Journal of Information Technology Research
Volume 15 • Issue 1

A New Fast Intersection Algorithm for Sorted Lists on GPU

Faiza Manseur, Ahmed Ben Bella Oran 1 University, Algeria*
https://orcid.org/0000-0002-1249-4576

Lougmiri Zekri, Ahmed Ben Bella Oran 1 University, Algeria

Mohamed Senouci, Ahmed Ben Bella Oran 1 University, Algeria

ABSTRACT

Set intersection algorithms between sorted lists are important in triangle counting, community detection in graph analysis, and in search engines where the intersection is computed between queries and inverted indexes. Many studies use GPU techniques for solving this intersection problem. The majority of these techniques focus on improving the level of parallelism by reducing redundant comparisons and distributing the workload among GPU threads. In this paper, we propose the GPU Test with Jumps (GTWJ) algorithm to compute the intersection between sorted lists using a new data structure. The idea of GTWJ is to group the data, of each sorted list, into a set of sequences. A sequence is identified by a key and is handled by a thread. Intersection is computed between sequences with the same key. This key allows skipping data packets in parallel if the keys do not match. A counter is used to avoid useless tests between cells of sequences with different lengths. Experiments on the data used in this filed show that GTWJ is better in terms of execution time and number of tests.

KEYWORDS

GPU, GTWJ, Intersection, Scalar Algorithm, Sequences, Sorted Sets, SVS

INTRODUCTION

The calculation of the intersection between sets is an important operation in numerous application domains, such as information retrieval for text search engines, graph analytics for triangles counting and community detection and database systems. It serves to create indexes, especially inverted ones, by computing for every term, in the documents database, the set of documents in which it appears. Also, it allows search engines to find the responses to queries by computing their intersections with the global inverted index. Since 1971, when the first work has appeared, many techniques have been published in order to accelerate intersection time between sets, especially between sorted ones. At the basis, these techniques use well defined data structures and exploit hard components of machines. Logically, time execution depends on the manner by which the comparisons between the elements of the entries are done. In this context, three approaches are adopted to this end:

- Accelerate sequential processing time using data structures that are well adapted to requirements;

DOI: 10.4018/JITR.298325 *Corresponding Author

- Exploit graphics processors (GPUs) and multi-core processors (CPUs) to exploit advantage of the parallelism they offer; or
- Exploit new plate-forms or frameworks, like map-reduce, for distributing the workload.

Since current machines are equipped with GPU cards that can easily be integrated into them and since they offer a high-level parallel environment at low cost. We propose a new GTWJ (GPU Test With Jumps) algorithm for calculating the intersection between sorted lists. The objectives of our proposition are multiple. Our first objective is to reduce the number of tests by avoiding comparing parts that will not necessarily be shared between the lists in the inputs. To achieve this objective, we modify the structure of the lists to be compared in order to avoid these unnecessary tests. Our second objective is to exploit the parallelism of the graphics card to speed up calculation times by reducing redundant comparisons and distributing the workload evenly among GPU threads. Thus, we have implemented our solution with the CUDA language which provides direct access to the graphics processor programming. This programming language has a complete instruction set, such as double precision calculation, and allows thousands of threads to be simultaneously used. This solution is compared to other solutions (Hwang & Lin, 1971; Demaine et al., 2001; Inoue et al., 2014). We present and comment these solutions in this paper.

The rest of the article is structured as follows. Section 2 provides related works; so, a state of the art is given here, it covers the set of algorithms which are published since 1971. We extend this section by presenting some research papers which applied intersection between lists to other fields. In section 3, we present three important algorithms, as we will compare our proposition with them. Section 4 gives details of our solution. Experiments are presented and commented in section 5. Section 6 concludes this paper.

Related Work

The intersection between lists is widely studied because of its importance in several areas, such as the creation of indexes in search engines and the detection of frequent patterns. The first work, which treated the intersection between two sorted lists was published in the year 1971 (Hwang & Lin, 1971). Authors proposed a linear algorithm for merging two ordered lists. They propose that if the lengths of the two lists to be compared are relatively equal, a linear comparison would be an excellent way to merge the two entries. If the difference between the two lengths is large, then the use of other methods, such as dichotomy (called also binary search) or interpolation, to locate the elements of the small list in the large one speeds up the merging process. This same work was updated in (Hwang & Lin, 1972). Authors conclude that for two sorted lists A et B where n_1 and n_2 are their lengths, respectively, with $n_1 \leq n_2$, the complexity for computing the intersection is $O(n_1+n_2)$. So, if $n_1=n_2=n$ then time complexity is O(2n). It is judicious to see that, exactly (n_1+n_2-1) operations are executed. Brown and Tarjan (1979) used an AVL tree to determine the intersection with the same complexity of (Hwang & Lin, 1972). Pugh (1990a) has proposed the use of skiplists to represent the data in memory and for the calculation of the intersection. These skiplists allow to avoid testing of non-shared parts. In this paper, only an average number of comparisons is given in the merging algorithm. Authors of (Demaine et al., 2000) studied the sets intersection, union and difference. They presented a study on a framework about optimization. A set of adaptive algorithms is given for calculating the three operations. These same authors have proven that their algorithms, according to a certain parameter α, are efficient. The study was applied essentially on balanced B-trees. This paper presented especially the algorithm adaptive which was compared to SvS (for Small vs Small) in (Demaine et al., 2001). SvS is very useful for extremely large data like in search systems. The details of SvS will be given in the next section. Author of (Barbay & Kenyon, 2002) have shown that the algorithm adaptive of (Demaine et al., 2000) is optimal and extended the work by giving a new algorithm with threshold for calculating the intersection of several lists in parallel. Exactly, for k sorted lists of the entry, authors

studied the problem for detecting the elements which belong to t sets from k, with $t \leq k$. This problem is called t-Threshold Problem. A bound of this problem is given.

Bentley and Yao (1976) has studied the intersection of two sets in the context of unbounded searching problem or in other words the problem of searching an ordered table of infinite size. Authors studied the complexity of many algorithms. They have begun by presenting the Unary Search (B_0) algorithm and its complexity. This algorithm is called as a subroutine in the Binary Search (B_1) and by recursion, they built the k-nested Binary Search (KBS or B_k). However, the complexity of this last algorithm can be too high. In order to remedy to this problem, authors propose the Ultimate algorithm whose objective is to choose appropriate values of k. This work is also called Galloping algorithm (Vladimir, 2013) as for locating an element of B, the binary search returns either it or its position in A. We must note that (Hwang & Lin, 1972) was the first work which has proposed the use of the binary search. It is important to see that the binary search is well only for n_1 too less than n_2 (Zekri et al., 2018).

Inspired by (Bentley & Yao, 1976) and by (Hwang & Lin, 1972), Beaza (Baeza-Yates, 2004) has introduced a new algorithm for calculating the intersection between two ordered sets. This work is called the double binary search, it can be seen as a balanced version of Hwang and Lin's (Hwang & Lin, 1972). For some conditions, this algorithm has a good average case of complexity. Baeza-Yates (2004) suppose two lists A and B where the difference between their two lengths is significant. Recursively, using the dichotomy, (Baeza-Yates, 2004) calculates the median of A and tries to locate it in B, using the dichotomy as well. If this value is found so it will be inserted in the shared set else the theoretical position L of this median is returned. If $L > n_1$ than the execution is stopped. This situation is the best case that can happen. If $L < n_1$ then the algorithm will continue. The computing of the median divides A into two subsets. The median of every part will be recursively binary searched in a corresponded part of A, which is also divided into two parts. The algorithm computes at every time which part is smaller and inverses the searching process. The algorithm ends when a subset is empty.

