Intersecting two families of sets on the GPU

Marta Fort[a,*], J. Antoni Sellarès[a], Nacho Valladares[a]

[a] *Universitat de Girona, Informàtica i Matemàtica Aplicada, Girona, Spain*

Abstract

The computation of the intersection family of two large families of unsorted sets is an interesting problem from the mathematical point of view which also appears as a subproblem in decision making applications related to market research or temporal evolution analysis problems. The problem of intersecting two families of sets $\mathcal{F}$ and $\mathcal{F}'$ is to find the family $\mathcal{I}$ of all the sets which are the intersection of some set of $\mathcal{F}$ and some other set of $\mathcal{F}'$. In this paper, we present an efficient parallel GPU-based approach, designed under CUDA architecture, to solve the problem. We also provide an efficient parallel GPU strategy to summarize the output by removing the empty and duplicated sets of the obtained intersection family, maintaining, if necessary, the sets frequency. The complexity analysis of the presented algorithms together with experimental results obtained with their implementation are also presented.

Keywords: Intersection of families of sets, Algorithm, Graphics processing unit (GPU), CUDA

1. Introduction

The problem of intersecting two families of sets, which is the problem we tackle in this paper, is not only interesting from a mathematical point of view, it also appears as a subproblem in decision making, market research and temporal evolution analysis applications, as shown in the following examples.

Let us assume that a pharmaceutical company is interested in producing a new drug combining different already existent medicines. If the medicines taken by the patients of the regions where the company is offering its products are known, an interesting and useful combination of medicines can be obtained by solving an intersection problem. In fact, in order to obtain several interesting combinations of drugs, we could consider all the possible pairs of sets of medicines and find their intersection, assuming that a set contains the drugs taken by an individual. Thus, we would know which drugs are simultaneously taken by several patients. Accordingly, we are interested in intersecting one family of sets $\mathcal{F}$ with itself, to obtain the intersection family $\mathcal{I}$. The obtained sets contain the combinations of medicines that are taken by at least two patients. If we eliminate the repeated sets of $\mathcal{I}$ while maintaining their frequency, we will be able to determine a useful combination of drugs that would be used by several patients according to the medication they are using now. This is an example where we are aiming to a useful combined medicament with applications in the pharmaceutical industry. It can be extrapolated to other industries trying to combine food additives, feedstock, textile material, etc. But we could also consider the intersection family to solve more general problems, for instance, to analyze the skills, weaknesses, plants or animals that coexist in different regions by extracting information from real data sets.

Note that, all the mentioned problems involve only one family $\mathcal{F}$ and we are interested in obtaining its intersection family $\mathcal{I} = \mathcal{F} \cap \mathcal{F}$. But there also exists another more general collection of problems where two different families, $\mathcal{F}$ and $\mathcal{F}'$, should be intersected. Most, but not all of the problems where $\mathcal{F} \neq \mathcal{F}'$ involve evolution along time. Related to this other collection of problems, we can be interested in the evolution of the ecosystem of several regions. For instance, we could determine which plants and animals appear together at time t_0, but also after a certain amount of time, at time t_1. Other examples involving the intersection of two different families of sets could be found when trying to determine perdurable skills or weaknesses. But also, when determining the symptomatology evolution, of several patients, in two different stages of an illness or under two different treatments.

*Corresponding author

Email addresses: mfort@ima.udg.edu (Marta Fort), sellares@ima.udg.edu (J. Antoni Sellarès), nacho.valladares@gmail.com (Nacho Valladares)

In some cases we could be interested in the intersection of more than two families. This is what happens in the flock pattern problem [25, 15]. The flock pattern analyzes spatio-temporal trajectories of hundreds of moving elements with hundred-thousands of time steps looking for sets of elements that move together during at least a given number of time steps. The problem is solved by finding a family of sets for each time step. Each set contains the entities whose locations, at the given time step, are simultaneously contained in a disk of predefined radius. To solve the problem, these families have to be intersected in order to obtain the sets of entities that move close enough during the desired number of time steps.

Moreover, this problem in an interesting problem to be solved in parallel for the following three reasons. Firstly, because the nature of the mentioned problems make that the amount of data that has to be handled is big enough; secondly, because it exhibits an inherent high computational complexity, and finally, because finding the intersection family of two families of sets in parallel is important from the mathematic point of view in its own right.

The increasing programmability and high computational rates of GPUs, together with Compute Unified Device Architecture (CUDA) and some programming languages which use this architecture such as 'C for CUDA' or OpenCL, make them attractive to solve problems which can be treated in parallel as an alternative to CPUs. The basis of the CUDA programming model can be found, among many others, in [20, 19, 14]. GPUs are used in different computational tasks where a big amount of data or operations have to be done, whenever they can be processed in parallel. Some recent works, in different fields ranging from numeric computing operations and physical simulations to bioinformatics and data mining, provide demanding algorithms that take advantage of the GPU parallel processing [22, 9, 13, 14, 16].

In this paper, we present an efficient GPU-based parallel approach, designed under CUDA architecture, for computing the intersection family of two large families of unsorted sets (further details are given in next Sections). Even though there are a lot of previous work related on sets intersection, see Section 2, this is the first time that the problem of finding the intersection family of two families of sets is specifically addressed in a paper, either using the GPU or the CPU. We also provide an efficient parallel GPU strategy to remove the repeated sets of a family of sets maintaining, if needed, their frequency. The complexity analysis of the presented algorithm together with experimental results obtained with its implementation, showing the efficiency and scalability of the approach, are also provided.

The paper is structured as follows. We start with the formal definitions of the intersection family of two families of sets and the related work, in Section 2. In Section 3, a brief global overview of our GPU algorithm is provided. In Section 4, we explained the approach to obtain the intersection family of two families of sets, including the strategy for removing the empty and repeated sets while maintaining their frequency. The complexity analysis and the experimental results, obtained with the implementation of our algorithms are given and discussed in Sections 5 and 6, respectively. Finally, in Section 7 conclusions are given.

2. Formal definition and previous work

Let $\mathcal{F} = \{S_0, \cdots, S_{k-1}\}$ and $\mathcal{F}' = \{S'_0, \cdots, S'_{k'-1}\}$ be two families of non-empty finite unsorted sets over a domain D. Finding the complete intersection family of two families of sets $\mathcal{F}$ and $\mathcal{F}'$ is to find the family $\mathcal{I}_c = \{S_0^0, \cdots, S_0^{k'-1}, \cdots, S_{k-1}^0, \cdots, S_{k-1}^{k'-1}\}$ with $k{\cdot}k'$ sets, where S_i^j are the intersection $S_i \cap S'_j$ of set S_i of $\mathcal{F}$ and set S'_j of $\mathcal{F}'$. Notice that, $\mathcal{I}_c$ may contain empty and repeated sets, and each set is identified so that we could know to which sets $S_i \in \mathcal{F}$ and $S'_j \in \mathcal{F}$ it corresponds to.

We also define the problem of finding the intersection family $\mathcal{I}$ of the families $\mathcal{F}$ and $\mathcal{F}'$, which is to find the family $\mathcal{I}$ of all the non-empty sets, which are the intersection $S_i \cap S'_j$ of a set S_i of $\mathcal{F}$ and a set S'_j of $\mathcal{F}'$. In this case, although the intersection of two different pairs of sets, each pair formed by a set of $\mathcal{F}$ and another of $\mathcal{F}'$, can provide the same intersection set, we only include it once in the family $\mathcal{I}$.

Figure 1 shows a table representing the complete intersection family $\mathcal{I}_c$ with the intersection of the sets of the families $\mathcal{F}$ and $\mathcal{F}'$ over the domain D and the resulting intersection family $\mathcal{I}$ after eliminating empty and repeated sets.

$\mathcal{F} \mid \mathcal{F}'$	$S'_0 = \{1,4\}$	$S'_1 = \{1,5,4\}$	$S'_2 = \{4,0,2,3\}$	$S'_3 = \{5,3,4\}$
$S_0 = \{3,0,1,2\}$	{1}	{1}	{0,2,3}	{3}
$S_1 = \{5,1\}$	{1}	{1,5}	{ }	{5}
$S_2 = \{2,0,3\}$	{ }	{ }	{0,2,3}	{3}
$S_3 = \{3,4\}$	{4}	{4}	{3,4}	{3,4}
$S_4 = \{1,3,2,5\}$	{1}	{1,5}	{2,3}	{3,5}
$S_5 = \{1\}$	{1}	{1}	{ }	{ }

$\mathcal{I} = \{\{1\},\{3\},\{4\},\{5\},\{5,1\},\{3,2\},\{4,3\},\{3,5\},\{0,2,3\}\}$

Figure 1: Intersection of two families.

In the reminder of the paper, we consider that the input families are different and that do not contain empty sets. However, our approach can be easily adapted to the case of two arbitrary input families. Note that, if one single family is considered, thus, $\mathcal{F} = \mathcal{F}'$, we have that $S_i \cap S_j = S_j \cap S_i$ and the complete intersection family will be $\mathcal{I}_c = \{S_0^0, \cdots, S_0^{k-1}, S_1^1, \cdots, S_1^{k-1}, \cdots, S_{k-1}^{k-1}\}$. Consequently, instead of k^2 sets, thanks to the commutative property of the intersection of sets, it only has $k \cdot (k+1)/2$ sets. In Figure 1, the part of the table below the main diagonal would not be considered.

The parallel strategy presented here is specially designed to intersect two large families of unsorted sets. For instance, each family would have hundred thousands of sets of, at most, hundreds of elements each. The presented strategy is not developed to simultaneously intersect a small number of very large sets, which is what has been widely studied in the existent papers. In fact, there are plenty of papers studying problems involving the intersection of sets, but, they are placed in very different contexts from the one considered in this paper. An important number of papers, obtain the intersection of two, and some times several, sorted sequences of sets. This problem has important applications in Web search engines, and has been widely studied theoretically [10, 5, 6, 3, 12], and experimentally [11, 4, 7, 12, 8]. There also exist several papers finding the intersection set in parallel [26, 27, 1, 2] by using the GPU. Moreover, in 2010, Hoffman presented and studied the maximal intersection query. Given a query set and a family of sets, they provide the member of the family having maximal intersection with the query set. Note that, the result of all these algorithms is a single set and never a family of sets which is what we are interested in. The intersection family of two families of sets is needed in [15] where the flock pattern is solved in parallel, the algorithm used there to obtain the intersection family, which is not described in detail in that paper, is the one we present in detail in this paper.

