# GPU-Accelerated Collocation Pattern Discovery*

Witold Andrzejewski and Pawel Boinski

Poznan University of Technology, Institute of Computing Science, Piotrowo 2, 60-965 Poznan, Poland

**Abstract.** Collocation Pattern Discovery is a very interesting field of data mining in spatial databases. It consists in searching for types of spatial objects that are frequently located together in a spatial neighborhood. Application domains of such patterns include, but are not limited to, biology, geography, marketing and meteorology. To cope with processing of these huge volumes of data programmable high-performance graphic cards (GPU) can be used. GPUs have been proven recently to be extremely efficient in accelerating many existing algorithms. In this paper we present GPU-CM, a GPU-accelerated version of iCPI-tree based algorithm for the collocation discovery problem. To achieve the best performance we introduce specially designed structures and processing methods for the best utilization of the SIMD execution model. In experimental evaluation we compare our GPU implementation with a parallel implementation of iCPI-tree method for CPU. Collected results show order of magnitude speedups over the CPU version of the algorithm.

## 1 Introduction

The enormous growth of spatial databases limits human abilities to interpret such data and to make useful conclusions. Automatic methods, known as *Knowledge Discovery in Databases* (KDD) are therefore required. KDD has been defined as a non-trivial process of discovering valid, novel, and potentially useful, and ultimately understandable patterns in large data volumes [8]. The most interesting part of this process is called data mining and consists in application of specially designed algorithms to find particular patterns in data.

Popular spatial data mining tasks include spatial clustering, spatial outliers detection, spatial classification and spatial associations. In this work we focus on the problem of detecting classes of spatial objects (the so-called spatial features) that are frequently located together. Each spatial feature can be interpreted as a characteristic of space in a particular location. Typical examples of spatial features include species, business types or points of interest (e.g., hospitals, airports etc.). For example, a mobile company providing multiple services for customers can be interested in relationships between particular factors in the neighborhood of mobile service requests. In ecology and meteorology, the co-occurrence among natural phenomenons can be very interesting for scientists [12].

* This paper was funded by the Polish National Science Center (NCN), grant No. 2011/01/B/ST6/05169.

B. Catania, G. Guerrini, and J. Pokorný (Eds.): ADBIS 2013, LNCS 8133, pp. 302–315, 2013.

Shekhar and Huang defined this data mining task as a *collocation pattern discovery* [11]. A spatial collocation pattern (or in short a *collocation*) is a set of spatial features that are frequently located together in a spatial proximity. Identification of such patterns requires computationally demanding step of searching for all instances of these patterns. Many algorithms for collocation pattern discovery problem have been developed [11,12,13,14,15,16]. However, no solutions utilizing hardware support to accelerate collocation pattern discovery have been proposed yet.

In this paper we propose an algorithm which utilizes the power of modern graphics processing units (GPUs) to accelerate the state of the art algorithm for the collocation pattern discovery.

The structure of this paper is as follows. In section 2 we formally define the terms used throughout the rest of the paper. In section 3, we present the state of the art algorithm for collocation pattern discovery and introduce basic concepts of general processing on GPUs. Section 4 presents our contribution - the GPU-accelerated version of the collocation mining algorithm. The results of experimental evaluation are presented in section 5. We summarize our paper and present plans for future work in section 6.

## 2 Definitions

In this section we introduce the basic collocation pattern mining concepts and definition of the collocation pattern mining problem.

**Definition 1.** *Let $f$ be a spatial feature. An object $x$ is an* ***instance*** *of the feature $f$, if $x$ is a type of $f$ and is described by a location and unique identifier. Let $F$ be a set of spatial features and $S$ be a set of their instances. Given a neighbor relation $R$, we say that the* ***collocation pattern*** *$C$ is a subset of spatial features $C \subseteq F$ whose instances $I \subseteq S$ form a clique w.r.t. the relation $R$.*

**Definition 2.** *The* ***participation ratio*** *$Pr(C, f_i)$ of a feature $f_i$ in the collocation $C = \{f_1, f_2, \ldots, f_k\}$ is a fraction of objects representing the feature $f_i$ in the neighborhood of instances of collocation $C - \{f_i\}$. $Pr(C, f_i)$* is equal to the number of distinct objects of $f_i$in instances of $C$ divided by the number of all instances with feature $f_i$. *The* ***participation index (prevalence measure)*** *$Pi(C)$ of a collocation $C = \{f_1, f_2, \ldots, f_k\}$ is defined as $Pi(C) = \min_{f_i \in C} \{Pr(C, f_i)\}$.*