Even if this algorithm is efficient, but not necessarily rapid, it cannot be used for computing the intersection of more than two ordered sets as the intersection result is not sorted (Zekri et al., 2018). In other words, if we want to computed the intersection between k ordered sets $s_1, s_2, .., s_k$, with $|s_1| \leq |s_2| \leq ... \leq |s_k|$ any algorithm like SvS (Demaine et al., 2001; Barbay et al., 2006) begins by computing the intersection between s_1, s_2 . The result R is always sorted. At every time, the intersection is computed by R and the next S_i (with $3 \leq i \leq k$) and is R updated. So, there is no need to apply a sorting procedure on R. This computation will take more time if (Baeza-Yates, 2004) is applied as R is not ordered.

Baeza-Yates et Salinger in (Baeza-Yates & Salinger, 2005, 2010) have performed experiments to compare (Baeza-Yates, 2004) with the Adaptive algorithm (Demaine et al., 2000, 2001). They showed that if the size A is too small compared to the size B then the dichotomy is an effective way to calculate this intersection. With well values for the length of the smallest set, the algorithm of (Baeza-Yates, 2004) can do better performances than Adaptive algorithm, however, in the information retrieval domain, it is not possible to impose entries with prescribed lengths. In the Experiments in (Zekri et al., 2018) showed that globally SvS make better performances. Bill et al. (2007) have used the word memory as the unit of representation of information. They perform a pre-processing to write the lists in RAM. Using a well-defined hash function, this one was defined initially in (Carter et al., 1978), authors calculate the intersection between two lists. The question in (Carter et al., 1978) was to find a manner for compute the space for representing a set. So (Bill et al., 2007) has took full advantage of this idea as it allows to test bucket of bits in one time. Tsirogiannis et al. (2009) have proposed a partitioning of the input in order to make a load balancing on a multi-core architecture. This work has presented three algorithms for computing the intersection of sorted and unsorted lists. for doing this, authors use a cache-resident micro-index and a hash function.

Tatikonda et al. (2009) have used the multi-core architecture to compute the intersection between lists. As new machines are equipped by many cores, authors exploit the communication via cache-memories, for accelerating the intersection computation of posting lists. To do this, the posting lists

are presented by skiplists and stored by using the PFordelta compression scheme (Zukowski et al., 2006) as skiplists (Pugh, 1990b). The objective of authors of (Ao et al., 2011) was to manage the heavy workload in search engines as they process thousands of queries per second. They implement a binary search algorithm on GPU. They couple linear regression and hash segmentation techniques on GPU.

Ding et al. (2008) proposed a hierarchical representation in memory where they couple hashing with sorting to detect the intersection. The result depends largely on the bit representation of the memory words. Schlegel et al. (2011) used the STTNI (STring and Text processing New Instruction) instructions defined in the Intel SSE 4.2 processor. The intersection is calculated according to the SIMD model on this processor. Their experiments showed that the comparison of two arrays in 8-bit blocks was the best way to achieve high performance. Inoue et al. (2014) proposed a Framework that runs on the base of the SIMD model to minimize the bad connections caused by the test operation. The execution of the test program was controlled at the bottom of the machine. It is a question of checking the progress of the program's ordinal counter. The authors proposed a prediction by advancing several pointers at once. Lemire et al. (2016) proposed the GALLOPING algorithm which, based on the SIMD model, compares 4 pairs of integers represented in 32-bits. This algorithm depends largely on the architecture of the processor on which it is running. GALLOPING can compare lists of unequal lengths. Otherwise, they use another algorithm of the SIMD type for intersection detection. Zhou et al. (2016) proposed an efficient Framework that runs on the GPU where the authors perform an intense process based on the SIMD model that produces an index for a search engine. The authors also proposed a binary hash function to represent the terms.

Fort et al. (2017) proposed a parallel algorithm on the GPU to calculate the intersection between two large families of small lists. They eliminated repeated and empty lists. The problem of list intersection can be encountered in many other fields like detection of communities in large graphs. For example, the computation of the clustering coefficient is based on the detection the detection of triangles as the problem is to define if the neighbor of my neighbor is my neighbor; see (Schank, & Wagner, 2005) for such work. The intersection detection was an important problem of the algorithms forward and edge listing (Latapy, 2008; Alon & Michael,1978). By using bitmaps for filtering out unmatched elements, (Zhang et al., 2020) presents FESIA for searching intersection between small sets. Authors use the SIMD model on modern CPUs.

All these paper deal with le set intersection problem; some of them work on CPU and others on GPU. Some paper use trees and other papers put assign the arrays directly in the memory. The major problem is for us that if we could propose an index so the computing process will be fast. This paper is not about a survey, where we have to give more details about the weaknesses of any cited paper especially since the number of pages is limited, but it is planned in our future works. We propose a new index and a new algorithm which exploits the index in order to compute the intersection in a fast way. More of this, we have compared our work to other works according to number of tests which the important parameter in the intersection computing process; this comparison has not done before.

Main algorithms

This section presents the set of algorithms that we have compared with our solution. We begin by presenting the naive algorithm which is the first algorithm in the field of computing the intersection between ordered sets. SvS algorithm computes the intersection by using the binary search. The parallel algorithm is the third algorithm presented in this section. It has implemented an extended SIMD version of the naive algorithm on GPU.

Scalar Naive Algorithm

The scalar algorithm is the simplest algorithm to search intersections between ordered sets. We refer here by the term 'naive'. It was proposed by (Hwang & Lin, 1971, 1972). It takes as input two sorted tables and returns the common elements between them. It has been proven that its best case is when both input tables are of the same sizes (Inoue et al., 2014; Zekri et al., 2018). Another advantage of this

algorithm is that it works well with inputs that are compressed with several algorithms such as delta encoding, because of its simplicity (Demaine et al., 2000). If the sizes of the inputs are different, the naive algorithm is less efficient and many techniques have been proposed to improve its performance. For example, algorithms based on binary search (Demaine et al., 2000; Sanders & Transier, 2007) reduce the number of comparisons and memory access by selecting values from the smallest set to locate them in the largest set. Similarly, hash-based techniques (Baeza-Yates & Salinger, 2005; Ding et al., 2008) or techniques using hierarchical data representations (Baeza-Yates & Salinger, 2005) improve performance by reducing the number of comparisons. However, most of these techniques are effective only when the sizes of the two input sets are significantly different (Inoue et al., 2014).

Figure 1. Example of Naive algorithm (Inoue et al., 2014)

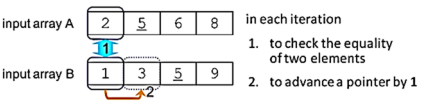


Figure1 presents an example of the naive algorithm. At every time for the two set A and B, naive compares the actual cells. If they are equal, naive marks the intersection else it increments the pointer of the smallest cellule. This process is repeated until one of the lengths is reached. Algorithm 1 gives the pseudo-code of naive.