In the current paper, apart from dealing with finding the intersection family of two families of sets, we also eliminate the repeated sets of the obtained complete intersection family. The problem of eliminating the repeated sets of a family of sets has been previously studied in [23, 18].

3. GPU algorithm overview

In this section, we give the general idea of the algorithm which is explained in detail in Section 4. We are interested in a work efficient parallel strategy which directly computes the not empty intersection sets. It avoids testing each pair of sets, $S_i \in \mathcal{F}$ and $S_j \in \mathcal{F}'$, to determine whether S_i^j does or does not have elements. On the contrary, the used strategy allows to add an element e in S_i^j if and only if $e \in S_i$ and $e \in S_j$, and consequently $e \in S_i^j$. Thus, instead of looking for the intersection of each set $S_i \in \mathcal{F}$ with each set $S'_j \in \mathcal{F}'$, we start by storing for each element of the domain, the indices of the sets containing that element. These lists of indices of sets where each element appears are stored in the so called apparitions vectors. We compute the two apparitions vectors A and A' associated to the families of sets $\mathcal{F}$ and $\mathcal{F}'$, respectively. From these apparitions vectors we can determine the total number of elements that will define the complete intersection family $\mathcal{I}_c$. In fact, each element e of the domain will appear exactly in $c_e \cdot c'_e$ sets, where c_e and c'_e are the number of indices of sets associated to the element e in the apparitions vectors A and A', respectively. Finally, we consider one thread per element of the complete intersection family which determines its corresponding element e of the domain and its intersection set S_i^j, then it stores e in the first empty position of S_i^j. After obtaining the complete intersection family, which in general contains many empty sets, we proceed to eliminate the empty sets. We also provide a strategy to eliminate duplicated sets while maintaining their frequency, if desired. Thus, at the end of the process we have the intersection family $\mathcal{I}$ with non empty nor repeated sets.

This is the idea of our algorithm, however, it starts with a preprocess to reduce the initial domain. Since there may be many elements of D that are not contained neither in $\mathcal{F}$ nor in $\mathcal{F}'$, it makes no sense to consider them when computing the apparition vectors of $\mathcal{F}$ and $\mathcal{F}'$. Thus, we start by eliminating the elements of D that do not appear at least once in $\mathcal{F}$ and once in $\mathcal{F}'$ and then, we find the intersection family.

4. Reporting the intersection family

In this section, we describe the several steps required to compute the intersection family $\mathcal{I}$ of two input families $\mathcal{F}$ and $\mathcal{F}'$ whose sets do not need to be sorted in any specific order. The section is organized as follows. First, we explain the reduction of $\mathcal{F}$ and $\mathcal{F}'$ by discarding the domain elements that do not appear in both families (Section 4.2). Then, we provide the method to compute the complete intersection family $\mathcal{I}_c$, finding the intersections between the two reduced families (Section 4.4), which is done by using two main structures: the GPU family structure (Section 4.1) and the apparitions vector structure (Section 4.3). Next, we present the algorithm to eliminate the empty sets (Section 4.5.1) and posteriorly the one to eliminate the duplicated sets (Section 4.5.2). Finally, we explain how the intersection family $\mathcal{I}$ is reported (Section 4.7). Since it may happen that we had not enough GPU memory to compute the whole intersection family at once, we also provide a way to obtain the intersection family by parts (Section 4.6). All the 1D-arrays we use during the whole process are stored in GPU global memory, except when it is specifically mentioned.

4.1. Storing the input families in the GPU

Without loss of generality, we can assume that the elements of the domain are indexed by n non-negative integers and consider $D = \bigcup S_i = \{0, \ldots, n-1\}$.

To represent a family $\mathcal{F}$ in the GPU, we use a data structure that consists of three 1D-arrays, the family, counting and positioning arrays denoted f, c and p, respectively. Array f stores the m elements contained in the k sets of the family, starting with the elements of S_0 and ending with the elements of S_{k-1}. The counting and positioning arrays have size k and store the sets cardinality and the position of f where each set starts, respectively. With this structure we can easily access to any set or element using the fact that the set S_i starts at position $p[i]$ of f and is stored in $c[i]$ consecutive positions of f. An example is shown in Figure 2 where the elements of S_2 start at $f[6]$ and are stored in 3 consecutive positions.

As we will see in the following sections, to solve the intersection problem we need, several times, the index i of the set $S_i \in \mathcal{F}$ to which an element of f, $f[x]$, corresponds to. With the presented family structure, this index i can be obtained with a dichotomic search on p. However, we could avoid using this dichotomic search if we add an extra array s of size m so that $s[x]$ is the index of the set to which $f[x]$ corresponds to, i.e $f[x] \in S_{s[x]}$. By using s, we would know in constant time the index of the set corresponding to an element of f, but we would use m extra integer values and we would have to take care to maintain the array s updated. As we will see next, the sets in $\mathcal{F}$ and the array f are reorganized several times and s would have to maintain the correspondence between the elements of f and the set index. In the rest of the paper, since the amount of memory needed to solve the problem without using s is big enough, we do not use s and we determine the index of the thread using a dichotomic search. However, if desired this option could also be considered.

We create one structure for the input family $\mathcal{F}$ and another for the input family $\mathcal{F}'$. We generate the arrays c, p and f representing $\mathcal{F}$ and c', p' and f' representing $\mathcal{F}'$ in the CPU while the input families are read, and then we transfer them to GPU memory.

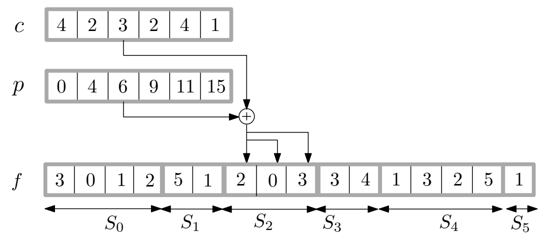


Figure 2: GPU data structure containing a family of six sets

4.2. Reducing the input families

To speed-up the computation of the intersection of $\mathcal{F}$ with $\mathcal{F}'$, we start with the following reduction process. First, we determine the domain $\overline{D} = (\bigcup S_i) \cap (\bigcup S'_j)$. Next, we find the new sets $\overline{S}_i = S_i \cap \overline{D}$ and $\overline{S}'_j = S'_j \cap \overline{D}$, for each set S_i of $\mathcal{F}$ and S'_j of $\mathcal{F}'$. The families determined by the sets $\overline{S}_i$ and $\overline{S}'_j$ are denoted $\overline{\mathcal{F}}$ and $\overline{\mathcal{F}}'$, respectively. Since $\overline{S}_i \cap \overline{S}'_j = S_i \cap S'_j$, instead of intersecting the family $\mathcal{F}$ with $\mathcal{F}'$ we intersect $\overline{\mathcal{F}}$ with $\overline{\mathcal{F}}'$. However, before starting the intersection computation, the empty and duplicated sets within each family are removed by using the algorithms presented in Section 4.5.1 and Section 4.5.2. In the case that we are interested in the frequency of the sets in the input families, we will also store an auxiliary array associated to each family, t and t', storing the frequency of each set in the corresponding family.

To determine $\overline{D}$ we create a 1D array $\overline{d}$ of size n initialized to zero. A parallel kernel is launched with one thread per element in $\mathcal{F}$. Each thread idx sets to one the position $f[idx]$ of $\overline{d}$. Then, we do the same with $\mathcal{F}'$ but setting to two those positions which already have a one. When we are done, $\overline{d}[e]$ contains a two value, whenever the element e is present at least once in each family. Finally, $\overline{d}$ is rewritten by setting those positions with a two to one, and the rest to zero. Additionally, an array $\overline{d}_{off}$ is created as the prefix sum of $\overline{d}$. It is used to maintain the correspondence between D and $\overline{D}$.

In the next step, we remove from $\mathcal{F}$ and $\mathcal{F}'$ all the elements e with $\overline{d}[e] = 0$. To do this, we launch a kernel with one thread per element in $\mathcal{F}$. Each thread idx checks whether $\overline{d}[f[idx]]$ is zero. In such a case, the thread determines the index i of the set the element idx belongs to, by locating idx in p using a dichotomic search. Next, by using atomic operations, $c[i]$ is decremented in 1 indicating that the set S_i has one less element and $f[idx]$ is set to -1. Additionally, we have a counter to determine the number of completely eliminated sets which is incremented when $c[i]$ has been set to zero.

Then, we can allocate $\overline{c}$, $\overline{p}$ according to the new sizes. The array $\overline{c}$ is filled, in parallel, with the non-zero elements of c, by using k threads. Because it is done in parallel the order of the resulting values in $\overline{c}$ does not correspond to the order in c. Thus, we maintain an auxiliary array, c_c, storing the correspondence between the positions of $\overline{c}$ and c. Then, $\overline{p}$ is computed as the prefix sum of $\overline{c}$, the new size of the family $\overline{m}$ is determined and $\overline{f}$ allocated. Finally, $\overline{f}$ is filled, in parallel, with m threads. Using $\overline{p}$, c, p, c_c and f we discard the elements of f that have been set to -1 while we copy the others to $\overline{f}$. The thread idx considers the element $e = f[idx]$. If $f[idx] = -1$ nothing is done, otherwise, it determines the index j of the set $S_j \in \mathcal{F}$ where $f[idx]$ belongs. Then, the thread considers $j_n = c_c[j]$, the index of the old-set $S_j \in \mathcal{F}$ in $\overline{\mathcal{F}}$, and e is stored in $\overline{f}[p[j_n] + c[j_n]]$ after decrementing $c[j_n]$ by one, with an atomic operation. Moreover, instead of storing e in $\overline{f}$, the element e is shifted by setting it to $\overline{d}_{off}[e]$, in order to keep on having as new domain $\overline{D} = \{0, \ldots, \overline{n} - 1\}$. Finally, $\overline{f}$ stores the elements of $\overline{\mathcal{F}}$.