**Theorem 1.** *The participation ratio and participation index are monotonically non-increasing with increases in the collocation size.*

The collocation pattern mining is defined as follows. Given (1) a set of spatial features $F = \{f_1, f_2, \ldots, f_k\}$ and (2) a set of their instances $S = S_1 \cup S_2 \cup \ldots \cup S_k$ where $S\,(1 \leq i \leq k)$ is a set of instances of feature $f_i \in F$ and each instance that belongs to $S$ contains information about its feature type, instance id and location, (3) a neighbor relationship $R$ over locations, (4) a minimum prevalence threshold (*min_prev*), find efficiently a correct and complete set of collocation patterns with a participation index $\geq$ *min_prev.*

# 3 Related Work

## 3.1 The iCPI-Tree Based Collocation Pattern Discovery

The general approach to collocation mining problem has been proposed in [11] and consists in three major steps. In the first step, a well-known Apriori strategy [1] is used to generate candidate collocations utilizing anti-monotonicity property of the prevalence measure. In the second step instances of such candidates are identified. Finally, in the last step, the prevalence measure is computed for each candidate. Candidates with the prevalence below the given threshold are filtered out.

Although there are researches that do not follow the aforementioned Apriori strategy (e.g., maximal collocation patterns [16], density based collocation patterns [14]), the general approach is the most popular one. Among the most notable general approach methods are *Co-Location Miner* [11], *Joinless* [15] and current state of the art *iCPI-tree* based method [13]. In the next paragraphs we briefly describe the idea behind this algorithm.

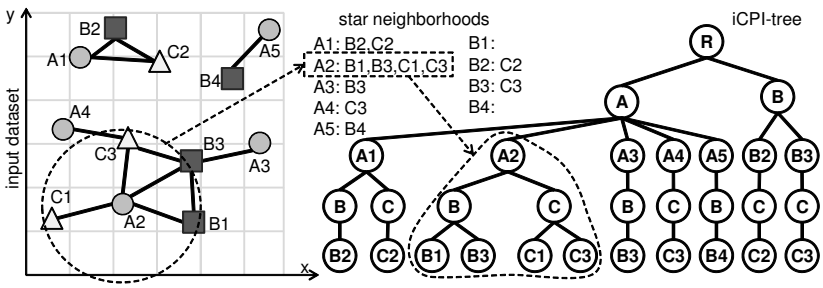


**Fig. 1.** Sample dataset and the corresponding iCPI-tree

At the beginning each spatial feature is denoted as a one element collocation. Next, two element candidates are generated following the Apriori strategy. To compute their prevalences, a list of their instances is required. In the iCPI-tree method, the concept of star neighborhoods (originally introduced in [15]) is used. For each object in space, a list of all neighbors with spatial features greater than the feature of this particular object is called a star neighborhood. Such information is stored in the form of an *iCPI-tree*. Each child of the root node is a subtree that contains neighbors for instances of a specific spatial feature. Sub-trees are composed of nodes representing spatial features of neighbors and leafs corresponding to neighbor instances. A sample dataset and a corresponding iCPI-tree is shown in Fig. 1. For example, a star neighborhood for object $A1$ consists of objects $B1, B3, C1, C3$ and is represented by a subtree below a node $A$ in the iCPI-tree. To identify instances of a candidate, e.g, $B, C$ we can easily read all neighbors with $C$ feature of $B$ instances ($B2, C2$ and $B3, C3$). In $n$-th iteration of the algorithm, $n$ size candidates are processed. For $n = 2$ all

instances generated from the tree are cliques (Def. 1). For $n > 2$ the following method for generating collocation instances is used. To identify instances of $n$ size collocation candidates, a set of instances from $n-1$ iteration is used. For each $n$ size candidate, instances of the prevalent $n-1$ size collocation with the same first $n-1$ features as candidate are expanded. Only common neighbors of all collocation instance objects can be used. For example, given the candidate $A, B, C$ we use consecutive instances of the collocation $A, B$, e.g., the instance $A2, B3$. We try to extend it with instances of feature $C$. To get clique instances of $A, B, C$ we look for $C$ instances that are simultaneously neighbors of $A2$ as well as of $B3$. Using the obtained iCPI-tree we can easily find that $A2$ has neighbors $C1$ and $C3$ while $B3$ has only the neighbor $C3$. Therefore, the only common neighbor is $C3$ and a new instance of candidate is $A2$,$B3$,$C3$. Such processing is repeated for each candidate. The algorithm stops when there are no new candidates. For details of the iCPI-tree based algorithm please consult the paper [13].

