposed by Huang et al. in 2004 [3]. The novelty of our algorithm is that we design a novel upper-bound filter based on aggregated counts of feature instances in grid cells. Compared with the existing multi-resolution filter [3], our upper bound filter is easier to parallelize on GPU and does not rely on the assumption that colocation instances are clumpy into a small number of cells.

The overall structure of our algorithm is shown in Algorithm 1. The algorithm identifies all prevalent colocation patterns iteratively. Candidate colocation patterns and their instance tables of cardinality $k+1$ are generated, based on prevalent patterns and their instance tables of cardinality k. Each candidate pattern of cardinality $k+1$ is then evaluated based on the participation index computed from its instance table. For cardinality $k = 1$, prevalent colocation patterns simply consist of the set of input features, and their instance tables are the instance lists for each feature (step 1 in Algorithm 1). Step 2 generates candidate patterns C_{k+1} of size $k+1$ based on prevalent patterns P_k using Apriori property [21] (i.e., a candidate pattern of size $k+1$ cannot be prevalent and thus needs not to be generated if any subset pattern of size k is not prevalent). Step 4 builds a grid index on spatial point instances with the cell size equal to the distance threshold. Step 5 counts the number of instances for each feature in every cell. This will be used in our upper bound filter. Step 7 starts the iteration. As long as the set of candidate patterns C_{k+1} is not empty, the algorithm evaluate each candidate pattern $c \in C_{k+1}$. When evaluate a candidate pattern, the algorithm first computes an upper bound of its participation index in parallel using GPU kernels based on the grid index (step 9–10). If the upper bound is below the threshold, the candidate pattern is pruned out. Otherwise, the algorithm runs into a refinement phase, generating the pattern instance table $I_{k+1}.c$ and computing the participation index PI. We design two different parallel refinement algorithms to speed up instance table generation: one using the grid to rule out unnecessary joins, the other using prefix-based hash joins (steps 13 to 16). After all prevalent patterns of cardinality $k+1$ are identified, the algorithm go to the next iteration (steps 19–22). Figure 1(b) illustrates the execution trace for $k = 1$ and $k = 2$.

Algorithm 1. Parallel-Colocation-Miner

```
Input: A set of spatial features $F$
Input: Instances of each spatial features $I[F]$
Input: Neighborhood distance threshold $d$
Input: Minimum prevalence threshold $\theta$
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($P_k$, $k+1$)
 3: Initialize $P_{k+1} \leftarrow \emptyset$
 4: Initialize instance tables $I_k$ $(k = 1)$ by feature instances
 5: Overlay a regular grid with cell size $d \times d$ (total $N$ cells)
 6: Compute $CountMap[N \times |F|]$ in one round instance scanning
 7: while $|C_{k+1}| > 0$ do
 8:   for each $c \in C_{k+1}$ do
 9:     Initialize $PCountMap[N \times |c|] \leftarrow 0$
10:     $Upperbound$ =ParallelCellAggregateFilter($CountMap$,$PCountMap$,$c$)
11:     if $Upperbound \geq \theta$ then
12:       $BitMap \leftarrow 0$ //initialize bitmap for instances of each feature
13:       if Hash Join Refinement then
14:         $(I_{k+1}.c, PI) \leftarrow$ ParallelHashJoinRefine($I_k, c$)
15:       else if Grid Search Refinement then
16:         $(I_{k+1}.c, PI) \leftarrow$ ParallelGridSearchRefine($I_k, CInstances, c, BitMap$)
17:       if $PI \geq \theta$ then
18:         $P_{k+1} = P_{k+1} \cup c$
19:   $P \leftarrow P \cup P_{k+1}$ //add prevalent patterns to results
20:   $k \leftarrow k+1$; $C_k \leftarrow C_{k+1}$; $P_k \leftarrow P_{k+1}$, $I_k \leftarrow I_{k+1}$ //prepare next iteration
21:   $C_{k+1} \leftarrow$ aprioriGen($P_k$, $k+1$)
22:   $P_{k+1} \leftarrow \emptyset$
23: return $P$
```

3.2 Cell-Aggregate-Based Upper Bound Filter

The proposed cell aggregate based upper bound filter first overlays a regular grid with its cell size equal to the distance threshold (shown in Fig. 2), and then computes an upper bound of participation index based on aggregated counts of feature instances in cells. Proposed filter is different from the existing multi-resolution filter [3] in that the computation of upper bound is not based on generating coarse scale colocation instance tables. There are two main advantages of cell aggregate based filter on GPU: first, it is easily 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.