This process is performed on the input families $\mathcal{F}$ and $\mathcal{F}'$. When we are done, we can guarantee that all the elements of $\overline{D}$ are present at least once in both families. Finally, we remove the empty and duplicated sets of $\overline{\mathcal{F}}$ and $\overline{\mathcal{F}}'$ by adapting the algorithm explained in Sections 4.5.1 and 4.5.2, respectively.

Abusing notation, from now on, the new sets, the corresponding families and the reduced domain are still denoted S_i, S'_j, $\mathcal{F}$, $\mathcal{F}'$ and D, respectively. We also still denote k and k' the number of sets in $\mathcal{F}$ and $\mathcal{F}'$, m and m' the total number of elements contained in $\mathcal{F}$ and $\mathcal{F}'$ which correspond to the sum of their sets cardinalities and n the number of elements in D.

4.3. Computing the apparitions vector

Given a family $\mathcal{F}$ with k sets, the apparitions vector is a structure containing n lists of set indices determining the sets where each element $e \in D$ appears. The list $A[e]$ contains the indices of those sets containing the element $e \in D$, thus, a set index i with $0 \leq i \leq k-1$ is stored in $A[e]$, whenever $e \in S_i$. Notice that, the total number of elements stored in A is exactly m. An example of such vector is showed in Figure 3, where the element 1 appears in the sets S_0, S_1, S_4 and S_5, so the indices 0, 1, 4 and 5 are stored in $A[1]$.

The apparitions vector A is stored in the GPU using the data structure previously used to store a family of sets. In this case, the structure, referred as the apparitions vector, consist of three 1D-arrays denoted a, c_A and p_A. The array a, of size m, stores the elements of the lists conforming A, the counting array c_A, of size n, stores the number of sets containing each element or equivalently the number of elements contained in each list $A[e]$. Finally, the positioning array p_A stores in $p_A[e]$ the position of the array a where the list $A[e]$ starts. As it happens in the family structure, in the apparitions structure we could also add an extra array s_A of size m storing for each element of a the domain element to which it is associated, i.e. if $a[x] \in A[e]$ then $s[x] = e$.

To compute the apparitions vector structure, we first compute the array c_A, initializing its elements to zero and launching a parallel kernel with n threads. The counting array is next obtained by using m threads, one per element in f. The thread with index idx reads its corresponding element $e = f[idx]$ and increments $c_A[e]$ by one. In Figure 3, $c_A[2]$ is incremented three times by three different threads. Note that, since many threads may modify the same memory position at the same time, the thread race condition can lead to incorrect results. To avoid this, we use the atomic operations where memory accesses are done with no thread interferences. The positioning array of A, p_A, is the prefix sum of c_A and is computed using the GPU scan algorithm.

To store the apparitions vector in the GPU array a, we run a CUDA kernel with one thread per element in f. The thread with index idx reads its corresponding element in f, $e = f[idx]$, determines, by using p and the index idx, the set S_j where e belongs to, and stores j in a. The set index j is determined by locating the thread index idx in p with a dichotomic search. To store j in a we re-initialize c_A to zero, during the process it is used to maintain the number of indices that have been already stored in each list of A. Thus, j is stored as the $c_A[e]$ element of the list $A[e]$ which corresponds to the position $p_A[e] + c_A[e]$ of a, and consequently when storing j the value of $c_A[e]$ has to be incremented by one. Since several threads can be storing set indices in the same list $A[e]$ at the same time, the value of $c_A[e]$ is obtained and incremented by one by using an atomic operation.

4.4. Determining the complete intersection family

To compute the intersections between the families $\mathcal{F}$ and $\mathcal{F}'$ we use their apparitions vector structures A and A'. The main idea is to count and report the elements

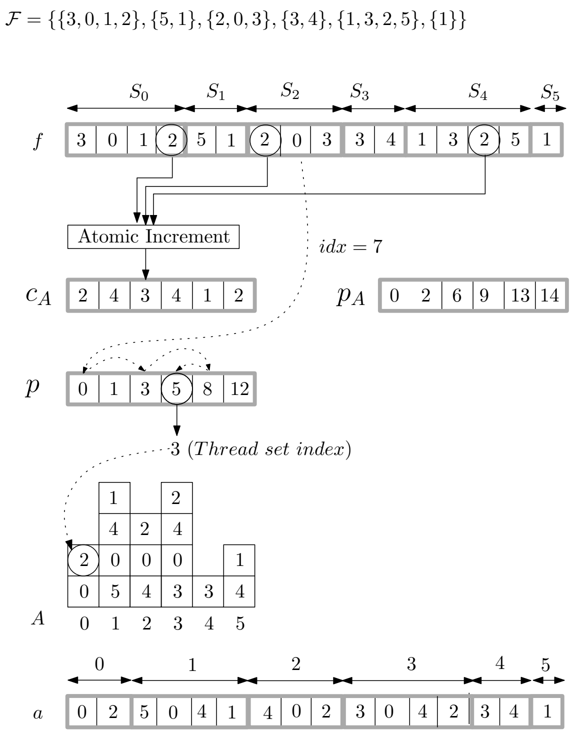


Figure 3: Apparitions vector a construction

in common between each pair of sets. This is done by using the fact that an element e is contained in the sets stored in $A[e]$ and $A'[e]$, and thus, e is contained in, and only in, the $c_A[e] \cdot c'_A[e]$ intersection sets obtained by intersecting two sets whose indices are, one in $A[e]$, and the other in $A'[e]$. The process is parallelized not only by storing the element e in each of the resulting intersection sets in parallel, but also considering the n elements of the domain in parallel.

In the first step, we construct the apparitions vectors A and A' following the algorithm presented in Section 4.3. Then, we count the number of intersection elements, corresponding to the sum of the cardinalities of the intersection sets. With this aim we create the array c_t, of size n, with $c_t[e] = c_A[e].c'_A[e]$. That is, $c_t[e]$ is the total number of intersection sets where the element e appears. Then, p_t is created as the prefix sum of c_t. From them we know the number of intersection elements, which is denoted m_{int}.

Next, we compute the intersection sets. The idea is to run a CUDA kernel with as many threads as intersection elements so that each thread stores the element it represents in the intersection set it corresponds to. The process takes place in two steps. First, we count the cardinality of each intersection set, and then, the intersection sets are obtained.

To determine the intersection sets cardinality, we create the counting intersection matrix c_{int} of size $k \cdot k'$ where $c_{int}[i][j]$ stores the number of elements in common between sets S_i and S'_j. In the GPU, c_{int} is linearized and stored in a $1D$ array. After initializing c_{int} to zero, we launch a kernel with m_{int} threads. We consider $c_A[e] \cdot c'_A[e]$ threads with consecutive indices for each element $e \in D$, thus, each thread is responsible for a specific intersection element $e \in S_i^j$. The thread starts determining the element e it corresponds to by locating the thread index idx in $p_t[e]$ with a dichotomic search. Then, once e is known, $p_t[e]$ is used to determine the positions of $A[e]$ and $A'[e]$ where the set indices i and j are stored, in fact $i = (idx - p_t[e])/c_{A'}[e]$ and $j = (idx - p_t[e]) - i \cdot c_{A'}[e]$. Finally, $c_{int}[i][j]$ is incremented by one using an atomic operation. An example of this process is shown in Figure 4.

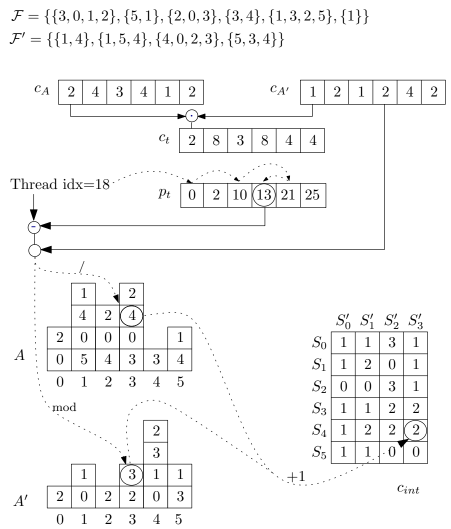


Figure 4: Counting intersections matrix obtention

In the second step, we create p_{int} as the prefix sum of the $1D$ array c_{int}. Finally, we allocate f_{int}, an array of size m_{int} to store the intersection sets. This is filled with the process used to compute c_{int}, but now instead of incrementing $c_{int}[i][j]$ in one, the element e is stored in the corresponding position of f_{int}. This is done by using $p_{int}[i][j]$ and c_{int} re-initialized to zero to know how many elements have already been stored in the set S_i^j obtained. When the process ends, arrays $c_{int}, p_{int}, f_{int}$ represent the complete intersection family $\mathcal{I}_c$.

Note that, since at the beginning of the process we have started eliminating duplicates after reducing the families, the frequency of each of the S_i^j sets of $\mathcal{I}_c$ is not necessarily one. In fact it is given by $t[i] \cdot t'[j]$, where $t[i]$ and $t'[j]$ store the frequencies of the sets S_i and S'_j, respectively. Thus, we can compute the frequency of each set of $\mathcal{I}_c$ and store them in a frequency array, t_{int}, associated to the complete intersection family $\mathcal{I}_c$. The frequency values can be stored in t_{int} when the first element of S_i^j is handled. Consequently, t_{int} has to be initialized to 0 and the frequency of a set S_i^j is set to $t[i] \cdot t'[j]$ the first time that the corresponding position of c_{int} is incremented.