### 3.2 General Processing on Graphics Processing Units

The rapid development of computer graphics cards has led to creating powerful devices performing highly parallel single instruction multiple data computations. This high performance may be utilized not only for graphics applications but also for any general processing applications. Newly developed APIs such as NVIDIA CUDA [10] and OpenCL [9] allow to relatively easily develop programs which utilize *graphics processing units* (GPUs) of graphics cards to accelerate their normal data processing tasks. In our solutions we utilize NVIDIA CUDA, though many of the results presented in this paper are also applicable to OpenCL based applications. NVIDIA CUDA is a low level API which while designed to be universal, is currently only implemented for NVIDIA GPUs. This will probably change in short time as there are new compilation frameworks currently in development, which allow to run CUDA programs on AMD and Intel GPUs as well [7]. Below we give a short description of NVIDIA CUDA API and its capabilities.

Computation tasks are submitted to GPUs in a form of kernels. A kernel is a function which is composed of a sequence of operations which need to be performed concurrently in multiple threads. Threads are divided by the programmer into equally sized *blocks*. A block is a one, two or three dimensional array of at most 1024 threads (or 512 depending on the graphics cards' architecture), where each thread can be uniquely identified by its position in this array. The set of blocks forms a so-called *computation grid*. Threads in a single block may communicate by using the same portion of the so-called shared memory which is physically located on-chip and therefore very fast. Threads running in different blocks may only communicate through the very slow *global memory* of the graphics card. Different memory types have different efficient access pattern requirements. Synchronization capabilities of the threads are limited. Threads in a single block may be synchronized; however, global synchronization of threads is achievable only by means of a costly workaround. Threads in a block are executed in 32 thread SIMD groups called warps (all of these threads should

perform the same instruction at the same time). A programmer implementing an algorithm using the NVIDIA CUDA API must take into account all of these low level GPU limitations to obtain the most efficient implementations.

To facilitate creation of programs performing parallel computations (not necessarily GPUs) many parallel primitives have been developed. For the purpose of solutions presented in this paper we utilize the following: *inclusive* and *exclusive scan*, *compact*, *sort*, *unique*, *reduce* and *reduce by key*. Most of these primitives are implemented for GPUs in such libraries as *Thrust* [3]. In our implementation we use this library though we implement our own version of the compact algorithm. Below we give short description of each primitive.

Given any array $a$ an *inclusive scan* finds an array $b$ of the same size such that each $b[i] = \sum_{k=1}^{i} a[i]$. An *exclusive scan* works similarly. For any array $a$ an *exclusive scan* finds and array $b$ of the same size such that each $b[i] = \sum_{k=1}^{i-1} a[i]$. Any associative binary operator may be used instead of sum. A *compact* algorithm given any array $a$ removes all entries that fulfill some condition. Most commonly an additional array of flags storing 0 and 1 (remove/keep) is supplied. A *sort* algorithm sorts any array. Sorting may also be based on some additional key array. A *unique* algorithm is used to find a set of all distinct values stored in a user supplied array. Thrust implementation requires data to be sorted first. A *reduce* algorithm performs reduction of all values within a user specified array $a$ by using any associative binary operator such as sum, i.e., it can be used to find a sum of all values within an array. A *reduce by key* algorithm is a more advanced version of *reduce* algorithm. Given two equally sized arrays $k$ and $v$ where array $k$ stores keys and array $v$ stores values, reduce by key performs reduction of all values belonging to the same key, i.e., for each distinct key value a single reduced result is obtained. Thrust implementation requires data to be sorted by key value.

## 4 The GPU-CM Algorithm

Our GPU-CM algorithm assumes that the iCPI-tree has already been built. Efficient algorithms for constructing this structure from raw input data already exist [13,15]. Moreover this algorithm step takes only a small fraction of the whole mining process time [4].

### 4.1 Data Structures

Following the solutions presented in [4] we represent an iCPI-tree as a hash map. Each entry in the hash map stores a list of instances of a single spatial feature $f_1$ which are neighbors of a single instance $i$ of some other spatial feature $f_2$. This structure is implemented in GPU memory as follows.

Lists of instance neighbors are stored in a single memory block. Due to performance reasons, each list has a constant length $L$ which can be any power of 2 (up to 32). If the number of neighbors is smaller than this value some entries are left empty. The memory block size is therefore equal to the number of lists

times $L$. Each entry on a list is an instance representation. Each feature instance is represented as a 32 bit word where the least significant 16 bits store instance number and more significant 8 bits are used for storing the feature number. The most significant 8 bits are set to zero. An empty entry is represented by the value $FFFFFFFFh$. Each list occupies continuous $L \times 4$ bytes of memory. Each list is sorted. We will denote this data structure as an *instance neighbor buffer*.