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 Fig. 2, the quadruplet of cell 0 includes cells $(0, 1, 4, 5)$, while the quadruplet of cell 15 is an empty set.

Based on the concept of quadruplet, we can check all potential colocation instances by examining all quadruplets. When examining 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.

Algorithm 2 shows details of proposed filter. The algorithms have three main variables, including $CountMap$, $PCountMap$, and $Quadruplet Aggregate$. $CountMap$ records the true aggregated instance count for each feature in every cell. $PCountMap$ records the instance count for each feature in every cell that potentially participates in the candidate colocation pattern. $QuadrupletAggregate$ is a local array for each cell to record the aggregated count of instances within the quadruplet for each pattern feature. Specifically, steps 1 to 11 computes potential number of participating instances for each feature in each cell in parallel. A kernel thread is allocated to each cell. For a specific cell i, the kernel first gets the quadruplet (step 2). Step 3 initializes a local array $QuadrupletAggregate$ with zero values. Steps 4 to 8 compute the aggregated count of instances for each pattern feature ($QuadrupletAggregate[f]$). If aggregated count of any pattern feature is zero, then there cannot be any candidate pattern instance in the quadruplet and thus the parallel kernel thread terminates (step 8). Otherwise, all feature instances in the quadruplet can potentially participate into colocation pattern instances. Steps 9 to 11 record the potential participating instances from each cell in the quadruplet. This is done by copying instance counts from $CountMap$ to $PCountMap$ for the 4 cells in the quadruplet. It is worth noting that duplicated-counting on the same cell is avoided since different GPU kernel threads may over-write the count for a cell with the same value. Finally, steps 12 to 14 compute the upper bound of participation index based on counts of potential participating instances in $PCountMap$. We use built in GPU library to compute the total counts of distinct participating instances in step 13.

Figure 2 provides an illustrative execution trace of Algorithm 2. Figure 2(a) shows the input spatial instances overlaid with a regular grid. The distance threshold is d. Assume that the candidate colocation pattern is (A, B). Figure 2(b) shows how the filter works. The $CountMap$ array stores the

Algorithm 2. ParallelCellAggregateFilter

```
Input: $CountMap$, feature instance count in cells
Input: $PCountMap$, participating feature instance count in cells
Input: $PR$, participation ratio
Input: Candidate colocation pattern $c$
Output: Upper bound of participation index, $upperBound$
 1: for each cell $i$ do in parallel
 2:   $QuadrupletCells = \textsc{getQuadruplet}(\text{cell } i)$
 3:   $QuadrupletAggregate[|c|] \leftarrow 0$
 4:   for each feature $f \in c$ do
 5:     for each cell $j \in QuadrupletCells$ do
 6:       $QuadrupletAggregate[f] \leftarrow QuadrupletAggregate[f] + CountMap[j][f]$
 7:     if $QuadrupletAggregate[f] == 0$ then
 8:       finish the parallel thread for cell $i$ //no pattern instance in the quadruplet
 9:   for each feature $f \in c$ do
10:     for each cell $j \in QuadrupletCells$ do
11:       $PCountMap[j][f] \leftarrow CountMap[j][f]$ //participating instance count
12: for each feature $f \in c$ do
13:   $PR[f] = \textsc{ParallelSum}(CountMap[\ ][f])/|I_1.f|$
14: $upperBound = \textsc{Min}(PR)$
15: return $upperBound$
```

number of instances for feature A and B in each cell. A GPU thread is assigned to each cell to compute the counts of feature instances within its quadruplet. For example, the leftmost GPU thread is assigned to cell 0. The aggregated instance count for this quadruplet $((0, 1, 4, 5))$ is shown by the leftmost $QuadrupletCount$ array, with 2 instances for A and 1 instance for B. Since instances from both features exist, the number potential participating instances in these four cells ($PCountMap$) are copied from corresponding cell values in $CountMap$, as shown by the fork branches close to the bottom. In contrast, the quadruplet of cell 1 $((1, 2, 5, 6))$ does not contain instances of A, and thus cannot contain colocation pattern instances.

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.