We want to mention, that the presented strategy to find the complete intersection family $\mathcal{I}_c$ is a robust strategy whose computational cost does not directly depend on the input families $\mathcal{F}$ and $\mathcal{F}'$. However, $\mathcal{I}_c$ can also be obtained with an alternative strategy whose computational cost directly depends on frequency of the elements of D in $\mathcal{F}$ and $\mathcal{F}'$. This other strategy would perform well, in terms of computational cost, when the products $c_A[e] \cdot c'_A[e]$ for all $e \in D$ do not differ much among them. The idea is that we consider one thread per element in D which is responsible to store e in all the intersection sets S_i^j where it is contained. Thus, it has to consider all the pairs obtained with $i \in c_A[e]$ and $j \in c'_A[e]$. This strategy may also have some variants, we can consider m threads and each thread considers one element e of one set $S_i \in F$ and stores e in the $c'_A[e]$ intersection sets S_i^j, $j \in c'_A[e]$, where it is contained. We discard these options because the work done for each thread depends on the number of sets where the element e appears. By analyzing the provided motivational examples, we can easily see that there are some elements of D which may appear much more frequently than the others. Thus, we consider that a strategy whose computational cost depends on the frequency of the elements in the families is not appropriate to solve the general problem tackled in this paper.

4.5. Obtaining the intersection family

The complete intersection family $\mathcal{I}_c$ has exactly the $k \cdot k'$ sets obtained by intersecting each set of $\mathcal{F}$ with each one of the sets of $\mathcal{F}'$. But some of them, usually many of them, correspond to empty or repeated sets. Depending on the problem we are solving, this enlarges c_{int}, and p_{int} unnecessarily.

In fact, by using a GPU parallel strategy we can not avoid finding the complete intersection family $\mathcal{I}_c$ to obtain the intersection family $\mathcal{I}$ without empty nor repeated sets. Since there does not exist efficient dynamic memory in the GPU, we have to allocate enough memory space to handle the worst case, which would contain $k \cdot k'$ different intersection sets.

4.5.1. Removing empty sets

To remove the empty sets from c_{int} and p_{int}, we use an auxiliary array c_e of size $k \cdot k'$ initialized to zero. While we compute the array c_{int}, we set to 1 the value of a position of c_e when a thread increments for the first time the value of the corresponding position of c_{int}, it is, when the stored value of this position of c_{int} is a 0. Thus, at the end $c_e[i][j] = 1$ if $S_i^j \neq \emptyset$. Then, we compute p_e as the prefix sum of c_e. The number of non empty sets is obtained from c_e and p_e, by summing up the value of the last position of c_e with the one of the last position of p_e.

Then, a new counting and positioning arrays c'_{int}, p'_{int} are allocated according to the number of non empty sets. By using a parallel kernel with $k \cdot k'$ threads we obtain c'_{int} storing the non zero elements of the $1D$ array c_{int} in c'_{int}. Now, using p_e, we can compute c'_{int} in parallel maintaining the order of the intersection sets. Finally, p'_{int} is computed as the prefix sum of c'_{int}. Notice that, f_{int} needs no modifications because empty sets are not present here.

We denote the complete family once the empty sets have been removed by $\mathcal{I}_r$. Abusing notation, in next Section, c_{int}, p_{int}, f_{int} and k_{int} will refer to $\mathcal{I}_r$ and not to the complete intersection family $\mathcal{I}_c$.

4.5.2. Removing duplicated sets

Our parallel approach to remove the duplicated sets is based on the two following observations: 1) two equal sets have the same cardinality; 2) equal sorted sets are stored in consecutive positions in a lexicographically sorted family, according to any total order of the domain elements.

By using these observations, we remove the duplicated sets in three steps. First, we split the intersection family $\mathcal{I}_r$ into groups of sets of equal cardinality. Then, we sort the sets elements and the sets of each group in lexicographical order. Finally, we remove the duplicated sets of each group checking whether two consecutive sets are equal. Even though the process may seem not very efficient, it does the minimum work required to check the element uniqueness according to the lower bound proved in [3].

Step 1: Splitting sets by cardinality

We sort c_{int} using a GPU parallel algorithm [21] which can reorganize an extra array, in this case p_{int}, maintaining the correspondence between the auxiliary and the sorted array. Thus, we have c_{int} sorted by increasing cardinality and p_{int} reorganized accordingly. In this way, we can access to the different sets of f_{int} in increasing cardinality order without problems. Then, we reorganize f_{int} so that the sets of equal cardinality become grouped together. It is done by obtaining $\tilde{p}_{int}$ as the prefix sum of c_{int}, re-initializing c_{int} to 0 and using the array $\tilde{f}_{int}$ of size m_{int} which will store the intersection family sets in increasing cardinality order. We consider m_{int} threads and the thread idx determines the value to be stored in $\tilde{f}_{int}[idx]$, as follows. It starts determining the index j of the set to which $\tilde{f}_{int}[idx]$ corresponds, by finding the first position of $\tilde{p}_{int}[j] \geq idx$ with a dichotomic search. Then $\tilde{f}_{int}[idx] = f_{int}[p_{int}[j]+c_{int}[j]^{old}]$, where $c_{int}[j]^{old}$ is the old value of $c_{int}[j]$ obtained when the value stored in this position is incremented by one using an atomic operation. Thus, we can consider that $\mathcal{I}_r$ has been split into groups of sets of the same cardinality, and we denote $\mathcal{I}_g$ the family containing only the sets of $\mathcal{I}_r$ of cardinality g.

Steps 2 and 3 are performed on each group $\mathcal{I}_g$. We use c_g, p_g and f_g to denote the corresponding counting, positioning and family arrays, meanwhile k_g and m_g represent the number of sets and elements of $\mathcal{I}_g$.

Step 2: Lexicographically sorting the sets of equal cardinality

To have equal sets in consecutive positions after sorting $\mathcal{I}_g$ in lexicographical order, we need to have the elements within each set also sorted. However, we do not need to sort the elements according to a particular order of the domain, any common ad-hoc order on the elements works. Thus, we use a simple and efficient technique, called weak sorting, which sorts the elements of all the sets according to an arbitrary total order computed on the fly [23, 18]. Weak sorting works as follows: a) associate to each element all the sets where it is contained; b) traverse the domain elements and place them to the sets they belong to. In this way, at the end, all the sets contain their elements stored in lexicographic order, according to the total order computed during the weak-sorting. We denote $\mathcal{I}_g^w$ the family of sets obtained from $\mathcal{I}_g$ after the weak sorting process.

The weak sorted family $\mathcal{I}_g^w$ is computed as follows. We use an already existing and implemented GPU sorting algorithm [21] to sort all the sets. However, this GPU sorting algorithm is very efficient sorting very large sets and in this case we have to sort many short sets. In order to take the maximum benefit of the GPU sorting algorithm, we compute $\mathcal{I}_g^r$ by mapping the elements in each set to new integers, guaranteing that the integers stored in S_i^r are smaller than all those in S_j^r, whenever $i < j$. To use not too big integers in $\mathcal{I}_g^r$, we use two arrays $\mathcal{I}_g^{min}$ and $\mathcal{I}_g^{max}$ with the minimum and maximum element stored in each set of $\mathcal{I}_g$. They are obtained by using a kernel m_g threads and atomic operations. There exist the shuffle strategies to extract the global maximum or minimum of an array [24], but we are not using them because we are finding k_{int} local minimums and maximums. Next, we store in $\mathcal{I}_g^{dif}$ the differences between $\mathcal{I}_g^{max}$ and $\mathcal{I}_g^{min}$ by using k_g threads. These differences are accumulated by using the exclusive scan algorithm and the result is again stored in $\mathcal{I}_g^{dif}$. Finally, $\mathcal{I}_g^r$ is computed by adding to each element of each set S_i the value $\mathcal{I}_g^{dif}[i] - \mathcal{I}_g^{min}[i]$. The family $\mathcal{I}_g^r$ is considered as a unique set by concatenating the different sets and $\mathcal{I}_g^r$ is sorted with a GPU sorting algorithm. Finally, the initial elements are recovered by substracting the previously added value. Thus, finally, we have the family $\mathcal{I}_g^w$ with the elements of the sets sorted by increasing order. An example of the process is shown in Figure 5.

Next, we lexicographically sort the sets of the family $\mathcal{I}_g^w$ using the radix sort algorithm, see Figure 6 a). We name i-column, the set determined by the i-th element of a each set of $\mathcal{I}_g^w$. The radix sort algorithm sorts the sets of $\mathcal{I}_g^w$ column by column. First, we sort the sets by the 0-column, then, the resulting sets of the first step are sorted by the 1-column, we continue until the last column is considered. It is important to remark that the sorting of each column must be stable in order to guaranty that the radix sort works.

We sort each column using a GPU stable sort algorithm [21]. Note that, the sorting of the i-column depends on the result of the sorting of the previous j-column for all $j < i$. Thus, in order to obtain the correct result we should reorganize the whole structure $\mathcal{I}_g^w$ after sorting each column. Since commonly $f_g \gg p_g$, instead of reorganizing all the elements of f_g at each iteration, we just reorganize p_g and the GPU sorting algorithm sorts according to it. Once the process ends, the sets are lexicographically sorted according to their elements which are also sorted.