A hash map is represented by two arrays: *keys* and *values*. Each key stores feature $f_1$ number as well as feature $f_2$ number and instance number $i$. This is encoded on a 32 bit word where the least significant 16 bits encode instance number $i$, more significant 8 bits encode feature $f_2$ identifier and the most significant 8 bits encode the feature $f_1$ number. Each entry in the *keys* array stores either a key value or is empty (equal to $FFFFFFFFh$). We use the open addressing scheme for hashing [2]. The array *values* stores, for each key stored in *keys* array, a pointer to the start of the appropriate list in the memory block described above. We will denote this hash map as an *instance neighbor hash map*.

During the collocation pattern mining process, the algorithm stores for each pattern (or a candidate for one) a list of its instances. To accelerate the search for instances of a specified pattern, another hash map is employed. This structure is more complicated than iCPI-tree representation and is implemented as follows.

Depending on the number of features, each pattern is represented as a sequence of several 32 bit words, where each bit corresponds to a single feature. We denote the number of pattern words as $BL$. All patterns are stored in a single memory block of size equal to the number of patterns times $BL$. Due to the access methods employed during collocation mining each pattern does not occupy continuous regions of memory. Instead, consecutive words in memory store first words of all patterns, then second words and so on. Patterns are sorted in the lexicographic order. We will denote this structure as *a pattern buffer*. Pattern instances are stored similarly in a separate memory block. Each pattern instance is represented as a sequence of 32 bit words where each word is a feature instance encoded as described at the beginning of this section. Consequently, this memory block's size is equal to the number of all pattern instances times the pattern length. Similarly as with patterns, continuous regions of memory store first words of all pattern instances, then second words and so on. Pattern instances of a single pattern occupy neighboring regions of memory, i.e., pattern instances are not interleaved. Moreover, pattern instances are sorted in the lexicographic order (within the groups corresponding to their patterns). We will denote this structure as a *pattern instance buffer*.

A hash map used for finding instances of a pattern is represented by two arrays: *keys* and *values*. Array *keys* stores pointers to starting positions of corresponding patterns in the pattern memory block described above or nulls if entry is empty. Array *values* stores, for each corresponding key, records that consist of: the number of pattern instances, the prevalence of the pattern and the pointer to the first word of a first pattern instance in the pattern instance memory block. We use open addressing scheme for hashing. We will denote this hash map as a *pattern hash map*.

One might notice that the *keys* array of the pattern hash map could store patterns instead of pointers to patterns. The current solution was used to achieve atomic insertions into the hash map. In general we follow a solution presented in [2] where a parallel hash map based on open addressing scheme is introduced (among others). This solution is based on compare and swap scheme. Unfortunately the *atomicCAS* CUDA function (perfoming atomic compare and swap) works only on 32 bit and 64 bit words. Atomic insertion of longer patterns is not possible by means of this function. We have decided to atomically store pointers (which are either 32 or 64 bit) to larger representations of patterns instead.

### 4.2 Initialization of Mining

The mining process starts with reading of the initial iCPI-tree and construction of iCPI-tree hashmap. Instance neighbor buffer is constructed sequentially. Additional data such as numbers of instances of each feature are also retrieved. Moreover, a temporary array of all instance neighbor hash map keys and values (pointers to lists) is sequentially constructed as well. Finally, all of these key - value pairs are inserted into the instance neighbor hash map in parallel.

After the instance neighbor buffer and the instance neighbor hash map are constructed, a pattern buffer, a pattern instance buffer and a pattern hash map for patterns of length 1 are constructed as well. While we have designed a parallel algorithm for this step, we will omit the details as the processing time of this step is negligible.

### 4.3 Generation of Candidates - Pattern Join

Generation of $n$-size candidates utilizes a pattern buffer which stores $n-1$ size patterns. As patterns in the pattern buffer are sorted in the lexicographic order, all $n-1$ size patterns that may be joined into an $n$ size pattern form groups of several consecutive, joinable patterns. We will refer to these groups as *join groups*. At the final stage of the algorithm, the join groups will be converted into *result groups*. A result group is a group of several consecutive patterns that are obtained from joining all patterns within a single join group. The order of groups is retained, i.e., given any two join groups $A$ and $B$, if group $A$ is before group $B$ in the input pattern buffer, the corresponding result groups will be in the same order in the output pattern buffer (unless one of the join groups contains only a single pattern and therefore does not create any join results). If the patterns within a result group are sorted in lexicographic order then this property will guarantee that the resulting pattern buffer is sorted globally.