The worst number of comparisons for the intersection of the two sorted sets of length $size_A$ and $size_B$ is exactly equal to ($size_A$ +$size_B$-1). So its complexity is O($size_A$ + $size_B$). In this case, the sets do not have common values. The best case requires Min ($size_A$, $size_B$) comparisons.

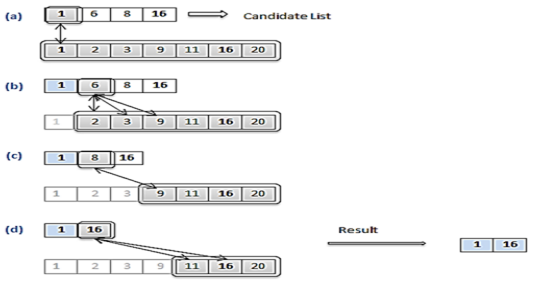
```
Algorithm 1: Scalar_Intersection (int [] A, int [] B, int [] C) {
int cp = 0, i = 0, j = 0;
While (i < size_A && j < size_B) {
if (A[i] == B[j]){
C [cp++] = A[i];
i++; j++;
}
else if (A[i] > B[j])
j++;
else
i++;}
return C;}
```

SvS Algorithm

SvS (Small versus Small) (Demaine et al., 2001; Barbay et al., 2006) is a simple and direct algorithm widely used for the calculation of the intersection between k sets, with |set[0]| ≤|set[1]| ≤... ≤|set[k − 1]|. The lists, which form at the input of SvS, are positioned in ascending order according to their lengths. SvS compares the lists with the shortest sizes, the result will be compared with the following list according to its positioning. The first smaller list is considered as a candidate list. SvS tries to locate its elements in the following list. Each time, it considers an element of the candidate list and performs a binary search to locate it in the other list. If the search is not successful, then it is deleted from the candidate list. During its entire execution, SvS saves the position ℓ of the element being located to continue the search from this position. SvS stops when the candidate list is empty or if it exhausts the k lists. It is important to note that all intersections are always sorted.

Figure 2 presents an example of SvS execution. The small list is considered a candidate list. SvS points to the first cells of the two lists, and tries to locate the content of the actual candidate cell in the longest list. If the intersection is a success then SvS increments the actual cells' counters, like in the step (a) of Figure 2. If the cells do not match then SvS increments the counter of the cell with the smallest content. In step (b) of Figure 2, the incrementing is performed till encountering a cell with content equal or greater to the candidate content. Having reached the cell number 4 with content grater then 6, SvS deletes this last one. The new candidate cell is with a content less than 9, so it is deleted. The new candidate cell is greater than the fourth cell of the longest list; so, SvS will increment the counter of this one and stops with the sixth cell which content is equal to the candidate cell. Algorithm 2 gives the pseudo code of SvS.

Figure 2. Example of the SvS algorithm



Demaine et al., (2000) considered the Swapping SvS variant, where the searched element is chosen from the set containing fewer remaining elements, instead of the first set (initially the smallest) defined in SvS. The problem is that the result list is not necessarily sorted, unlike the result of SvS which always sorted. So, if we would run this variant over several lists, it will be necessary to add a sorting procedure. This addition could increase the time of overall execution.

```
Algorithm 2: SvS_Intersection (set, k)
Sort the sets by size (|set[0]| ≤|set[1]| ≤. . . ≤|set[k-1]|).
Let the smallest set set[0] be the candidate answer set.
for each set S from set do initialize ℓ[S] = 0.
for each set S from set do
for each element e in the candidate answer set
do
binary search for e in S in the range ℓ[S] to |S|, and update ℓ[S]
to the rank of e in S.
if e was not found then
remove e from candidate answer set, and advance e to the next
element in the answer set.
end if
end for
end for
```

Parallel Algorithm

The main objective of Inoue (Inoue et al., 2014) is to reduce the branch mispredictions as they cause difficult jumps. In other word, authors want to reduce the number of costly hard-to-predict conditional branches. The technique is to advance pointer by more than one element at a time. The

key for the authors is to use comparisons by blocks. Technically, (Inoue et al., 2014) Hwang &amp (Hwang & Lin, 1971)has extended the scalar fusion algorithm by comparing several values from each input table. They call the number of elements compared at the same time, the block (S). We refer to this algorithm in the name of its author (Inoue). If the total number of elements in an input table is not a multiple of the size of the S block, we can simply return to the naive approach for the other elements. Inoue et al. (2014) predicted that if the size of the block is equal to 4 then their scalar algorithm presents its best performance. Also, it predicates that if the size of the larger array sizeB is more than twice the size of the smaller array sizeA, S = 3 is the best if the two input arrays have the same size. They used the SIMD instructions with a block of multiples of 4. They assumed that the number of output elements is much smaller than the number of input elements, and the size of the input sets are not significantly different.

Figure 3. Example of the Inoue algorithm sequence (S = 2) (Inoue et al., 2014)

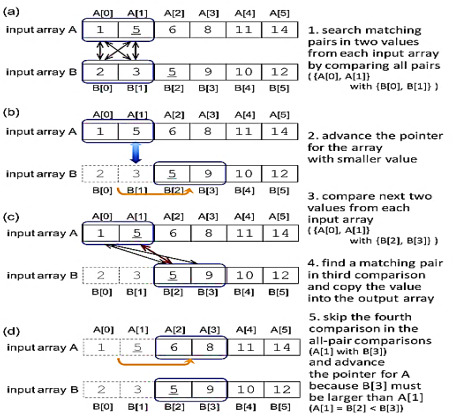


Figure 3 shows an example of how this algorithm works on two sets with S=2. In step (a), Inoue compares all combinations of cells and advances the pointer for the array with the smaller value, as it is shown in instruction (b). The algorithm compares the cells of the precedent block by those new cells. Also, in step (c), Inoue performs all comparisons; and continue to increase the pointer of the block with small values until finding a match. In some cases, some comparisons are avoided as implicitly, some cells of the second array are greater than some cells of the first array. The problem, these cases bust be implemented. So, more time will be spent for avoiding these tests.

```
Algorithm 3: Inoue_Intersection (int [] A, int [] B, int [] C) {
int Cpos = 0;
while (1) {
int Adat0 = A[Apos]; int Adat1 = A[Apos + 1];
int Bdat0 = B[Bpos]; int Bdat1 = B[Bpos + 1];
if (Adat0 == Bdat0) {
C[Cpos++] = Adat0;
}
else if (Adat0 == Bdat1) {
C[Cpos++] = Adat0;
Goto advanceB;
}
else if (Adat1 == Bdat0) {
```

```
C[Cpos++] = Adat1;
goto advanceA;
}
if (Adat1 == Bdat1) {
C[Cpos++] = Adat1;
goto advanceAB;
}
else if (Adat1 > Bdat1) goto advanceB;
else goto advanceA;
advanceA:
Apos+=2;
if (Apos >= Aend) { break; } else { continue; }
advanceB:
Bpos+=2;
if (Bpos >= Bend) { break; } else { continue; }
advanceAB:
Apos+=2; Bpos+=2;
if (Apos >= Aend || Bpos >= Bend) {break;}
return C;
}
```

Contribution

We propose a solution which minimizes the number of comparisons made as much as possible. This minimization leads to a rapid restitution of intersection result. In order to accelerate the intersection computation, related works in section 2 have proposed many preprocessing techniques. The simplest one is to take in the input ordered sets. Our idea is to create a new structure for presenting the input in memory and to exploit it in a concise way (Zekri et al., 2018). In this structure we will divide the initial tables into fragments called sequences. Each sequence is preceded by two fields; the first one is an identifier of the sequence and the second is the number of elements contained in the sequence.