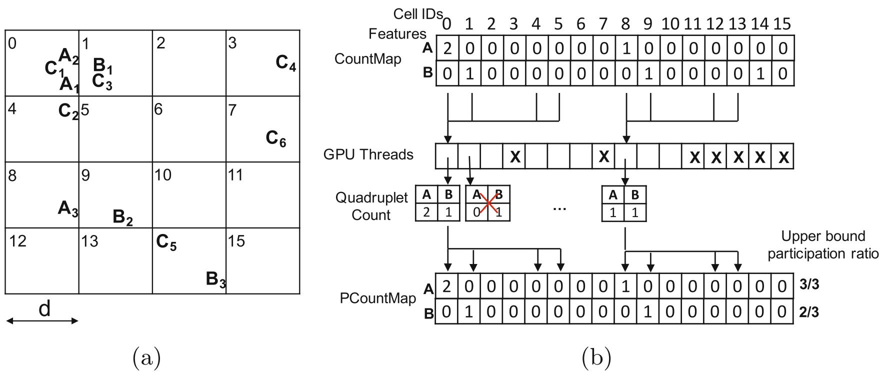


Fig. 2. Grid-aggregate based Upper Bound Filter: (a) A regular grid (b) An execution trace of upper bound filter

The algorithm is correct since it computes the exact participation index of a candidate pattern if it passes the upper bound filter. □

3.3 Refinement Algorithms

The goal of the refinement phase is to generate the instance table of a candidate colocation pattern, and to compute participation index. Generating colocation instance tables is the main computational bottleneck, and thus is done in GPU. As shown in Algorithm 1, we have two options for refinement algorithms, a geometric approach based on grid search called ParallelGridSearchRefine and a combinatorics approach based on prefix-based hash join called ParallelHashJoin-Refine, similar to sequential algorithms discussed in Huang et al. [3]. We now introduce the two algorithms below.

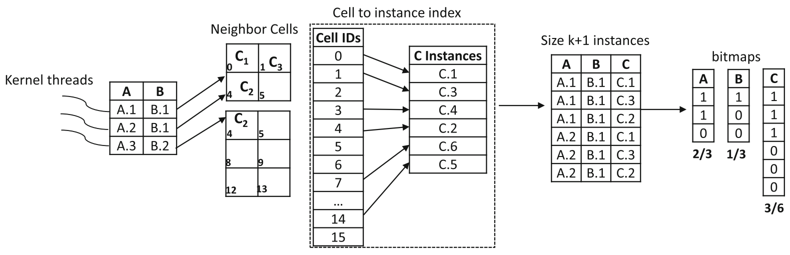
Geometric approach: The geometric approach generate an instance table of a size $k+1$ pattern based on the instance table of a size k pattern. For example, when generating the instance table of pattern (A, B, C), it starts from the instance table of (A, B) and joins each row of the table with instances of the last feature type C. In order to reduce redundant computation, we utilize the grid index and only check the instances of the last feature type within neighboring cells. Algorithm 3 provides details of the proposed grid-based refinement algorithm. Step 2 is kernel assignment. Each kernel thread first finds out all neighboring cells of the size k row instance $rIns$ (step 3). Then, for each neighboring cell, the kernel thread finds out every instance ins of the last feature type in the cell. It joins ins with size k instance $rIns$ to create a size $k+1$ pattern instance $rInsC$ if they are spatial neighbors (steps 4 to 7). The new pattern instance $rInsC$ is inserted into the final instance table $I_{k+1}.c$, and a bitmap is updated to mark the instances that participate into the colocation pattern (steps 8 to 10). Finally, the participation ratio and participation index are computed (steps 11 to 13).

Algorithm 3. *ParallelGridSearchRefine*

```
Input: $I_k$, instance table of patterns of size $k$
Input: $CellInstances$, feature instances for each cell
Input: $BitMap$, bitmap for participating instances from different features
Output: $I_{k+1}.c$, instance table of colocation $c$ (size $k+1$) if prevalent
Output: $PI$, participation index of pattern $c$
 1: //$I_k.(c[1..k])$ is instance table of sub-pattern of $c$ with first $k$ features
 2:
 3: Initialize $I_{k+1}.c \leftarrow \emptyset$
 4: for each row instance $rIns \in I_k.(c[1..k])$ do in parallel
 5:   get neighborhood cells of first feature instance in $rIns$
 6:   for each cell $i$ in neighborhood do
 7:     for each instance $ins$ of feature type $c[k+1]$ in cell $i$ do
 8:       if $ins$ is neighbor of all feature instances in $rIns$ then
 9:         Create new row instance of $c$, $rInsC =< rIns, ins >$
10:         $I_{k+1}.c = I_{k+1}.c \cup rInsC$ //add new instance into $c$'s instance table
11:         for each feature $f \in c$ do
12:           $BitMap[f][rInsC[f]] =$ true
13: for each feature $f \in c$ do
14:   $PR[f] =$ ParallelSum$(BitMap[\,][f])/|I_1.f|$
15: $PI =$ Min$(PR)$
16: return $I_{k+1}.c$, $PI$
```