Step 3: Eliminating duplicates

Once we have the elements within each set sorted and the sets of the family $\mathcal{I}_g^w$ lexicographical sorted, equal sets are stored in consecutive positions. Thus, we only need to compare adjacent sets to eliminate duplicates, see Figure 6 b). To determine if two consecutive sets, S_i^w and S_{i+1}^w, are equal, we compare the element e_i^j at

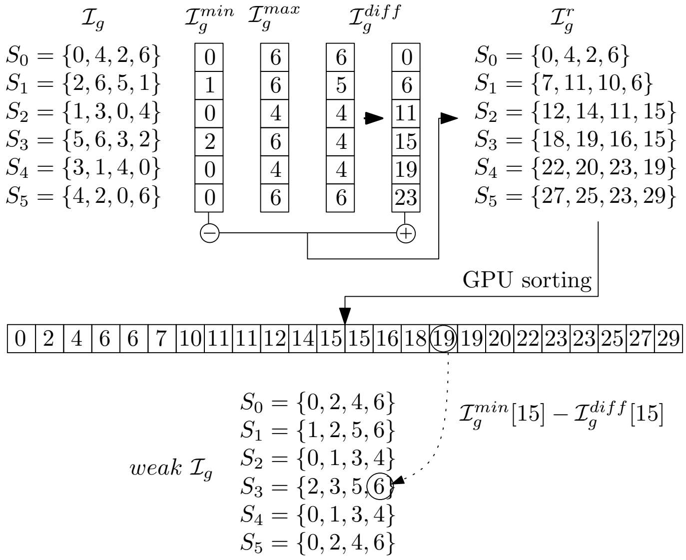


Figure 5: Sorting the sets elements

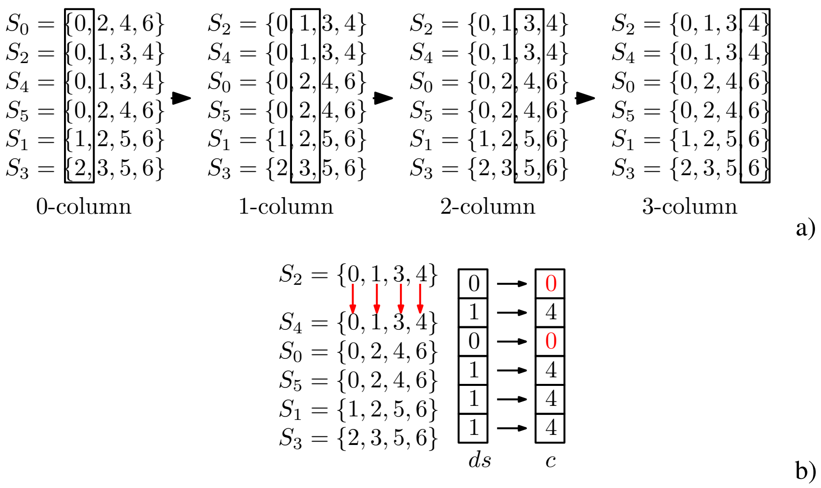


Figure 6: a) Lexicographically sorting $\mathcal{I}_g$ b) Removing duplicates

position j of the set S_i^w with the element e_{i+1}^j at position j of the set S_{i+1}^w, for each $0 \leq j < g$. If and only if we find two different elements, the set S_i^w is different from set S_{i+1}^w.

Since all the elements and all the sets can be compared independently, checking whether two sets are equal is a very parallelizable process. Duplicated sets are eliminated by first creating an array ds of size k_g, initialized to zero. Then, we launch a kernel with $m_g - g$ threads assigning an element per thread, except for the elements of the last set which is not compared to any other set. Each thread compares its corresponding element, e_i^j with e_{i+1}^j. If they are different $ds[i]$ is set to one indicating that the set S_i^w and S_{i+1}^w are different. When the process finishes, those positions in ds containing a zero correspond to the duplicated sets. We then set to zero the corresponding positions of the original counting array c indicating that they have to be removed. Notice that, if there exist equal sets only the last apparition of the set is marked with a one in ds. This last set maintains its original counting value, meanwhile the other counting values have been set to zero.

When steps 2 and 3 have been computed for each group, the original $\mathcal{I}_r$ structure is updated according to the new size. The counting and positioning arrays c and p are resized and recomputed as it was done when removing empty sets. Array f is also recomputed to remove duplicates by using both the original and the resized positioning and counting arrays and the original array f.

The frequency array t, which stores the frequency that each set appears in $\mathcal{I}_r$, can be determined from $ds[i]$ using a prefix sum-like algorithm where at the end, the vector ds, denoted by ds_e, contains in each position the value $ds_e[i] = (1 - ds[i])(1 + ds_e[i-1])$ for $i > 0$ and $ds[0] = 1 - ds[0]$. After computing $ds_e[i]$, the frequency array t can be computed while c and p are resized. The number of apparitions of the set whose last apparition is stored in the position $j > 0$ of the counting array is $1 + ds_e[j-1]$, and the frequency of the set S_0 is 1 if and only if $ds_e[0] = 0$.

Note that, the eliminating duplicates process can be done with families whose sets have associated a frequency array t_{int} (Section 4.6). In this case, the array t_{int} has to be reorganized according to the counting and positioning arrays during the initial steps of the strategy. In this case, the frequency of each set is obtained by using t_p, the inclusive prefix sum of t_{int}, the frequency of the set whose first apparition is stored in the position $i + 1$ of the counting array is $t_p[j] - t_p[i]$, where i and j, $i < j$, are again two consecutive positions of the counting array with a non zero value.

When the process ends, the intersection family $\mathcal{I}$ without empty nor repeated sets together with their frequency, if needed, has been obtained.

4.6. Dealing with huge families

Since the initial 1D-arrays c_{int}, t_{int} and p_{int} representing the complete intersections family $\mathcal{I}_c$ have size $k \cdot k'$, depending on the input families, the GPU memory may not be sufficient to store them. When this happens the intersection family can be obtained by parts in the following way.

The apparitions vector A is created as before, meanwhile, the apparitions vector A' and the structure representing the complete intersections family $\mathcal{I}_c$ are created in several iterations. For each iteration, we determine two set indices α and β with $0 \leq \alpha < \beta < k'$, so that we can store in the GPU all the needed 1D-arrays to represent both, the subfamily $\mathcal{F}'_{[\alpha,\beta]} = \{S'_j \text{ with } \alpha \leq j \leq \beta\}$ and the intersection family $\mathcal{I}_{c[\alpha,\beta]}$ obtained as the intersection of $\mathcal{F}$ with $\mathcal{F}'_{[\alpha,\beta]}$. Then $\mathcal{I}_{c[\alpha,\beta]}$ is computed in the GPU and once the empty and duplicated sets have been removed, we transfer the output vectors $c_{int}[\alpha,\beta]$ and $f_{int}[\alpha,\beta]$ associated to $\mathcal{I}_{[\alpha,\beta]}$ to the CPU. All the GPU arrays, except for those needed to store the family $\mathcal{F}$, are deleted to restart the algorithm with the next subfamily of $\mathcal{F}'$.

In CPU memory, we store three vectors of arrays, v_c, v_n and v_f. In v_c, we store the arrays $c_{int}[\alpha,\beta]$ copied from the GPU, so that $v_c[i]$ contains the array $c_{int}[\alpha,\beta]$. Similarly, $v_n[i]$ and $v_f[i]$ contain the arrays $t_{int}[\alpha,\beta]$ and $f_{int}[\alpha,\beta]$, respectively, obtained in the i-th iteration. When all the sets of $\mathcal{F}'$ have been considered, v_f may contain duplicated sets appearing in different arrays that should be eliminated while updating their frequency array. This is again done in the GPU, with this aim, we build the $\mathcal{I}$ structure allocating the 1D-arrays c_{int},t_{int}, p_{int} and f_{int} in the GPU memory. The arrays of v_c are stored one after the other in c_{int} and those of v_n and v_f in t_{int} and f_{int}, respectively. The array p_{int} is computed as the prefix sum of c_{int}. Finally, duplicates are eliminated with the previously provided algorithm. Note that there is no need to store in the GPU the whole intersection family. We only need to be able to load all the sets of a given cardinality in the GPU at the same time, because they are the ones susceptible to be repeated.

4.7. Reporting intersection sets

According to Section 4.2, the input families $\mathcal{F}$ and $\mathcal{F}'$ have been preprocessed so the elements of the postprocessed families only contain elements present in both families. Thus, the elements and the sets have been reindexed according to the domain $\overline{D} = (\bigcup S_i) \cap (\bigcup S'_j)$.

In order to restore the elements to their original values, we use the already created array $\overline{d}_{off}$ to reindex the elements of the family $\mathcal{I}$. We launch a parallel kernel with one thread per element in $\mathcal{I}$, each thread reindexes its corresponding element value according to $\overline{d}_{off}$. Then, we only have to copy from the GPU to the CPU memory the structure representing the family $\mathcal{I}$ together with the final frequency array.

In order to determine the intersection set of any two original sets and locate it in the output of our algorithm in constant time, we can proceed as follows. During the preprocess (Section 4.2) we use two integer arrays of size m and m', to store a -1 in its i-th position when the original set S_i is deleted from the family, and its corresponding new index in the reduced family when S_i is not deleted. Thus, if in the position of this array corresponding to the original set S_i there is a -1, any intersection set where this S_i appears is the empty set. Otherwise, the intersection set S_i^j corresponding to the intersection of two original sets which have been indexed as sets S_a and S'_b during the preprocess, corresponds to set with index $a \cdot k' + b$ of the complete intersection family $\mathcal{I}_c$. Finally mention that, we can locate this set in $\mathcal{I}$ by transferring some extra information to the CPU or making a few changes in the GPU algorithm.

5. Complexity analysis

When we analyze the complexity of a GPU algorithm, we should take into account the total work, the thread work, the number of accesses to memory and the transferred values between CPU and GPU. The total work is the total number of instructions realized during the whole algorithm. The thread work is the number of operations done by a single thread and gives an idea of the degree of parallelism obtained. Despite the parallelization does not decrease the total work complexity of the algorithm, it has an important effect on the running times. Finally, a GPU algorithm performance also depends on the number of memory accesses and on the transferred values from CPU to GPU and vice versa.

Table 1 contains the GPU complexity analysis of each step of our approach, here we analyze each part of the algorithm independently.

The total work of the initial step is $O(m \log(k \cdot n) + m' \log(k' \cdot n) + n + k)$. The terms of the total work with a log factor are done by threads doing the log term work each, meanwhile the other factors are done by threads doing $O(1)$ work each.

Concerning the determination of the intersection family, the total work is $O(n + m \log k + m' \log k' + k \cdot k' + m_{int} \log n)$. Again, the terms of the total work with a log factor are done by threads doing the log term work each, meanwhile the other factors are done by threads doing $O(1)$ work each. Notice that in this case, n, m, k and k' refer to the already reduced families, thus, they are not the original values but those obtained after the reduction process.