- **SeqID:** is the identifier of the sequence. It is the quotient of the Euclidean division of the current value of the table by the sequencing value.
- **Nseq (the Number of cells):** is the number of elements contained in a sequence. These fields will help to predict the number of elements to jump to the next sequence; hence the gain in the number of comparisons between elements of the two compared tables.

Creating Sequences

In information retrieval, indexes must be well constructed in order to extract information quickly and efficiently information. In our method, the sequences play the role of such index. The sequencing value is obtained in an experimental way and varies according to the distribution of values, in other words, it depends on the minimum and maximum values in the table. Each sequence contains all the values between (SeqID * sequencing value) and ((SeqID +1) * sequencing value).

Figure 4. Example of sequences creation for an array of integers

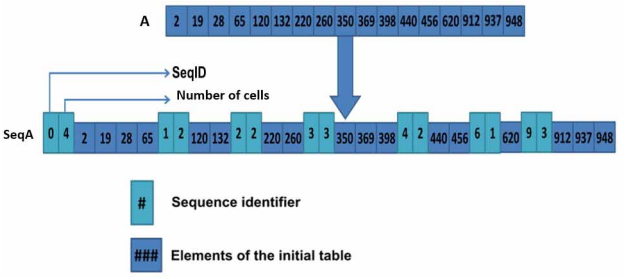
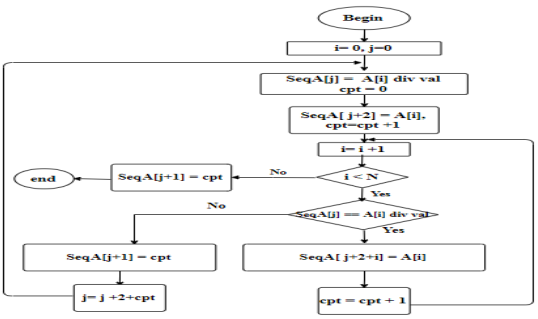


Figure 4 presents an example for a sequencing value equal to 100, the sequences are 0, 100, 200, 300... etc. The first sequence has a SeqID equal to 0 and contains 4 numbers. The value 2 is equal to 0*100+2. The second SeqID is equal to 1 and contains two values. By the same calculus, 132 is equal to 1*100+32. Every sequence is obtained by applying Euclidean division by 100.

Figure 5. Flowchart of preprocessing

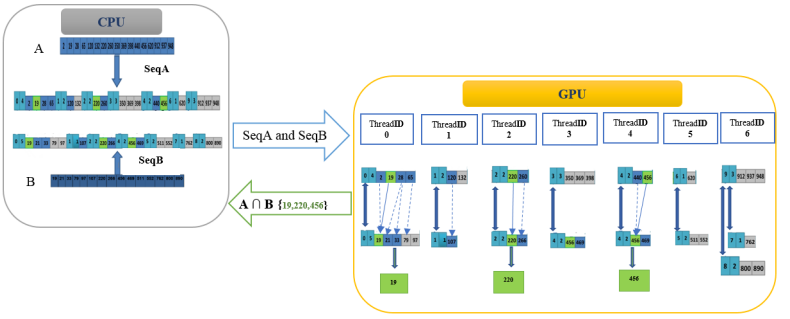


The flowchart of the Figure 5 presents the processing for producing the set SeqA which is deduced from the sorted input A. The cells SeqA[j] and SeqA[j+1] are SeqID and Nseq respectively. At every time, SeqA[j] receives the Euclidian division of A[i] by the sequencing value Val. In the experimentation section, we will give the best values of Val which can produce the best intersection computing times.

GTWJ Algorithm

The GTWJ (for GPU Test With Jump) algorithm takes as input two sorted lists A and B, and transforms them into two sequences SeqA and SeqB with a well-specified sequencing value and returns the common elements between them. The returned elements of the intersection set are themselves sorted. So, it is easily to search intersection between more than two sorted lists unlike (Baeza-Yates, 2004). The GTWJ algorithm processes sequences as elements of a sorted table and performs a search like the Naive algorithm between sequences in each table and between the elements of sequences with the same SeqID. We implemented the algorithm on a GPU card using CUDA C. The lists A and B are initially read by the CPU, transformed into sequences and then transferred to the GPU's global memory. We have assigned for each sequence a thread that will compare the SeqIDs and look for common elements between the two sequences with the same SeqID (see Figure 6, Figure 7).

Figure 6. Example of GTWJ (pre-processing and calculation on GPUs)



Each time, two SeqIDs are equal, the corresponding thread searches for the intersection between these two sequences. If the SeqID value of the first table is elevate than the SeqID of the second table, then the thread passes to the next sequence in the second table and compares it with the SeqID of current sequence, otherwise it ends. The algorithm ends when there are no more sequences in one of the two tables (see Figure 7).

Figure 7. Flowchart of the GTWJ algorithm

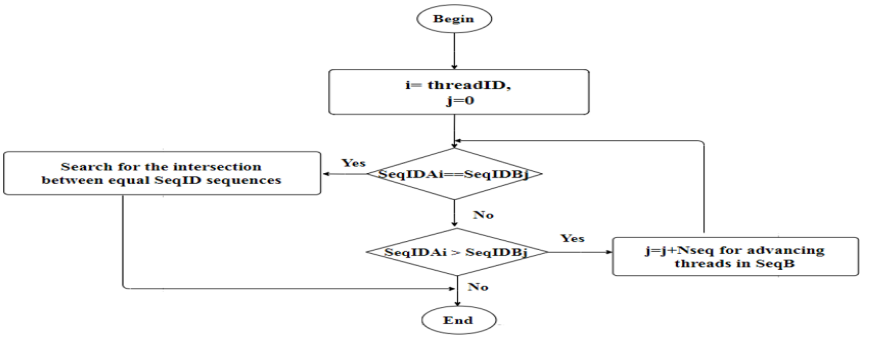
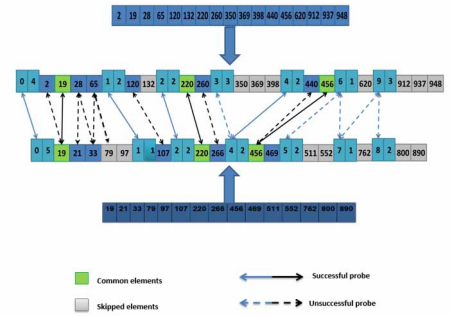


Figure 8 shows an example of how GTWJ works. It will perform 11 comparisons between the elements and 9 comparisons between the SeqID (20 in all) and 15 avoided elements. Naive will perform 26 comparisons for the same inputs. The advantage of the GTWJ algorithm is that it has the ability to avoid sequences, such as sections: 3, 5, 6, 7, 8 and 9 and cases, as in both sections 0 and 1.