Figure 3 shows an example. The input data is the same as Fig. 1. Assume that the candidate pattern is (A, B, C). A kernel thread is assigned to each row instance of table (A, B). Thread 1 is assigned to instance (A_1, B_1), and it scans all neighboring cells of A_1 (cells $0, 1, 4, 5$). Based on the cell to instance index, the kernel thread checks all instances of feature C (C_1, C_3, C_2) in these cells, and conducts a spatial join operation. The final output size $k+1$ instances from this thread are (A_1, B_1, C_1), (A_1, B_1, C_3), and (A_1, B_1, C_2).

One issue in GPU implementation is that we need to allocate memory for an output instance table, and specify the specific memory location to which each kernel thread writes its results. For example, in the output instance table of pattern (A, B, C) in Fig. 3, the first kernel thread generates 3 row instances, so



A	B
A.1	B.1
A.2	B.1
A.3	B.2

Cell IDs
0
1
2
3
4
5
6
7
...
14
15

C Instances
C.1
C.3
C.4
C.2
C.6
C.5

A	B	C
A.1	B.1	C.1
A.1	B.1	C.3
A.1	B.1	C.2
A.2	B.1	C.1
A.2	B.1	C.3
A.2	B.1	C.2

A	B	C
1	1	1
1	0	1
0	0	1
		0
		0
		0

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

the second kernel thread has to start with the $4th$ row when writing its instances. It is hard to predetermine the total required memory and enforcing memory coalesce when threads are writing results. Thus, we use a two-run strategy in which in the first run we can calculate the exact size of output instance table as well as slot counts of the number of row instances generated by each kernel thread. In the second run, we allocate memory for output instance table, and use the slot counts to guide which row a kernel thread needs to start from when writing results. Similar to the grid-based refinement, we use two-run strategy to allocate memory and enforce memory coalesce.

Algorithm 4. *ParallelHashJoinRefine*

```
Input: $I_k$, instance table of patterns of size $k$
Input: $BitMap$, bitmap for instances of different features
Output: $I_{k+1}.c$, instance table of colocation $c$ if prevalent
Output: $PI$, participation index of pattern $c$
 1: //$I_k.c1$ and $I_k.c2$ instance tables of $c1 = c[1..k]$ and $c2 = c[1..k-1, k+1]$
 2: for each row instance $rIns1$ in $I_k.c1$ do in parallel
 3:   for each row instance $rIns2$ in $I_k.c2$ starting with $rIns1[1]$ do
 4:     if $rIns1$ and $rIns2$ forms an instance of $c$ then
 5:       Create new instance $rInsC$ by merging $rIns1$ and $rIns2$
 6:       $I_{k+1}.c \leftarrow I_{k+1}.c \cup rInsC$
 7:       for each feature $f \in c$ do
 8:         $BitMap[f].[rInsC[f]] = \text{true}$
 9: for each feature $f \in c$ do
10:   $PR[f] = \textsc{ParallelSum}(BitMap[\,][f])/|I_1.f|$
11: $PI = \textsc{Min}(PR)$
12: return $I_{k+1}.c$, $PI$
```

Prefix-based hash 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. 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 [3], 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 hash join instead. A prefix-based hash index is built on the second table based on instances of the first spatial feature. Details are shown in Algorithm 4. A kernel thread is allocated to each row in the first size k instance table $I_k.c1$ (step 2). The kernel thread then scans all rows in the second size k instance table $I_k.c2$ that has the same first feature instance. For example, if the row in the first table is (A_1, B_1), then the thread only scans rows starting with A_1 in the instance table of (A, C). If the two rows satisfy the join condition (sharing the same first k instances, and having last instances as neighbors), a size $k+1$ instance is created and inserted into output size $k+1$ table (steps 5 to 8). Finally, the participation index is computed (steps 9 to 11). It is worth noting that when generating instance tables of size $k = 2$ patterns, we use the grid-based method since hash-index cannot be created in that case.