The pattern join algorithm works in several sequential stages, though each stage can be composed of parallel operations. First stages identify join groups, find their number (denoted $k_{JG}$), compute their size and the size of corresponding result groups (the number of combinations of two). Several important arrays are created:

- *groupSizes* - an array of size $k_{JG}$ which contains for each join group its size,

- *joinCounts* - an array of size $k_{JG}$ which contains for each join group the size of corresponding result group,
- *positions* - an array of size $k_{JG}$ which contains for each join group an index of the last pattern for this group within the input pattern buffer,
- *scannedJoinCounts* - an array of size $k_{JG}$ which is a result of exclusive scan operation performed on *joinCounts* array.

Next stages compute an additional important auxiliary array called *scannedJoinFlags* of size equal to the number of result patterns ($k_P$). The obtained array contains (for each pattern in each result group) a reference number of the corresponding join group.

The final stage creates the final pattern join. Each *join group* is converted into a *result group*. First, a pattern buffer of size $k_p \times BL$ is allocated (recall that $BL$ is the number of 32 bit words needed to encode a single pattern). Next, $k_p$ threads are started. Each thread, based on the array *scannedJoinFlags* determines the reference number of the corresponding join group. Given this value, the thread retrieves from the *scannedJoinCounts* array the position at which its corresponding result group should start in the result pattern buffer. The thread also computes the difference between its global number and the retrieved position to find its position *pos* within the corresponding result group. This value is then decomposed into numbers of two sequences within the corresponding join block via the following formulas: $p_1 = bs - 1 - \left\lceil 0.5(\sqrt{8(jc - 1 - pos) + 9} - 1) \right\rceil$ and $p_2 = pos - 0.5p_1(2bs - p_1 - 3) + 1$, where: $bs$ is the corresponding join group size retrieved from the array *groupSizes* and $jc$ is the corresponding result group size retrieved from the array *joinCounts*.

These formulas accomplish two tasks: (1) a threads position within the corresponding result group is decomposed into a combination of two patterns within the corresponding join group and (2) the joined patterns will be sorted lexicographically within the result group. The positions $p_1$ and $p_2$ are converted into global positions within the input pattern buffer by adding the appropriate value from the *positions* array (and increasing by 1). Finally, the thread joins the two patterns by performing a binary bitwise OR operation between all words of the two patterns and stores the result in the resulting pattern buffer.

### 4.4 Generation of Candidates - Candidate Pruning

Each pattern obtained through joining must be checked whether its every subpattern is prevalent or not. To perform this check for patterns of length $n$ we utilize: a pattern buffer obtained in the previous algorithm step (see section 4.3) and a pattern hash map for prevalent patterns of length $n-1$. Assume there are $k_p$ patterns obtained through joining. First, an array *flags* of size $k_p$ is allocated. Next, $k_p$ threads are started. Each thread retrieves its corresponding pattern from input pattern buffer and sequentially generates all $n$ subpatterns of length $n-1$. Each subpattern is checked whether the corresponding entry exists in the pattern hash map. If all subpatterns of a single pattern have corresponding

entries in the pattern hash map, the thread stores 1 in the corresponding position in the *flags* array, otherwise 0 is stored.

As a second step a parallel compact algorithm is employed to remove all patterns except for patterns with the corresponding flag equal to 1. As parallel compact does not change the order of patterns, the lexicographic order is retained. The obtained pattern buffer will be referred to as a *candidate pattern buffer*.

### 4.5 Generation of Instances

In this section we describe an algorithm which performs the most time consuming step of the collocation pattern discovery - the generation of instances. Let us introduce several useful terms. Let the number of candidate patterns be equal to $k_C$. Any $n-1$ size pattern that is a prefix of a $n$ size candidate pattern will be denoted as a *candidate prefix pattern*. Instances of a candidate prefix pattern will be called *candidate prefix pattern instances*. The last feature of a candidate pattern will be called an *extending feature*.

The basic idea for instance generation algorithm is based on the following observations. In the basic iCPI-tree based algorithm finding instances of some candidate pattern $C$ with extending feature $f_e$ involves: (1) retrieving every candidate prefix instance $P_i$, (2) finding for each feature instance of $P_i$ a list of its neighbors with feature $f_e$ by means of iCPI-tree and (3) finding a common part of these lists. As processing of every candidate prefix instance is independent it is a natural candidate for parallelization. Our algorithm processes each candidate prefix instance of every candidate pattern in parallel. Let $k_I$ be the number of processed prefix instances. Notice that each candidate prefix instance might be processed more than once if there are several candidates with the same candidate prefix pattern but different extending feature. Each candidate prefix instance can have two (local and global) numbers. A *local candidate prefix instance number* is a number of the candidate prefix instance within the group of candidate prefix instances of a single candidate prefix pattern. A *global candidate prefix instance number* is a number of candidate prefix instance within the set of all candidate prefix instances used in generation of candidate pattern instances. As an input the algorithm utilizes: a candidate pattern buffer containing $n$ size patterns obtained in the previous stage, a pattern buffer, a pattern instance buffer, a pattern hash map of prevalent $n-1$ size patterns as well as an instance neighbor buffer and an instance neighbor hash map. The main instance generation algorithm requires also several auxiliary arrays which can be computed in parallel as well:

- *listPointers* - an array of size $k_C$ which for every candidate pattern stores a pointer to the first candidate prefix instance in the pattern instance buffer,
- *instanceCounts* and *scannedInstanceCount* - instanceCounts is an array of size $k_C$ which for every candidate pattern stores the number of corresponding candidate prefix instances, scannedInstanceCount is a result of perfoming a parallel inclusive scan on the instanceCounts array,
- *correspondingPatterns* and *extendingFeatures* - arrays of size $k_I$ which map global candidate prefix instance number to the number of the corresponding

candidate pattern and its extending feature respectively, entries in both of these arrays are sorted by the corresponding pattern.

In the first step, for each candidate prefix instance a list of common $f_e$ feature instances is generated. Two arrays are created: (1) *listSizes* of size $k_I$ which will store lengths of each neighbor list and (2) *newNeighbors* of size $k_I \times L$ (recall that $L$ is the length of neighbor list in instance neighbor buffer structure) which will store the neighbor lists. The memory alignment and the structure of the *newNeighbors* array is the same as that of the instance neighbor buffer. To perform this step $k_I \times L$ threads are started. Each group of consecutive $L$ threads cooperates to generate one neighbor list in the *newNeighbors* array. Each thread at start determines the following information: (1) the global number candidate prefix instance $c \in 0, \ldots, k_I - 1$ (the same for each of $L$ consecutive threads), (2) the corresponding position within the neighbor list $l \in 0, \ldots, L - 1$, (3) the corresponding candidate pattern number $p$ from the *correspondingPatterns* array and (4) the extending feature $f_e$ from the *extendingFeatures* array. Next, based on the global candidate prefix instance number $c$ each thread at start determines the local candidate prefix instance number. This is done by substracting *scannedInstanceCounts[p-1]* from $c$. If $p = 0$ the candidate pattern instance number is equal to $c$. Each thread also retrieves from the *listPointers* array a pointer to the first candidate prefix instance of their corresponding candidate pattern $p$. Based on this pointer and the local candidate prefix instance number each thread computes the address of the first feature instance of the corresponding candidate prefix instance that it is going to process. Now each of the L threads work synchronously to find an intersection of the lists of neighbors with $f_e$ feature of every feature instance being a part of the processed candidate prefix instance. The whole process is performed almost solely in the fast shared memory of GPU. When the threads finish their work they copy their results into the *newNeighbors* array and store the lengths of the obtained lists into the *listSizes* array.

After the neighbor lists are found, the algorithm performs a modified parallel compact algorithm which removes empty entries from the obtained lists, but every remaining entry is materialized in the resulting array as a complete candidate pattern instance, i.e., the corresponding candidate prefix instance with the appended entry. The resulting array is built in such a way that its structure and properties are the same as that of the pattern instance buffer.

### 4.6 Computation of Prevalence

The computation of prevalence is based on several classic parallel algorithms such as sort, reduce by key and unique (see section 2 for details). The process starts with computing, for every feature instance of each candidate pattern instance, a value composed of: its position within the candidate instance, a candidate pattern number, and an instance identifier constructed as described in section 4.1. An array of such values is then sorted in the lexicographic order. Next, non unique values are removed via the unique algorithm. The obtained array

now stores for every candidate pattern a list of unique feature instances at each position of its instances. Next, in parallel a feature instance number is removed from every value in the obtained array (although feature identifier remains). Finally, the reduce by key algorithm is used to count the number of distinct values in the array obtained in the previous step (the array is treated as key array and value array is composed of ones). The obtained results store for each position of every candidate pattern a number of unique feature instances appearing in candidate pattern instances. Based on these values, the participation ratios and prevalences of all candidate patterns are computed (in parallel).

Non prevalent candidate patterns and their corresponding instances are removed via the parallel compact algorithm. Only the prevalent patterns and their instances remain (in the pattern buffer and the pattern instance buffer respectively). Finally, a new pattern hash map is constructed (in parallel). These structures now form an input of the next iteration of the collocation pattern discovery algorithm.