Figure 8. Example of the GTWJ algorithm processing



The GTWJ algorithm is presented by the following pseudo code for two sets. The result is recorded in the variable C. We recall that C is always sorted.

```
Algorithm 4: GTWJ_Intersection (int [] SeqA, int [] SeqB, int []
C) {
int i = threadIdx.x + blockDim.x * blockIdx.x;
int j = 0; int cp = 0;
while(j < size_SeqB){
if(SeqIDA_i == SeqIDB_j)
{
while (i < NSeqA_i && j < NSeqB_j)
{
if (SeqA[i] == SeqB[j]) {
C[cp++] = SeqA[i];
i++; j++;}
else if(SeqA[i] < SeqB[j]) {i++;}
else j++;
} //end while
else if (SeqIDA_i > SeqIDB_j) {j = j+ NSeqB_j;}
else break;
} //end while
return C;}
```

Naive executes ($size_A + size_B$) operations. Due to our tables composition, GTWJ would execute ($size_A + size_B$ - NS) where NS is the number of avoided cells. The experiments presented in the next section will show the gains in terms of time execution and avoided cells.

Experimental EVALUATION

The experiments were conducted on a machine equipped with an Intel® processor Core™ i7-5500U CPU @ 2.40 GHz, 8GB of RAM and a 2GB Nvidia GeForce 840M card. We use Visual Studio 2013 software under Windows, Our CUDA code is compiled with the NVIDIA Compiler using CUDA version 8.0. We have implemented the GTWJ, Naive, SvS and Inoue algorithms on sorted sets generated in the same way as described by Baeza-Yates in (Baeza-Yates, 2004). We generate series of random integers uniformly distributed. We must note here, that the effectiveness of any solution is related with the correctness of the result. All the papers presented have used synthetic datasets like (Baeza-Yates, 2004) and natural numbers like (Lemire et al., 2016).

Like all works of this research field, we have compared the different algorithms on the time execution which is the major and important metric. We have also compared them on the number of

executed tests between cells as the time execution is directly related to this parameter. We will also show that our index allows us to jump useless parts.

The important works (Chen et al., 2013) and (Li et al., 2021) have used real datasets for proposing new solutions; authors have also proposed interesting frameworks for conducting statistical tests, in order to show efficient comparisons. In our future works, we will use these approaches for better comparisons; but the object of this research is to stay in the same logic of the works of this research field.

We recall that the input is always sorted, so the preprocessing time itself is not considered as it is presented in all papers of the related works, see for example (Ding et al., 2008).

Variation of the Sequence's Size

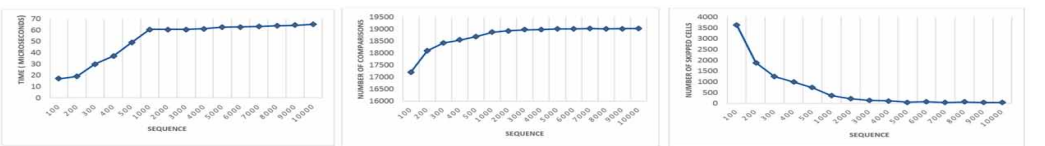
In this section, we have varied the lengths of sequences in order to see their impacts on the intersection computing time. Also, we tried to find the sequence which gives the best computing time. These experiments concern GTWJ only. In the first time, we searched this sequence in the case of tables with equal sizes. In the second one, we tried to find it in the case where tables are with different lengths. In all experiments, we have considered the time execution, the number of tests and the number of skipped cells as metrics of comparison.

Tables with Equal Sizes

In these experiments, we have taken 50000 as the size of these two tables. We have varied the sequence value from 100 to 10000. We have collected information about the three considered metrics. The best sequence will minimize the time execution and number of tests and maximize the number of skipped cells. The results of this experiment are presented in the Figure 9. Figure 9(a) shows that the more the sequencing value is higher, the more the execution time increases. This figure shows that the best sequencing values vary between 100 and 1000. Although 100 is the best choice. Figure 9(b) justifies the result obtained in the previous figure; the appearance of the two graphs is similar. We then notice that when the sequences are too long, the comparison numbers increase. This figure confirms that 100 is the best sequencing value.

At the base, we had the idea that if we could jump some parts in the tables, we will get a rapid intersection computation. This is our main idea. The results illustrated in Figure 9(c) are exactly what we want to have. This Figure 9(c) shows that the number of skipped cells is inversely proportional to the size of the sequence. The more the sequence increases, the fewer cells are skipped. This finding is logic as when the sequence is long it means that we get closer to the naive method where all the elements are scanned. It is like comparing tables without any preprocessing as the number of sequences becomes small. Like Figure 9(b), Figure 9(c) confirms that for a sequence equal to 100 gives the highest number of skipped cells.

Figure 9. (a) Execution time by varying the sequence for tables with equal sizes, (b) Number of tests by varying the sequence for tables with equal sizes, (c) Number of skipped cells by varying the sequence for tables with equal sizes



(a) (b) (c)

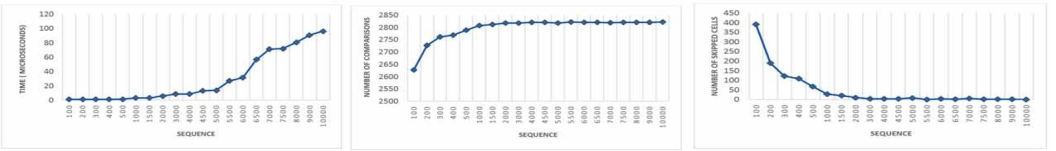
Tables with Different Sizes

In this case, we have taken 1000 as the size of the first table and 100000 as the size of the second one. We tried to locate the smaller table in the biggest one as explained above. By conducting the same experiment, we have found similar results as shown in Figure 10. GTWJ gives the best results when the sequences are small even for tables with different sizes. Figure 10(a) shows that the interval [100, 500] gives the best results.

Figure 10(b) confirms the same result obtained in Figure 9(b). In these experiments, the number of comparisons grows when the size grows. This is logical as when the size grows the situation is like when we apply naive comparisons by scanning all tables. This is the contrary of what we want to do. Figure 10(b) confirms that, for small sequence, more parts of the entries are jumped and for a size equal to 100, the smallest number of comparisons is obtained.

Figure 10(c) shows that indeed for small sequences, GTWJ would perform more jumps; this result is logical as when sequences are long more cells will be compared but for smallest sequence more parts of the compared tables will be jumped. The idea behind GTWJ is to divide for more jumped parts.

Figure 10. (a) Execution times by varying the sequence for tables with different sizes, (b) Number of comparisons by varying the sequence for tables with different sizes, (c) Cells jumped by varying the sequence for tables with different sizes



(a) (b) (c)

In the following part, we will present the comparing results between Naive, Inoue, SvS and GTWJ algorithms. For this latter, we have executed it with four versions GTWJ100, GTWJ1000, GTWJ5000 and GTWJ10000 according to the sizes of sequences. The objective is to conðrm the results of the precedent experiments.