An illustrative execution trace is shown in Fig. 4. The raw input data is still the same. Each kernel thread is allocated to a row in instance table (A, B). For example, thread 1 works on pattern instance (A_1, B_1), and scans instance table (A, C). Based on the hash index on A instances, the thread only needs to check (A_1, C_1), (A_1, C_3) and (A_1, C_2). It turns out that B_1 is a neighbor for all C_1, C_2 and C_3. So these instances are inserted to the final output instance table (A, B, C).

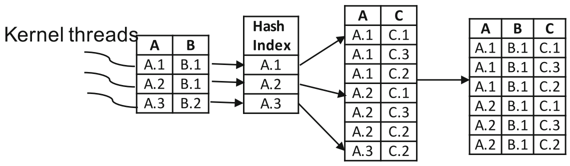


Fig. 4. Illustrative execution trace for hash-join-based refinement

4 Evaluation

The goals of our evaluation are to:

- Evaluate the speedup of GPU colocation algorithms against a CPU algorithm.
- Compare cell-aggregate-based filter with multi-resolution filter on GPU.
- Compare grid-based refinement with hash-join-based refinement on GPU.
- Test the sensitivity of GPU algorithms to different factors.

Experiment Setup: As shown in Fig. 5, we implemented four GPU colocation mining algorithms with two filter options (**M** for multi-resolution filter and **C** for cell-aggregate based filter) and two refinement options (**G** for grid-based and **H** for hash-join based). We also implemented a CPU colocation mining algorithms by Huang et al. [3] (multi-resolution filter, grid-based instance table generation for size $k = 2$, and sort-merge based instance table generation for size $k > 2$). We only compared computational performance since all methods produce the same patterns. For each experiment, we measured the time cost of one run for CPU algorithm, and averaged time cost of 10 runs for GPU algorithms. Algorithms were implemented in C++ and CUDA, and run on a Dell workstation with Intel(R) Xeon(R) CPU E5-2687w v4 @ 3.00 GHz, 64 GB main memory, and a Nvidia Quadro K6000 GPU with 2880 cores and 12 GB memory.

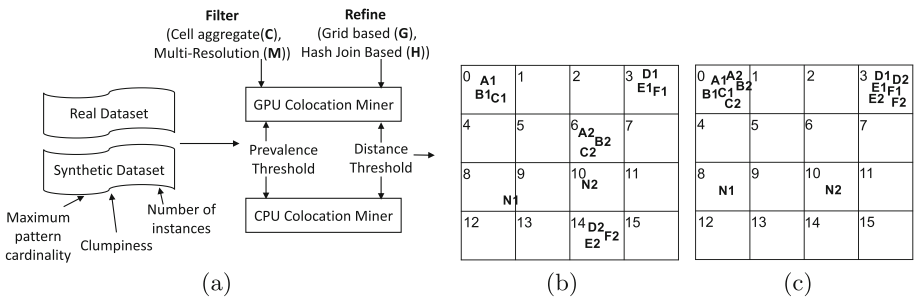


Fig. 5. Experiment setup (a) Experiment design with different candidate approaches; (b) An example of synthetic dataset generated with 2 maximal patterns (A, B, C) and (D, E, F), each pattern with 2 instances with a clumpiness of 1, 2 noise instances N_1 and N_2; (c) Another synthetic dataset similar to (b) but with a clumpiness of 2.

Dataset description: The real dataset contains 13 crime types and 165,000 crime event instances from Seattle in 2012 [22]. The synthetic data is generated similarly to [3]. Figure 5(b–c) provide illustrative examples. 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 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.

Evaluation metric: We used the speedup of proposed GPU algorithms over the CPU algorithm on computational time.

4.1 Results on Synthetic Data

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 Fig. 6(a). GPU algorithms in the plot are based on grid-based filtering. We can see that the speedup of both GPU algorithms increases with the number of feature instances. The grid-based refinement gradually becomes superior over the hash join based refinement in GPU algorithms as the number of instances increases. The reason can be that the cell-instance index in grid-based refinement is done once and for all, while the prefix-based hash index in hash-join based refinement needs to be created repeatedly for each new instance table. The comparison of two approaches with 250,000 instances (the first two points in the curve) may be less conclusive since the running time for both approaches is too small (far below one second).