## 5 Experiments

### 5.1 Implementation and Testing Environment

For the purpose of this paper we have prepared two implementations of the iCPI-tree based collocation pattern discovery algorithm: for GPU and for CPU. The GPU version uses the solutions described in section 4. The CPU version uses similar data structures as the ones described in section 4.1, however the parallelization of computations is done differently. Instead of SIMD approach, multiple instruction multiple data approach is used (MIMD). CPU implementation uses OpenMP [6] to parallelize instance generation and prevalence computation on a multi-core CPU. The implementation starts the number of threads equal to the number of cores of a CPU and the computation tasks are distributed among the started threads. The parameter $L$ (length of lists in instance neighbor buffer) for both implementations was set to 8.

Experiments were run on a computer with Core2 Duo 2,1Ghz CPU (CPU implementation started two threads) and 8GB of RAM and GeForce 580GTX graphics card with 1.5GB of RAM (Fermi architecture) working under Microsoft Windows 7 operating system.

### 5.2 Data Sets and Experiments

To evaluate our GPU version of the algorithm we have prepared 10 synthetic datasets. We have used a synthetic generator similar to the one described in [15]. The number of data objects ranged from 25 K to 120 K, the number of spatial features ranged from 30 to 90 and 20 to 80 percent of total instances were noisy instances. Two kinds of datasets have been prepared: dense and sparse. Dense datasets have been generated by reducing the size of spatial framework ten times in each dimension while preserving the number of objects (sparse datasets were generated over a grid of size of 10000x10000 units).

We have conducted three series of experiments. Each time we measured *speedup*, i.e., the ratio of execution time of CPU version of the algorithm to the execution time of GPU implementation. In the first series we have examined how increasing of the minimum prevalence threshold affects speedup in both dense and sparse datasets. In the second series we investigated the influence of the distance threshold between neighbors on speedup for two opposite levels of minimum prevalence. Finally we examined how speedup changes with increasing size of the input dataset.

### 5.3 Results and Interpretation

Figure 2(a) presents results of the first experiment which tested the influence of the minimum prevalence parameter on the algorithm performance. Two interesting observations can be made. First, notice that the density of the dataset does not influence the speedup, i.e., while processing times may change, GPU version is faster than CPU version by approximately the same factor for the same minimum prevalence. Second, the speedup drops monotonically with the increase of minimum prevalence. In our testing environment for minimum prevalence equal to 0.2 the achieved speedup is roughly twice the speedup achieved for the minimum prevalence 0.6. This is due to the fact that the increased minimum prevalence parameter causes less candidate patterns to be generated and (indirectly) less candidate prefix instances to be processed. This in turn causes less threads to be started. Less threads mean that: (1) memory transfers may not be hidden, (2) GPU multiprocessors might not have recieved a full load and (3) instance generation step takes less time in comparison to other algorithm steps (as we have primarily focused on optimizing instance generation step this could cause the observed speedup to drop).

Figure 2(b) presents results of the second experiment which tested the influence of the distance threshold between neighbors on the algorithm performance. One can notice that in general, the larger the distance threshold, the greater the speedup. The explanation of this observation is very similar to the one made in the previous experiment. For large distance thresholds, each feature instance has more neighbors. Consequently neighbor lists in the instance neighbor buffer are longer and more instances are generated in the instance generation step of the algorithm. As we have shown in the previous paragraph, the larger the number of instances, the higher speedups can be achieved. One can also notice that we have performed this experiment for two values of minimum prevalence parameter: 0.2 and 0.6. Plots corresponding to these two values confirm the observations made in the previous experiment: the lower the minimum prevalence the higher the observed speedup. Moreover, similarly as before, one can notice that for minimum prevalence 0.2 observed speedup is twice the speedup observed for minimum prevalence 0.6.

Figure 2(c) presents results of the third experiment which tested the influence of the number of feature instances (objects) on the observed speedup. What is surprising is that the observed speedup grows linearly (for the tested datasets) with respect to the number of objects. It is of course obvious that the speedup


**Fig. 2.** Experiment results

cannot grow forever. The same curve for larger instances would asymptotically approach the maximum achievable speedup dependant on relative performances of GPU and CPU. The monotonic dependency can be explained similarly as before. Greater number of objects leads to greater number of pattern instances and therefore candidate prefix instances. The larger the number of such instances, the larger the number of threads started and the better the utilization of GPU.