Removing the empty sets is a completely parallelized part of the process, it has a total work of $O(k \cdot k')$ work, which is done with the same amount of threads doing $O(1)$ work each. Finally, removing duplicates and counting their frequency requires splitting the sets by cardinalities which is done with a standard sorting algorithm with $O(m_{int} \log m_{int})$ total work, which is the minimum required time according to the element uniqueness lower bound [3]. Next, the sets are handled per groups of equal cardinality. Sorting the sets elements, of the sets of cardinality g, requires a total work of $O(m_g \log m_g)$. Lexicographically sorting the sets is done with a sorting algorithm sorting g times k_g elements, representing a total work of $O(g k_g \log k_g)$. Once all the cardinalities have been considered, the total work done is $O(m_{int} \log m_{int})$ which is needed to sort the sets. Once they are already sorted, eliminating duplicates and obtaining the frequencies requires $O(m_{int} \log k_{int})$ total work with $O(\log k_{int})$ work per thread. Obtaining the output family without duplicates takes $O(m_{int} \log k_{int})$ total work with $O(m_{int})$ threads doing $O(\log k_{int})$ work each.

In the CPU, we only create the counting an positioning arrays of the input families, thus, it takes $O(m + m')$ time.

Summarizing, the provided approach has a CPU time complexity of $O(m + m')$, transfers $m + m' + 2(k + k')$ integer values to the GPU and $m_{int} + k_{int}$ to the CPU. Concerning the GPU total work is of $O(m \log(k \cdot n) + m' \log(k' \cdot n) + n + k \cdot k' + m_{int} \log m_{int})$ which is, as well, the number of global memory accesses. Notice that, the algorithm is quasilinear with respect to the number of elements in the input and output families. The other factors of the total work complexity are not directly related to the input nor the output of the problem, but the parts of the algorithm where they appear are completely parallelized, doing $O(1)$ work per thread, which can not be improved.

The execution time of our parallel algorithm when p threads are considered is $O((m \log(k \cdot n) + m' \log(k' \cdot n) + n + k \cdot k' + m_{int} \log m_{int})/p)$. In the best case, this time is $O(\log k + \log k' + \log n + \sum \log k_g)$, this happens when $p = O(mx)$ and mx is the maximal number of used threads, which corresponds to the maximum of the values n, m, m', $k \cdot k'$ and m_{int}.

Note that, by using the extra array s in the family and

		Total work	Thread work	Mem. access
$\overline{D}$ determination	d initialization to 0	$O(n)$	$O(1)$	$O(n)$
	elements in $\mathcal{F}$	$O(m \log k)$	$O(\log k)$	$O(m \log k)$
	elements in $\mathcal{F}'$	$O(m' \log k')$	$O(\log k')$	$O(m' \log k')$
	$\overline{D}$ obtention	$O(n)$	$O(1)$	$O(n)$
	d_{off} computation	*n values prefix sum*		
$\overline{\mathcal{F}}$ computation	$\overline{m}$ determination	$O(m \log k)$	$O(\log k)$	$O(m \log k)$
	new c and c_c	$O(k)$	$O(1)$	$O(k)$
	new positioning	*$\overline{k}$ values prefix sum*		
	new $\mathcal{F}$	$O(m \log n)$	$O(\log n)$	$O(m \log n)$
$\overline{\mathcal{F}'}$ computation	*equivalent to the previous one with k', m' and $\overline{k}'$*			
Apparition vector A	c_A initialization	$O(n)$	$O(1)$	$O(m)$
	c_A obtention	$O(m)$	$O(1)$	$O(n)$
	p_A computation	*n values prefix sum*		
	a	$O(m \log k)$	$O(\log k)$	$O(m \log k)$
Apparition vector A'	*equivalent to the previous case with k' and m'*			
$\mathcal{I}$ computation	c_t initialization	$O(n)$	$O(1)$	$O(n)$
	p_t obtention	*n values prefix sum*		
	c_{int} initialization	$O(k \cdot k')$	$O(1)$	$O(k \cdot k')$
	c_{int}, c_e obtention	$O(m_{int} \log n)$	$O(\log n)$	$O(m_{int} \log n)$
	t_{int} obtention	$O(m_{int})$	$O(1)$	$O(m_{int})$
	p_{int} obtention	*$k \cdot k'$ values prefix sum*		
	f_{int} obtention	$O(m_{int} \log n)$	$O(\log n)$	$O(m_{int} \log n)$
Empty sets removal	c_e initialization	$O(k \cdot k')$	$O(1)$	$O(k \cdot k')$
	p_e obtention	*$k \cdot k'$ values prefix sum*		
	new c_{int}, t_{int} and p_{int}	$O(k \cdot k')$	$O(1)$	$O(k \cdot k')$
Cardinalities sorting	sorted c_{int}, t_{int} and p_{int}	$O(k_{int} \log k_{int})$	*sorting k_{int} values*	
*Sets sorting	$\mathcal{I}_g^{min}$ and $\mathcal{I}_g^{max}$	$O(m_g \log k_g)$	$O(\log k_g)$	$O(m_g \log k_g)$
	$\mathcal{I}_g^{dif}$ computation	$O(k_g)$	$O(1)$	$O(k_g)$
	$\mathcal{I}_g^{dif}$ accumulation	*k_g values prefix sum*		
	$\mathcal{I}_g^{r}$ computation	$O(m_g \log k_g)$	$O(\log k_g)$	$O(m_g \log k_g)$
	$\mathcal{I}_g^{w}$ obtention	$O(m_g \log m_g)$	*sorting m_g values*	
*Lexicographical order	*GPU stable sorting of g columns with k_g values each*			
*Duplicates elimination	ds initialization	$O(k_g)$	$O(1)$	$O(k_g)$
	ds computation	$O(m_g)$	$O(1)$	$O(m_g)$
	c_{int} and t_{int} update	$O(k_g)$	$O(1)$	$O(k_g)$
Final $\mathcal{I}$ computation	final c_{int}	$O(k_{int})$	$O(1)$	$O(k_{int})$
	final p_{int}	*k_{int} values prefix sum*		
	final f_{int}	$O(m_{int} \log k_{int})$	$O(\log k_{int})$	$O(m_{int} \log k_{int})$
	final t_{int}	$O(k_{int})$	$O(1)$	$O(k_{int})$
Reporting $\mathcal{I}$	reindexing f_{int}	$O(m_{int})$	$O(1)$	$O(m_{int})$

Table 1: Complexity analysis (* refers to the group of sets of cardinality g)

apparitions structure, with the element to index correspondence, the log k, log k', log n and log k_{int} factors would disappear. Consequently, by using this auxiliary array, the $O(\log x)$ work done per thread would become $O(1)$. On the contrary, if we use the alternative strategies provided to compute $\mathcal{I}_c$, the work done by each thread in this part of the algorithm would increase. If one thread per element of D was considered each thread would have $O(k \cdot k')$ work, and if on thread per element in $\mathcal{F}$ was considered it would become $O(k')$.

6. Results

In this section, we analyze the results obtained with the GPU implementation of our algorithms and compare the response of our GPU implementation with a CPU implementation that solves the intersection family problem.

The experimental results are obtained using an i7-2630QM CPU and 8GB RAM with a Tesla K40 graphics card. The families of sets are stored in binary files providing a very efficient way to read the families from file and to load them to the GPU.

We use two very different types of families, we first use families containing very similar sets and then families with sets with random uniformly distributed elements on the domain. Thus, we check not only the algorithm goodness, but also its response depending on the input families by analyzing the algorithm behavior in two very different settings. For both experiments, we intersect two families of sets with $k = k'$, the number of sets k and k' vary from 5.000 to 50.000 and each set contains between 100 and 150 elements of a domain of 10^5 elements.

Families of similar sets. These families contain sets that do not differ much between them, such sets may represent the entities forming flock [15], the medicaments of patients suffering a specific illness, the animals or plants habiting in different regions sharing a climate, etc. The experimental results obtained with these families are presented in Figure 7.

In Figure 7 a), we show the running times, in seconds, of the main steps of the algorithm: the input families storage and reduction, the computation of the complete intersection family I_c including the apparitions vectors, the determination of $\mathcal{I}_r$ removing the empty sets and of $\mathcal{I}$ eliminating the duplicated sets. The accumulated value, corresponding to the columns height, gives the total running time of the algorithm.

As expected, the total running times increase with the size of the input families, in this case, they vary between 0.5 and 1.2(s). Note that, the most expensive steps are the reduction of the input domain and the duplicated sets removal. The running times to compute the complete intersection family I_c and to remove the empty sets are small with respect to the others. The time needed to reduce the initial families takes from 0.12 to 0.5(s). However, trying to reduce the intersection family is worth, because it may provide an interesting decrease in the memory requirements as it happen in this case.

Note that, the reduction of the input family speedups the whole process and allows handling bigger families without problems because it reduces significantly the space needs, as it can be seen in Figure 7 b) and c). These figures show the decrement in the number of elements of the domain, n, and of the whole families, m and m', when the reduction step is realized. In this case, there are many elements of the domain that do not appear in any set, and the number of elements that are simultaneously present in both families is really small. Thus, the reduction in the domain cardinality is huge, and the preprocess reduces significantly the amount of memory needed in the following steps.

Similarly, reducing the complete intersection family $\mathcal{I}_c$ to the intersection family $\mathcal{I}$, also diminishes the space needed to store $\mathcal{I}$, as it is corroborated in Figure 7 d) and e). Figure 7 d) shows the number of sets used to store the complete family $\mathcal{I}_c$ and the ones needed to store I_r, the intersection family after removing the empty sets. Eliminating empty sets is worth in terms of memory and does not provide important effects on the running times, it takes no more than 0.001(s). Finally, removing duplicated sets is time-expensive but it may produce important benefits in memory saving, as shown in Figure 7 d) where the number of non empty sets and the total number of elements of $\mathcal{I}_r$ are compared with the number of sets and elements defining the intersection family $\mathcal{I}$.