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 (see Fig. 5(b) versus (c)) but we only have limited memory. The results are summarized in Fig. 6(b). We can see that the grid based refinement is persistently better than hash-join based refinement (around 30 versus 5). The reason is that when the clumpiness is high, there are a large number of pattern instances being formed combinatorially. Many of them share the same prefix (i.e., first feature instance). Thus, each GPU kernel thread in prefix-based hash-join refinement was loaded with heavy computation when doing the join operation, impacting the parallel performance. In contrast, in the grid-based refinement, each GPU kernel thread only scans a limited number of instances within neighboring cells.

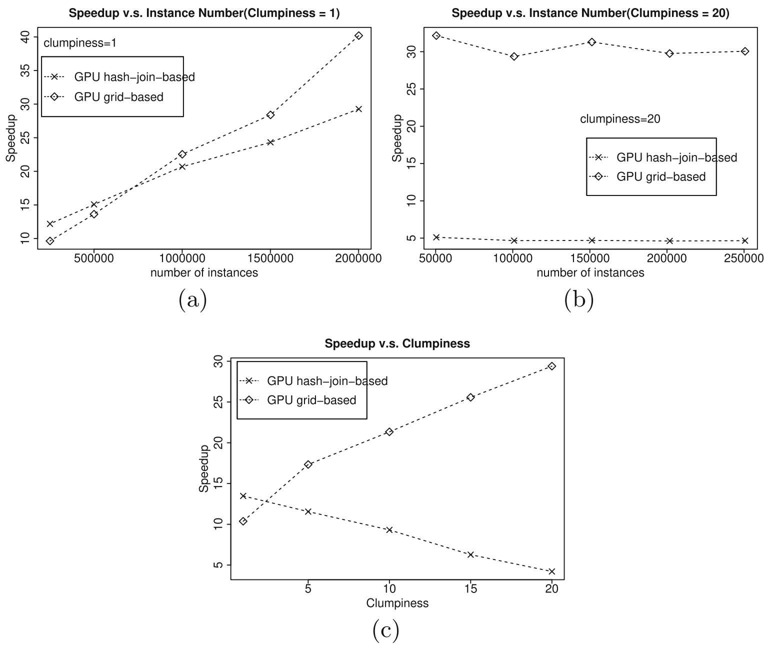


Fig. 6. Results on synthetic datasets: (a) effect of the number of instances with clumpiness as 1 (b) effect of the number of instances with clumpiness as 20 (c) effect of clumpiness with the number of instances as 250 k

Effect of Clumpiness. We set the number of instances to 250 k, and the prevalence threshold to 0.5. Grid-based filtering was used for GPU algorithms. We varied the clumpiness value as 1, 5, 10, 15, and 20. Results in Fig. 6(c) confirm with our analysis above that when clumpiness is higher, the performance of grid-based refinement gets better while the performance of hash-join based refinement gets worse.

Table 1. Comparison of filter and refinement on synthetic data (time in secs)

Clumpiness	Approaches	Filter time	Refine time	Total time	Speedup
1	CPU Baseline	15.3	18.8	34.1	-
	GPU-Filter:M, Refine:H	0.8	1.1	1.9	17.9x
	GPU-Filter:C, Refine:G	**0.2**	**0.7**	**0.9**	**37.9x**
	GPU-Filter:C, Refine:H	**0.2**	**1.0**	**1.2**	**28.4x**
20	CPU Baseline	0.9	407.5	408.4	-
	GPU-Filter:M, Refine:H	**0.1**	97.3	97.4	4.2x
	GPU-Filter:C, Refine:G	**0.1**	**13.8**	**13.9**	**29.4x**
	GPU-Filter:C, Refine:H	**0.1**	96.9	97	4.2x

Comparison on Filter and Refinement. We also compared the computational time of filter and refinement phases of GPU algorithms in the above experiments for the cases with the largest number of instances. Details are summarized in Table 1. When clumpiness is 1, the grid-based filter is much faster than the multi-resolution filter in GPU algorithms (0.2 s versus 0.8 s), making the overall GPU speedup better (37.9 and 28.4 times versus 17.9 times). The reason is that a low clumpiness significantly impacts the multi-resolution filter (coarse scale instance tables cannot be much smaller than true instance tables), while the grid-based filter was less sensitive to clumpiness (more robust) since the time cost of grid-based filtering does not depend on instance distribution. When clumpiness is 20, the refinement phase becomes the bottleneck. The grid-based refinement has a significantly higher speedup than the hash-join refinement (29.4 times versus 4.2 times).