Comparison of Algorithms on Tables with Equal Sizes

In these series of comparisons, we have varied the sizes of the input tables from 1000 to 50000. The results of these experiments are shown on Figure 11(a) and, as the gap between the time executions of the algorithms, we have zoomed the results on Figure 11(b). This zoom concerns the results of GTWJ only.

Figure 11(a) shows that SvS has consumed more time than the rest of algorithms. Its problem resides in the logic it has adopted for computing the intersection. SvS proposes the use of a repetitive binary search for locating the elements. The process of filling and unstacking from the stack takes a lot of time. This technique is not good for tables with equal sizes. SvS does not apply any preprocessing on the entries so it will scan all entries. This figure shows also that Naive is better than SvS in terms of time execution as it makes a directed growing scan unlike the case of using binary search. We can see also that Inoue has made a better performance as it was initially adapted for GPU. We have chosen the better configuration for Inoue as we have taken bloc sizes s=4 (Inoue et al., 2014). The excess time is due to the supplementary instructions as it performs all possible comparisons in each block. Figure 11(a) shows also that GTWJ, executed with different sequence sizes, has presented better times.

Figure 11(b) gives the different execution times obtained by the different GTWJ variants. This figure confirms the results obtained on Figure 9(a), Figure 9(b) and Figure 9(c). We can see that GTWJ10000 has consumed a long time. This is clear as when sequences are too long, the computing

seems like the naive algorithm. This remark is also correct for GTWJ1000 which has been faster than GTWJ5000 as a confimation of precedent results. Finally, GTWJ100 was the fastest among all algorithms. For sequence size equal to 100, the computing of the intersection as the threads can manipulate consecutively a lot of small parts which can be processed quickly.

Figure 11. (a) Execution time for tables with equal sizes, (b) Zoom on Figure 11(a)

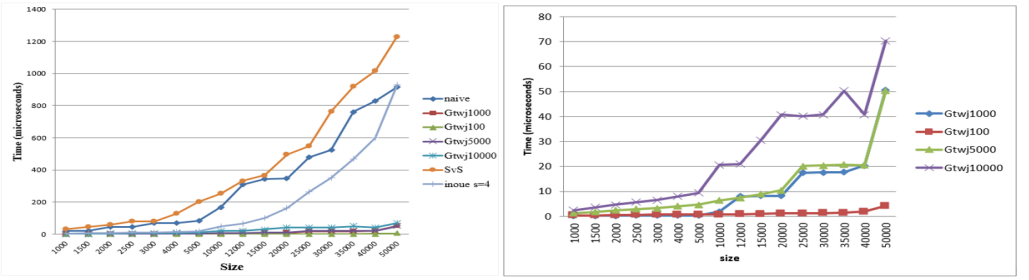
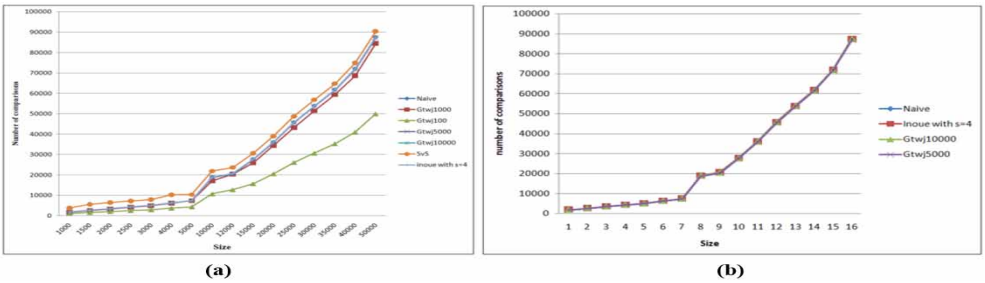


Figure 12(a) depicts the number of comparisons performed in the executions of these seven algorithms. Figure 12(a) shows only the curves of SvS, Naive, GTWJ1000 and GTWJ100 algorithms as the results of the algorithms have presented nearly the same results, which was surprising as shown on Figure 12(b).

On Figure 12(a), we can see that SvS makes more comparisons than the rest of algorithms. The use of the binary search increases this number, as this method will test the cells many times for comparing it with the actual cell of the first table. This figure shows GTWJ1000 is better than SvS in terms of number of comparisons. We can also see that for sizes greater than 10000, GTWJ1000 has performed a smaller number of comparisons than GTWJ10000, GTWJ5000 and Inoue.

Figure 12. (a) Number of comparisons for tables with equal size, (b) The hidden results of Figure 12(a)

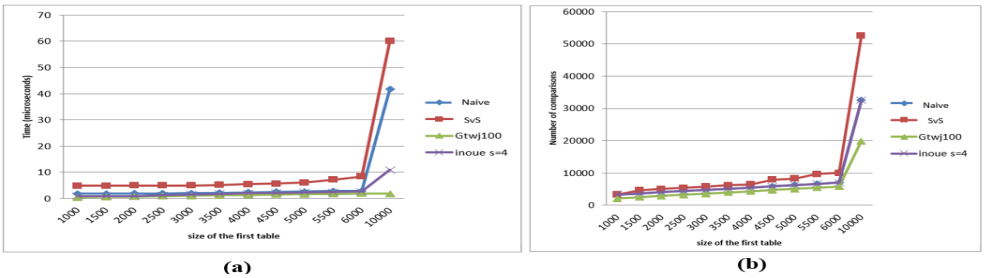


In this same context of comparison, Figure 12(a) confirms that for size equal to 100, GTWJ has made a smaller number of comparisons comparatively to other algorithms. This small number is due to the decomposition as when the sequence size is small. We will have more and more sequences which can increase the probability to have more different sequences, which can help our algorithms to jump these parts. The tables' constructions explained above helps a lot for jumping more parts (sequences and cells).

Comparison According to Tables with Different Sizes

In this experiment, we set the size of the second table at 50,000 and we varied the size of the ðrst table from 1000 to 10000. We also set the GTWJ sequencing value to 100. The results of these experiments are illustrated in Figure 13(a).

Figure 13. (a) Execution time by varying the size of the first table, (b) Number of comparisons by varying the size of the first table



This figure shows that SvS has consumed more time compared to the other algorithms. The Naive algorithm also consumes a high amount of time compared to Inoue. These three algorithms test all tables' elements. SvS is always penalized by the implementation of the binary search. Like Figure 11(a), Inoue is in the third place in terms of execution time consumption. As we can see, GTWJ100 was faster than the others because it manages to skip many sequences from which it performs fewer tests according to what we have already advanced in this part of experiments.

Figure 13(b) shows that SvS, Naive and Inoue algorithms have touched all the cases of both tables, which increase their execution times. This figure shows that GTWJ100 has made less number of comparisons; this is explained by the preprocessing of the entries as it can jump cells and parts according to Nseq, the number of cells contained in a sequence, and SeqID as GTWJ can avoid testing completely sequences if their SeqId's do not match. These results join with the experiments shown in this section. This reduced number of comparisons results is induced by the reduced number of execution times as shown in Figure 13(a).