Families of uniformly distributed random sets. In this second experiment, we consider two random uniformly distributed families, and thus, they may completely differ between them. This is a typical setting when analyzing problems from a mathematical point of view. In this case, the different steps of the algorithm have a completely different behavior with respect to the families of similar sets, as can be seen from the information provided in Figure 8.

Figure 8 a) presents, as in the previous experiment, the running times of the main steps of the algorithm. With this kind of families, the total running times vary between 8 and 99 seconds. Now, the most expensive steps are the computation of the intersection sets and

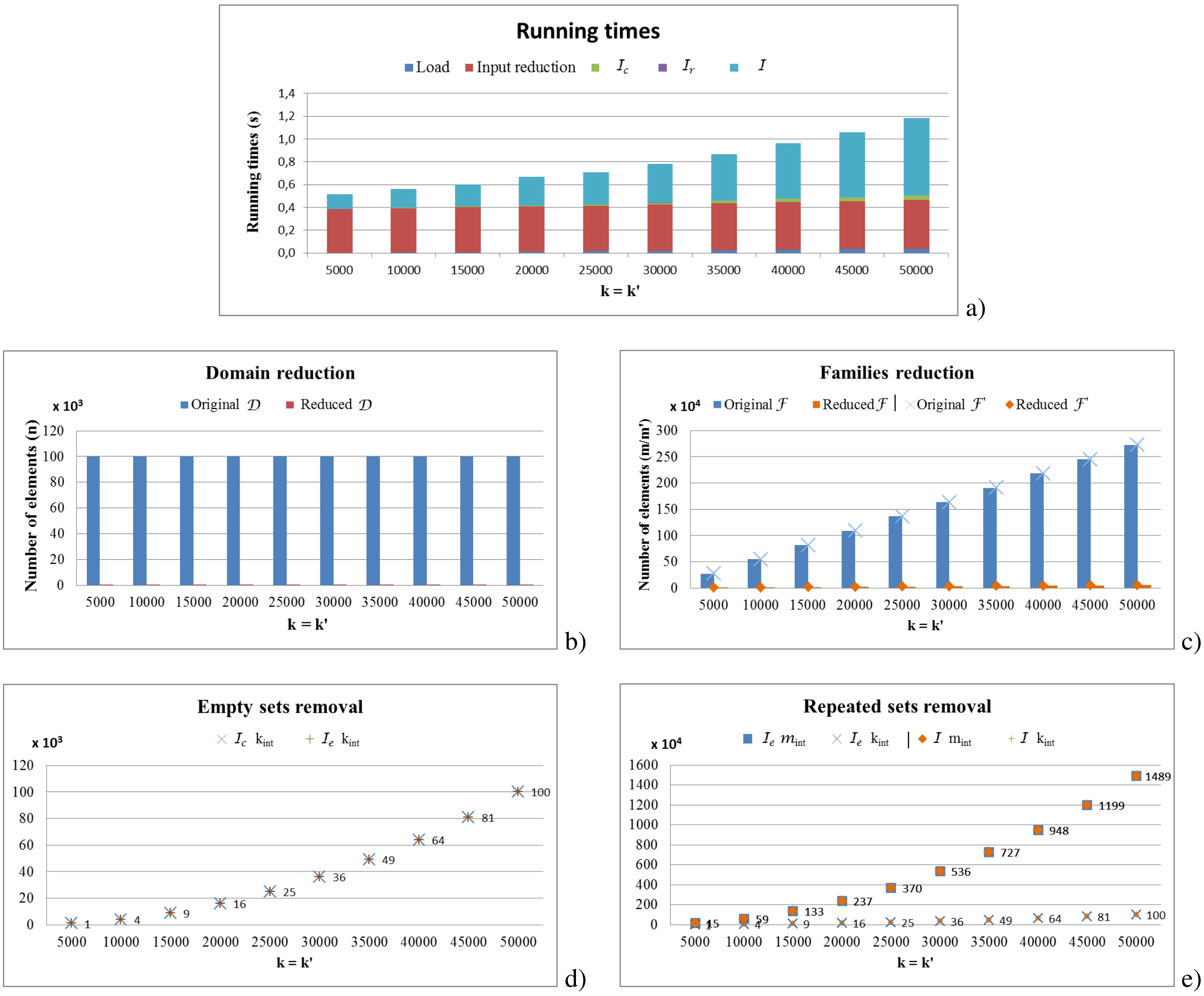


Figure 7: Families of similar sets experimental results

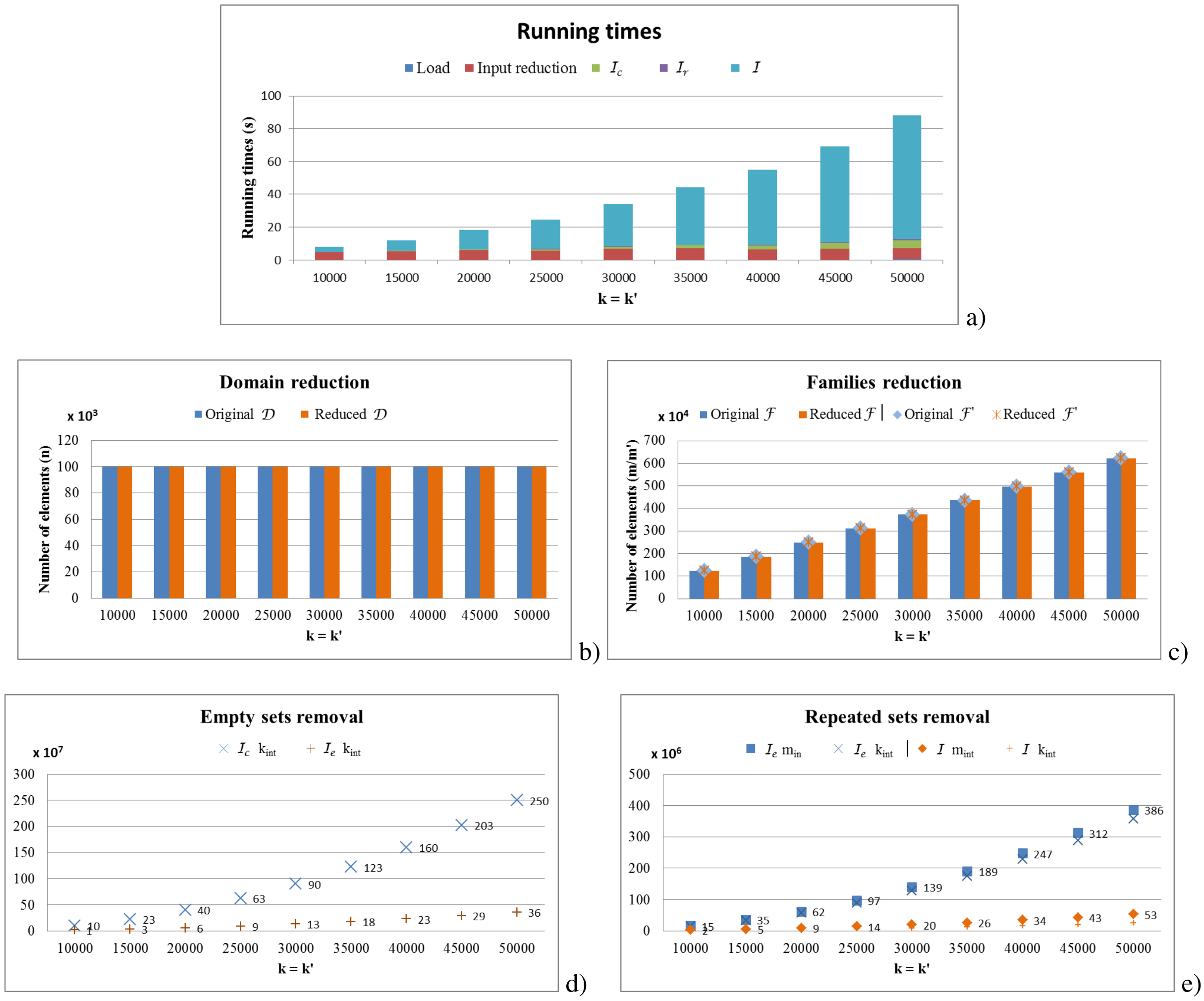


Figure 8: Families of uniformly distributed random sets experimental results

the elimination of duplicates. In Figure 8 b) we can see that, contrarily to what happens in the first case, the gain obtained with the reduction of the input families is inappreciable. However, in this case this preprocess represents from 5 to 6(s).

On the other hand, the output has again many duplicated an empty sets, as it can be seen in Figure 8 c) and d). Hence, eliminating them provides a big positive impact in the output size. The number of sets of $\mathcal{I}$ with respect to the number of sets of $\mathcal{I}_c$ is reduced in a 98% in average, and the number of stored elements in a 85%. Thus, the amount of memory needed to store $\mathcal{I}$ is much smaller, more than a 90% smaller, than the memory needed to store $\mathcal{I}_c$.

Thus, from both experiments we can conclude that trying to reduce the input families is worth. In fact, it may produce an important reduction in the domain and in case that this reduction is irrelevant, the increment in the running times is not very significant with respect to the total running time. Concerning the reduction of $\mathcal{I}_c$ to $\mathcal{I}$, it depends on the posterior use we have to do of the intersection family. But in general in terms of usability and chances to extract information, $\mathcal{I}$ will be much more interesting than $\mathcal{I}_c$.

CPU-GPU speedup ratios. Finally, in order to compare our GPU algorithm response with the fastest CPU algorithm. In order to find an appropriate CPU algorithm, we use standard algorithms to compute sets intersections. We also implement a CPU version of the proposed GPU strategy, which avoids checking for empty intersection sets. Among the standard algorithms, we have considered the C++ Standard, the boost and the thrust libraries. The boost libraries only allow interval sets intersections, meanwhile the C++ Standard library and the thrust ones handle sets intersections. In both cases, we have tried to store the families as vectors of vectors, as sets of vectors and as sets of sets, using sets implies sorting the elements while they are inserted in the sets.