4.2 Results on Real World Dataset

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 (we did not chose thresholds lower than 0.3, because there would be too many instance tables exceeding our memory capacity). The clumpiness of the real dataset was high due to a large density of points. Results are summarized in Fig. 7. As we can see, as the prevalence threshold gets higher, the pruning ratio (candidate patterns being pruned out) gets improved (Fig. 7(a)). The GPU algorithm with grid based refinement is much better than the GPU algorithm

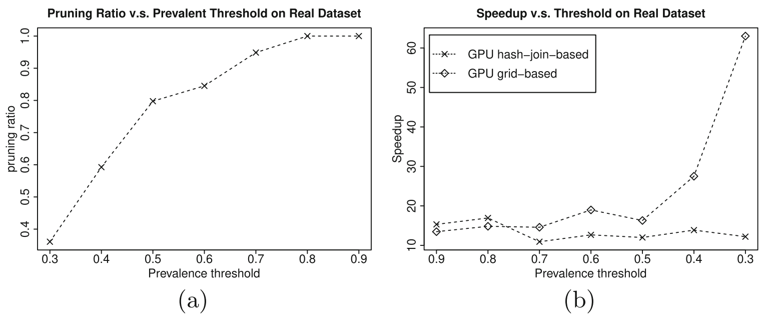


Fig. 7. Results on real world dataset: (a) pruning ratio versus prevalence thresholds (b) speedup versus prevalence thresholds

with hash join based refinement. This is consistent with the results on synthetic datasets when the clumpiness is high.

Comparison of Filter and Refinement. We also compared the detailed computational time in the filter and refinement phases. The distance threshold was 10 meters, and the prevalent threshold was 0.3. Results are summarized in Table 2. Due to a high clumpiness, the refinement phase is the bottleneck, and the grid-based refinement is better than the hash-join based refinement (63.2 times overall speedup versus 12.7 times overall speedup).

Table 2. Comparison of filter and refinement on real dataset (time in secs)

Approaches	Filter time	Refine time	Total time	Speedup
CPU baseline	6.5	340.9	347.4	-
GPU-Filter:M, Refine:H	0.2	26	26.8	13x
GPU-Filter:C, Refine:G	**0.5**	**5.0**	**5.5**	**63.2x**
GPU-Filter:C, Refine:H	**0.5**	26.9	27.4	12.7x

5 Discussion

Preliminary results show that GPU algorithms are promising for the colocation mining problem. One limitation of the proposed GPU algorithm is memory bottleneck. Our algorithm generates instance tables of candidate colocation patterns in GPU. When spatial points are dense and the number of points is large (e.g., millions), such instance tables can reach gigabytes in size. In our implementation, we generate one instance table each time in GPU global memory, and transfer the results to the host in pinned memory. Results showed that the time cost of memory copy was significantly lower (due to pinned memory) than the time cost of GPU computation. In case that the GPU global memory is insufficient for a very large instance table, we can slice it into smaller pieces, compute one piece each time, and transfer results to the host memory. We need to store all relevant instance tables of cardinality k in host memory when computing instance tables of cardinality $k + 1$. Thus, the host memory size can also be a bottleneck. This may be less a concern in future when the main memory price gets lower.

6 Conclusion and Future Work

This paper investigates GPU colocation mining algorithms. We propose a novel 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 design two GPU refinement algorithms, based on grid-based search and prefix based hash-join. We provide theoretical analysis on the correctness and completeness of proposed algorithms. Preliminary results on both real world data and synthetic data on various parameter settings show that proposed GPU algorithms are promising.

In future work, we will explore further refinements on GPU implementation to achieve higher speedup, e.g., avoid redundant distance computation in instance table generation. We will also explore other computational pruning methods.

Acknowledgement. We would like to thank Dr. Simin You for helpful comments and suggestions.

References

1. Koperski, K., Han, J.: Discovery of spatial association rules in geographic information databases. In: Egenhofer, M.J., Herring, J.R. (eds.) SSD 1995. LNCS, vol. 951, pp. 47–66. Springer, Heidelberg (1995). doi:10.1007/3-540-60159-7_4
2. Morimoto, Y.: Mining frequent neighboring class sets in spatial databases. In: Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 353–358. ACM (2001)
3. Huang, Y., Shekhar, S., Xiong, H.: Discovering colocation patterns from spatial data sets: a general approach. IEEE Trans. Knowl. Data Eng. **16**(12), 1472–1485 (2004)
4. Yoo, J.S., Shekhar, S., Smith, J., Kumquat, J.P.: A partial join approach for mining co-location patterns. In: Proceedings of the 12th Annual ACM International Workshop on Geographic Information Systems, pp. 241–249. ACM (2004)
5. Yoo, J.S., Shekhar, S.: A joinless approach for mining spatial colocation patterns. IEEE Trans. Knowl. Data Eng. **18**(10), 1323–1337 (2006)
6. Boinski, P., Zakrzewicz, M.: Collocation pattern mining in a limited memory environment using materialized iCPI-tree. In: International Conference on Data Warehousing and Knowledge Discovery, pp. 279–290. Springer, Heidelberg (2012)
7. Mohan, P., Shekhar, S., Shine, J.A., Rogers, J.P., Jiang, Z., Wayant, N.: 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, pp. 122–132. ACM (2011)