Unfortunately, due to limited GPU memory we were not able to run instances large enough to achieve maximal speedups. Algorithm's memory requirements depend on data and query characteristics such as the number of objects, the number of spatial features, data density and the minimal prevalence. In our case, 1.5GB of GPU memory was enough to run dense datasets of size up to 120K objects, 90 spatial features, for the minimal prevalence of 0.2.

## 6 Summary and Future Work

In this paper we have presented an algorithm performing most operations of the state of the art collocation pattern discovery algorithm in parallel on GPU. We have compared our implementation to the multi-threaded CPU implementation and shown that GPU offers an order of magnitude speedup over the CPU version.

While the results are very promising, there is still a lot of work to do. The main problem is the limited graphics cards memory. This problem can be tackled in two ways. First, we plan on modifying our algorithm to be able to process variable length neighbor lists. Current solution wastes both memory and computing power (a lot of threads process empty positions on these lists). Second, we plan on adapting the solutions introduced in [5] which already try to solve the collocation pattern mining problem in limited memory conditions. We also plan designing an algorithm for efficient construction of iCPI-trees on GPU (currently done on CPU). Finally we also want to approach a more difficult problem of the maximal collocation pattern discovery on GPUs.

## References


1. Agrawal, R., Srikant, R.: Fast Algorithms for Mining Association Rules in Large Databases. In: Proceedings of the 20th International Conference on Very Large Data Bases, pp. 487–499. Morgan Kaufmann Publishers Inc., San Francisco (1994)
2. Alcantara, D.A.F.: Efficient Hash Tables on the GPU. Ph.D. thesis, University of California, Davis (2011)
3. Bell, N., Hoberock, J.: Thrust: A Productivity-Oriented Library for CUDA. In: GPU Comuting Gems: Jade edition, pp. 359–371. Morgan-Kauffman (2011)
4. Boinski, P., Zakrzewicz, M.: Collocation Pattern Mining in a Limited Memory Environment Using Materialized iCPI-Tree. In: Cuzzocrea, A., Dayal, U. (eds.) DaWaK 2012. LNCS, vol. 7448, pp. 279–290. Springer, Heidelberg (2012)
5. Boinski, P., Zakrzewicz, M.: Partitioning Approach to Collocation Pattern Mining in Limited Memory Environment Using Materialized iCPI-Trees. In: Morzy, T., Härder, T., Wrembel, R. (eds.) Advances in Databases and Information Systems. AISC, vol. 186, pp. 19–30. Springer, Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-32741-4_3
6. Chapman, B., Jost, G., van der Pas, R.: Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). The MIT Press (2007)
7. Farooqui, N., Kerr, A., Diamos, G., Yalamanchili, S., Schwan, K.: A framework for dynamically instrumenting gpu compute applications within gpu ocelot. In: Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-4, pp. 9:1–9:9. ACM, New York (2011)
8. Fayyad, U., Piatetsky-Shapiro, G., Smyth, P.: From Data Mining to Knowledge Discovery in Databases. AI Magazine 17, 37–54 (1996)
9. Khronos Group: The OpenCL Specification Version: 1.2 (2012), http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf/
10. NVIDIA Corporation: Nvidia cuda programming guide (2012), http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf
11. Shekhar, S., Huang, Y.: Discovering Spatial Co-location Patterns: A Summary of Results. In: Jensen, C.S., Schneider, M., Seeger, B., Tsotras, V.J. (eds.) SSTD 2001. LNCS, vol. 2121, pp. 236–256. Springer, Heidelberg (2001)
12. Shekhar, S., Huang, Y.: The multi-resolution co-location miner: A new algorithm to find co-location patterns in spatial dataset. Tech. Rep. 02-019, University of Minnesota (2002)
13. Wang, L., Bao, Y., Lu, J.: Efficient Discovery of Spatial Co-Location Patterns Using the iCPI-tree. The Open Information Systems Journal 3(2), 69–80 (2009)
14. Xiao, X., Xie, X., Luo, Q., Ma, W.Y.: Density Based Co-location Pattern Discovery. In: Proceedings of the 16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS 2008, pp. 29:1–29:10. ACM, New York (2008), http://doi.acm.org/10.1145/1463434.1463471
15. Yoo, J.S., Shekhar, S., Celik, M.: A Join-Less Approach for Co-Location Pattern Mining: A Summary of Results. In: Proceedings of the IEEE International Conference on Data Mining, Washington, pp. 813–816 (2005)
16. Yoo, J.S., Bow, M.: Mining Maximal Co-located Event Sets. In: Huang, J.Z., Cao, L., Srivastava, J. (eds.) PAKDD 2011, Part I. LNCS, vol. 6634, pp. 351–362. Springer, Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-20841-6_29