In Figure 9, we can see the best CPU running times obtained with the implemented CPU versions, the GPU running times presented in Figure 7 and Figure 8 associated to the right vertical axis, and the correspondent CPU versus GPU speedup ratio associated to the left vertical axis. All the running times are obtained by using a i7-2630QM CPU, 8GB RAM. The best CPU running times have been obtained by considering sets of vectors of the C++ Standard library and the intersection of sets of the thrust libraries. Figure 9 a) corresponds to the running times and speedups when considering the families with similar sets, and Figure 9 b) to the uniformly distributed random families.

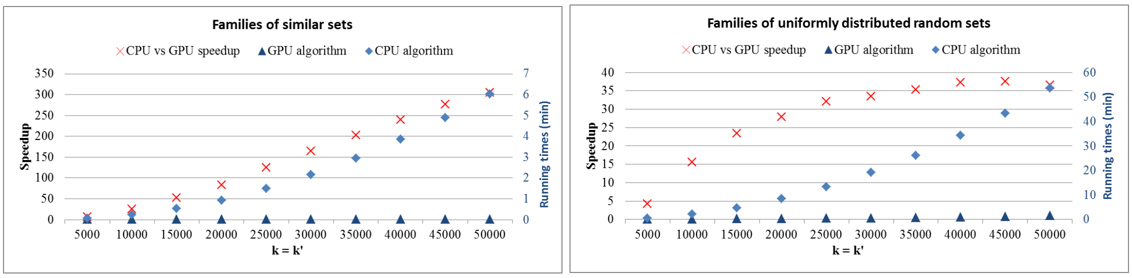
Note that, we present the running times needed to obtain $\mathcal{I}$ which are not much different from the running times needed to obtain $\mathcal{I}_c$ or $\mathcal{I}_r$ when the CPU is used. In fact, in some cases, it is even quite faster obtaining $\mathcal{I}$ than $\mathcal{I}_c$ or $\mathcal{I}_r$, this is because the amount of memory needed to store $\mathcal{I}$ is much smaller than the needed to store $\mathcal{I}_c$ or $\mathcal{I}_r$, and storing big amounts of data slows down the algorithm. For instances, finding $\mathcal{I}$ takes from 3.6(s) to 6.04(min) when similar sets are considered, and from 0.51 to 53.70(min) when uniformly distributed sets are used, and the running times to compute the corresponding $\mathcal{I}_r$ families become from 3.8(s) to 5.96(min) and from 0.47 to 43.6(min), respectively.

According to Figure 9, we can conclude that our GPU implementation is much faster and scalable than the CPU one. In fact, it could be expected because our GPU algorithm does not do extra work and most of the work done is done in parallel. According to the figure, the GPU algorithm is from 7 to 306 times faster than the CPU algorithm when considering families with similar sets, and from 4 to 37 times faster when considering uniformly distributed random families. If instead of computing $\mathcal{I}$ we consider the times needed to compute $\mathcal{I}_r$ the obtained speedup ratios vary between 10 and 709 for families of similar sets and between 5 and 205 for families of uniformly distributed random sets. The speedups obtained when considering $\mathcal{I}_r$ are bigger than the ones obtained to compute $\mathcal{I}$ because the increment of time when deleting duplicates with respect to the total running of the GPU strategy is bigger than the corresponding increment when considering the CPU algorithm. Thus, the ratio between the CPU and GPU running times needed to compute $\mathcal{I}_r$ is bigger than the ratio obtained considering the running needed to obtain $\mathcal{I}$.

We also want to remark that, the presented GPU algorithm provides exact results that always match with the ones obtained with the CPU. It is expected from the algorithm, because there are no precision errors that could affect the obtained results, but we have also checked it by comparing the GPU and the CPU algorithms output.

7. Conclusions

In this paper, we presented an exact parallel GPU-based approach, designed under CUDA architecture, for computing the intersection of two families of sets. We have provided a parallel GPU-based approach for removing duplicated sets in a family of sets, and main-



a) b)

Figure 9: CPU vs GPU algorithm running times

taining the frequency of each set in the original family of sets.

The complexity analysis of the algorithms together with experimental results has been presented. In both cases, the degree of parallelism of the provided strategy is shown, first theoretically and next experimentally. We studied the response of the intersection algorithm depending on different characteristics of the input and output families. Two very different kinds of synthetic families have been considered. In the first case, the families contain sets that do not differ much among them. In the second case, the families contain random uniformly distributed sets. The experimental results showed the scalability and efficiency of the approach specially compared with the CPU implementation as the speedup ratios corroborate. We also studied the computational and memory savings reached with the input domain reduction and the empty and duplicated sets removal.

8. Acknowledgments

We thank the reviewers for their valuable comments, which helped us to improve the paper.

Work partially supported by the Spanish Ministerio de Economía y Competitividad under grant TIN2014-52211-C2-1-R .

[1] R. R. Amossen, R. Pagh, A new data layout for set intersection on gpus, in: IPDPS, IEEE, 2011, pp. 698–708.

[2] N. Ao, F. Zhang, D. Wu, D. S. Stones, G. Wang, X. Liu, J. Liu, S. Lin, Efficient parallel lists intersection and index compression algorithms using graphics processing units, PVLDB 4 (8) (2011) 470–481.

[3] R. A. Baeza-Yates, A fast set intersection algorithm for sorted sequences, in: S. C. Sahinalp, S. Muthukrishnan, U. Dogrusöz (Eds.), CPM, Vol. 3109 of Lecture Notes in Computer Science, Springer, 2004, pp. 400–408.

[4] R. A. Baeza-Yates, A. Salinger, Experimental analysis of a fast intersection algorithm for sorted sequences, in: M. P. Consens, G. Navarro (Eds.), SPIRE, Vol. 3772 of Lecture Notes in Computer Science, Springer, 2005, pp. 13–24.

[5] J. Barbay, C. Kenyon, Adaptive intersection and t-threshold problems, in: D. Eppstein (Ed.), SODA, ACM/SIAM, 2002, pp. 390–399.

[6] J. Barbay, Optimality of randomized algorithms for the intersection problem, in: A. A. Albrecht, K. Steinhöfel (Eds.), SAGA, Vol. 2827 of Lecture Notes in Computer Science, Springer, 2003, pp. 26–38.

[7] J. Barbay, A. López-Ortiz, T. Lu, A. Salinger, An experimental investigation of set intersection algorithms for text searching, Journal of Experimental Algorithmics 14 (7) (2009) 7–24.

[8] V. Boa, Experimental Comparison of Set Intersection Algorithms for Inverted Indexing, in: ITAT 2013 Proceedings, CEUR Workshop Proceedings, Vol. 1003, pp. 5864

[9] N. Coll, M. Fort, N. Madern, J. A. Sellarès, Multi-visibility maps of triangulated terrains, International Journal of Geographical Information Science 21 (10) (2007) 1115–1134.

[10] E. D. Demaine, A. López-Ortiz, J. I. Munro, Adaptive set intersections, unions, and differences, in: D. B. Shmoys (Ed.), SODA, ACM/SIAM, 2000, pp. 743–752.

[11] E. D. Demaine, A. López-Ortiz, J. I. Munro, Experiments on adaptive set intersections for text retrieval systems, in: A. L. Buchsbaum, J. Snoeyink (Eds.), ALENEX, Vol. 2153 of Lecture Notes in Computer Science, Springer, 2001, pp. 91–104.

[12] B. Ding, A.D. Konig, Fast Set Intersection in Memory, Proc. of the 37th. International conference of VLDB Endowment, Vol. 4, N.4, 2011, pp. 255-266

[13] M. Fort, J. A. Sellarès, Approximating generalized distance functions on weighted triangulated surfaces with applications, Journal of Computational and Applied Mathematics 236 (15) (2012) 3461–3477.

[14] M. Fort, J. A. Sellarès, N. Valladres, Finding extremal sets on the GPU, Journal of Parallel and Distributed Computing, 74(1) (2014), 1891–1899.

[15] M. Fort, J. A. Sellarès, N. Valladres, A parallel GPU-based approach for reporting flock paterns, International Journal of Geographical Information Science, 28(9) (2014), 1877–1903.

[16] M. Fort, J. A. Sellarès, N. Valladres, Computing and visualizing popular places, Knowledge and Information Systems, 40(2) (2014), 411–437.

[17] B. Hoffmann, M. Lifshits, Y. Lifshits, D. Nowotka, Maximal intersection queries in randomized input models, CoRR abs/1004.0092.

[18] F. Henglein, Generic top-down discrimination for sorting and partitioning in linear time, J. Funct. Program. 22 (3) (2012) 300–374.

[19] NVIDIA, Nvidia CUDA C Programming Best Practices Guide 4.1, Tech. rep., Nvidia Corporation (2011).

[20] Nvidia, CUDA Programing Guide 4.1, Tech. rep., Nvidia Cor-

poration (2012).
[21] N. Corporation, Nvidia CUDA 4.1 SDK samples,(2012).
[22] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, T. J. Purcell, A survey of general-purpose computation on graphics hardware, Computer Graphics Forum 26 (1) (2007) 80–113.
[23] R. Paige, Efficient translation of external input in a dynamically typed language, in: IFIP Congress (1), 1994, pp. 603–608.
[24] http://on-demand.gputechconf.com/gtc/2013/presentations/S3174-Kepler-Shuffle-Tips-Tricks.pdf
[25] M. R. Vieira, P. Bakalov, V. J. Tsotras, On-line discovery of flock patterns in spatio-temporal data, in: D. Agrawal, W. G. Aref, C.-T. Lu, M. F. Mokbel, P. Scheuermann, C. Shahabi, O. Wolfson (Eds.), GIS, ACM, 2009, pp. 286–295.
[26] D. Wu, F. Zhang, N. Ao, F. Wang, X. Liu, G. Wang, A batched gpu algorithm for set intersection, in: ISPAN, IEEE Computer Society, 2009, pp. 752–756.
[27] D. Wu, F. Zhang, N. Ao, G. Wang, X. Liu, J. Liu, Efficient lists intersection by cpu-gpu cooperative computing, in: IPDPS Workshops, IEEE, 2010, pp. 1–8.