Comparison Between TWJ50 and TWJ100

As we have searched for the best sequence size in the first experiments (see the figures 9 and 11), we have tried to look if for sequence's size smaller than 100, we could have better performances for GTWJ. In this context, we have executed GTWJ with the sizes 100 and 50 and we have collected the execution times. Figure 14 shows the performances of these to variants while Figure 15 gives the gap in time between GTWJ50 and GTWJ100.

Figure 14. Comparison between GTWJ50 and GTWJ100

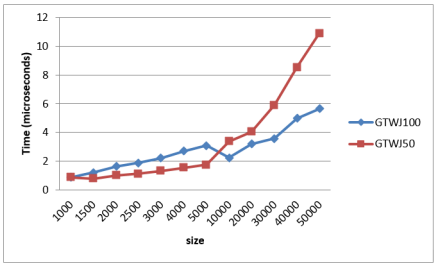
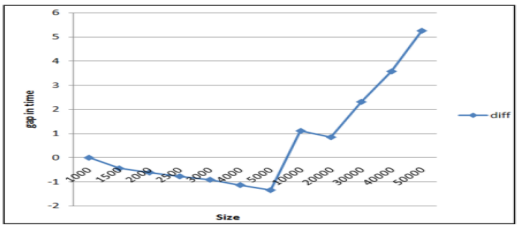


Figure 14 shows that GTWJ50 was faster than GTWJ100 for tables with sizes smaller than 5000 while for higher sizes, GTWJ100 has presented better times than GTWJ50. But the gap in time of Figure 15 was not so large between GTWJ50 and GTWJ100 as the highest gap has reached only 1.34 microseconds, but for large sizes, the gap between GTWJ100 and GTWJ50 was too high. We can conclude here, that for 100 as sequences size is so interesting and is sufficient to having good results.

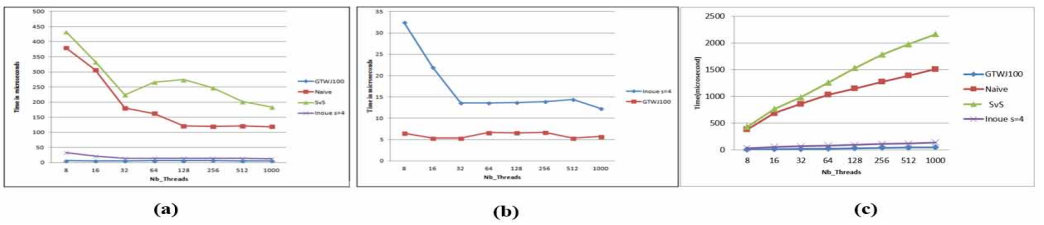
Figure 15. Comparison between GTWJ50 and GTWJ100



Execution Time According to the Number of Threads for Tables of Equal Sizes

In the experiments which concern the variation of the number of threads for looking at its impact on executions' times, we have fixed the entries' sizes to 50000. The results are presented on Figure 16.

Figure 16. (a) Execution time by varying the number of threads for tables of equal sizes, (b) Execution times of GTWJ100 and Inoue for equal sizes, (c) Accumulated execution times by varying the number of threads for tables with equal sizes



We can see on Figure 16(a) that the increase in the number of threads has made Naive and SvS faster; but Naive has given better performances than SvS. It appears that SvS is not suitable for such cards; we think the execution of SvS on CPU is well for it. This figure shows that Naive and SvS have consumed more times than Inoue and GTWJ100.

Figure 16(b), which is a zoom on Figure 16(a), gives the executions' times of Inoue and GTWJ100. It is clear that Inoue has been faster when the number of threads has been increased. Its better time was reached for 1000 threads. We can also see that globally the number of Threads has not a high impact on the GTWJ100; its execution times were not high. GTWJ100 has presented best performances for 26, 32 and 512 threads. For us, GTWJ100 is more impacted by the tables' construction than by the increase in the number of threads, as for a high number of sequences, it will be possible to jump more and more parts in the entries.

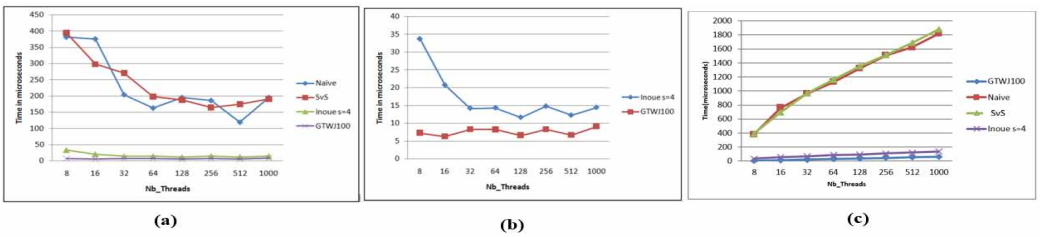
Figure 16(c) illustrates the accumulated execution times of these four algorithms. This figure shows that SvS has not presented good performances even though the number of threads has increased but Naive can give better performances when the number of threads increases. This figure shows that Inoue is better than precedent algorithms as it was initially presented to be executed on GPU

cards. Our proposal has presented better performances and has been constant in its execution time. The preprocessing is its key of success as it is possible to avoid testing larger parts of the entries.

Execution Time According to the Number of Threads for Tables of Different Sizes

As the GPU offers a high parallelism but defining a high number of threads, a high-performance computing method will explore efficiently these threads in order to speed up calculations. In this context, we have varied the number of threads in order to see if there is a configuration which can lead to quick computing.

Figure 17. (a) Execution time by varying the number of threads for tables with different sizes, (b) Execution times performed by GTWJ100 and Inoue, (c) Accumulated execution times by varying the number of threads for tables with different sizes



In this objective, we have conducted intensive experiments on GTWJ100, Naive, Inoue and SvS algorithms as shown on Figure 17(a) and Figure 17(b). For tables with different sizes, we have varied the number of threads Nb_threads per bloc and fixed the size of the first table to 10000 and 50000 as the size of the second one. We tried to locate the elements of the first table into the second one. The experiments results are presented on Figure 17(a). We can see that the increase in the number of threads has accelerated the execution of Naive and SvS, but their execution times have remained high comparatively to Inoue and GTWJ100. The best execution time has been obtained for 512 threads per bloc while for 256 threads per bloc SvS has presented its best performance.

Figure 17(b) gives the results of GTWJ100 and Inoue. We can see that for all variations GTWJ100 has been faster than Inoue. More of this the best number of threads that helps GTWJ100 to be faster is 16 threads per bloc, where GTWj100's execution time has reached 6.33 microseconds. Globally for 16, 128 and 512 threads per bloc, GTWJ100 was very fast.

Figure 17(c) gives the accumulated execution times obtained in precedent figures. This figure shows that Naive and SvS were slow in terms of execution times. It is important to know that these two algorithms were not proposed in their first propositions to be implemented on such support of parallelism. This figure demonstrates that GTWJ100 was the fastest algorithm even if Inoue has also been fast. This one was proposed to be executed on GPU cards and for comparison bloc S=4 it has presented its best performances. We must report the high gap executions' times of GTWJ100 and Inoue, in one part, and Naive and SvS executions' times, in another part, despite if Figure 17(c) shows that their execution times are near. A zoom on this figure will show this high gap.