8. Wang, S., Huang, Y., Wang, X.S.: Regional co-locations of arbitrary shapes. In: Nascimento, M.A., Sellis, T., Cheng, R., Sander, J., Zheng, Y., Kriegel, H.-P., Renz, M., Sengstock, C. (eds.) SSTD 2013. LNCS, vol. 8098, pp. 19–37. Springer, Heidelberg (2013). doi:10.1007/978-3-642-40235-7_2
9. Liu, B., Chen, L., Liu, C., Zhang, C., Qiu, W.: RCP mining: towards the summarization of spatial co-location patterns. In: Claramunt, C., Schneider, M., Wong, R.C.-W., Xiong, L., Loh, W.-K., Shahabi, C., Li, K.-J. (eds.) SSTD 2015. LNCS, vol. 9239, pp. 451–469. Springer, Cham (2015). doi:10.1007/978-3-319-22363-6_24
10. Celik, M., Kang, J.M., Shekhar, S.: Zonal co-location pattern discovery with dynamic parameters. In: Seventh IEEE International Conference on Data Mining. ICDM 2007, pp. 433–438. IEEE (2007)
11. Barua, S., Sander, J.: SSCP: mining statistically significant co-location patterns. In: Pfoser, D., Tao, Y., Mouratidis, K., Nascimento, M.A., Mokbel, M., Shekhar, S., Huang, Y. (eds.) SSTD 2011. LNCS, vol. 6849, pp. 2–20. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22922-0_2
12. Barua, S., Sander, J.: Mining statistically significant co-location and segregation patterns. IEEE Trans. Knowl. Data Eng. **26**(5), 1185–1199 (2014)
13. Barua, S., Sander, J.: Statistically significant co-location pattern mining (2015)
14. Yoo, J.S., Bow, M.: Mining top-k closed co-location patterns. In: 2011 IEEE International Conference on Spatial Data Mining and Geographical Knowledge Services (ICSDM), pp. 100–105. IEEE (2011)
15. Huang, Y., Xiong, H., Shekhar, S., Pei, J.: Mining confident co-location rules without a support threshold. In: Proceedings of the 2003 ACM Symposium on Applied Computing, pp. 497–501. ACM (2003)
16. Yoo, J.S., Boulware, D., Kimmey, D.: A parallel spatial co-location mining algorithm based on mapreduce. In: 2014 IEEE International Congress on Big Data (BigData Congress), pp. 25–31. IEEE (2014)
17. Andrzejewski, W., Boinski, P.: GPU-accelerated collocation pattern discovery. In: Catania, B., Guerrini, G., Pokorný, J. (eds.) ADBIS 2013. LNCS, vol. 8133, pp. 302–315. Springer, Heidelberg (2013). doi:10.1007/978-3-642-40683-6_23
18. Andrzejewski, W., Boinski, P.: A parallel algorithm for building iCPI-trees. In: Manolopoulos, Y., Trajcevski, G., Kon-Popovska, M. (eds.) ADBIS 2014. LNCS, vol. 8716, pp. 276–289. Springer, Cham (2014). doi:10.1007/978-3-319-10933-6_21
19. Yoo, J.S., Boulware, D.: Incremental and parallel spatial association mining. In: 2014 IEEE International Conference on Big Data (Big Data), pp. 75–76. IEEE (2014)
20. Andrzejewski, W., Boinski, P.: Parallel GPU-based plane-sweep algorithm for construction of iCPI-trees. J. Database Manage. (JDM) **26**(3), 1–20 (2015)
21. Agrawal, R., Srikant, R., et al.: Fast algorithms for mining association rules. In: Proceedings of 20th International Conference on Very Large Databases, VLDB. vol. 1215, pp. 487–499 (1994)
22. 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