Conclusion

The processing of data in search engines passes by the construction of indexes. These ones are obtained by computing intersections by the crawled documents while the processing of queries passes by computing intersections between them and the indexes. The problem of intersection computation of sorted sets as attracted high intention. In this paper, we have presented a new solution for computing it on GPU cards as theses ones offer high parallelism.

For computing the intersection by two sorted lists, we have proposed to reconstruct them into sequences and to count the number of cells into every sequence. These two parameters are sufficient to avoid unnecessary testing of cells. We have implemented our algorithm on graphics processor under the CUDA architecture. We have also presented a large of the state of the art. In order to show the efficiency of our proposal, we have compared it with three other algorithms. The experiments are based on synthetic data. We have taken the response times and the number of comparisons as comparison metrics. These experiments showed that our solution performed better than other algorithms. Currently, we are extending it for the calculation of inverted indexes.

FUNDING AGENCY

Publisher has waived the Open Access publishing fee

REFERENCES

Alon, I., & Michael, R. (1978). Finding a minimum circuit in a graph. *SIAM Journal on Computing*, *7*(4), 413423.

Ao, N., Zhang, F., Wu, D., Stones, D.S., Wang, G., Liu, X., Liu, J., & Lin, S. (2011). Efficient parallel lists intersection and index compression algorithms using graphics processing units. *VLDB, 4*(8).

Baeza-Yates, R. (2004). A Fast Set Intersection Algorithm for Sorted Sequences. *Proceedings of the Annual Symposium on Combinatorial Pattern Matching*, 400–408. doi:10.1007/978-3-540-27801-6_30

Baeza-Yates, R., & Salinger, A. (2005). Experimental Analysis of a Fast Intersection Algorithm for Sorted Sequences. *Proceedings of SPIRE*, 3772, 13–24. doi:10.1007/11575832_2

Baeza-Yates, R., & Salinger, A. (2010). Fast Intersection Algorithms for Sorted Sequences. Algorithms and Applications. Ukkonen Festschrift, LNCS 6060, 45–61. doi:10.1007/978-3-642-12476-1_3

Barbay, J., & Kenyon, C. (2002). Adaptive intersection and t-threshold problems. *Proceedings of the 13th ACM-SIAM Symposiumon Discrete Algorithms (SODA)*, 390–399.

Barbay, J., Lopez-Ortiz, A., & Lu, T. (2006). Faster adaptive set intersections for text searching. *5th Int Workshop on Experimental Algorithms (WEA)*, 4007, 146–157. doi:10.1007/11764298_13

Bentley, J. L., & Yao, A. C. (1976). An almost optimal algorithm for unbounded searching. *Information Processing Letters*, *5*(3), 82–87. doi:10.1016/0020-0190(76)90071-5

Bill, P., Pagh, A., & Pagh, R. (2007). Fast evaluation of union-intersection expressions. *Proceedings of the International Conference on Algorithms and Computation*, 739–750. doi:10.1007/978-3-540-77120-3_64

Brown, M. R., & Tarjan, R. E. (1979). A fast merging algorithm. *Journal of the Association for Computing Machinery*, *26*(2), 211–226. doi:10.1145/322123.322127

Carter, L., Floyd, R., Gill, J., Markowsky, G., & Wegman, M. (1978). Exact and approximate membership testers. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing (STOC '78)*. ACM Press.

Chen, Y. H., Hong, W. C., Shen, W., & Huang, N. N. (2013). Electric Load Forecasting Based on a Least Squares Support Vector Machine with Fuzzy Time Series and Global Harmony Search Algorithm. *Energies*, *6*(4), 1887–1901.

Demaine, E., Lopez-Ortiz, A., & Munro, I. (2000). Adaptive set intersections, unions, and differences. *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 743-752.

Demaine, E., Lopez-Ortiz, A., & Munro, J. I. (2001). Experiments on adaptive set intersections for text retrieval systems. In A. L. Buchsbaum & J. Snoeyink (Eds.), *ALENEX 2001. LNCS* (Vol. 2153, pp. 91–104). Springer. doi:10.1007/3-540-44808-X_7

Ding, S., He, J., Yan, H., & Suel, T. (2008). *Using graphics processors for high-performance ir query processing*. ACM. doi:10.1145/1367497.1367732

Fort, M., Sellarès, J. A., & Valladares, N. (2017). Intersecting two families of sets on the GPU. *Journal of Parallel and Distributed Computing*, *104*, 167–178. doi:10.1016/j.jpdc.2017.01.026

Hwang, F. K., & Lin, S. (1971). Optimal merging of 2 elements with n elements. *Acta Informatica*, *1*(2), 145–158. doi:10.1007/BF00289521

Hwang, F. K., & Lin, S. (1972). A simple algorithm for merging two disjoints linearly-ordered sets. *SIAM Journal on Computing*, *1*(1), 31–39. doi:10.1137/0201004

Inoue, H., Ohara, M., & Taura, K. (2014). Faster Set Intersection with SIMD instructions by Reducing Branch Mispredictions. *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, *8*(3), 293–304. doi:10.14778/2735508.2735518

Latapy, M. (2008). Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical Computer Science*, *407*(1-3), 458–473. doi:10.1016/j.tcs.2008.07.017

Lemire, D., Boytsov, L., & Kurz, N. (2016). SIMD compression and the intersection of sorted integers. *Software—Practice, 46*(6), 723-749.

Li, M. W., Wang, Y. T., Geng, J., & Hong, W. C. (2021). Chaos cloud quantum bat hybrid optimization algorithm. *Nonlinear Dynamics*, *103*(1), 1167–1193. doi:10.1007/s11071-020-06111-6

Pugh, W. (1990a). *A skip list cookbook*. UMIACS-TR-89-72.1.

Pugh, W. (1990b). Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, *33*(6), 668676. doi:10.1145/78973.78977

Sanders, P., & Transier, F. (2007). Intersection in Integer Inverted Indices. *Proceedings of the Workshop on Algorithm Engineering and Experiments*, 71–83.

Schank, T., & Wagner, D. (2005). Approximating clustering coefficient and transitivity. *Journal of Graph Algorithms and Applications*, *9*(2), 265–275. doi:10.7155/jgaa.00108

Schlegel, B., Willhalm, T., & Lehner, W. (2011). *Fast Sorted-Set Intersection using SIMD Instructions*. WS on Accelerating Data Management Systems Using Modern Processor and Storage Architectures.

Tatikonda, S., Junqueira, F., Cambazoglu, B., & Plachouras, V. (2009). On efðcient posting list intersection with multi-core processors. *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, 738-739.

Tsirogiannis, D., Guha, S., & Koudas, N. (2009). Improving the performance of list intersection. *VLDB*, *9*(August), 2428.

Vladimir, B. (2013). Experimental Comparison of Set Intersection Algorithms for Inverted Indexing. *ITAT Proceedings, CEUR Workshop Proceedings*, 1003, 58–64.

Zekri, L., Manseur, F., & Belhadj, O. (2018). *Calcul de l'intersection entre listes triées à base de sauts (computing the intersection between sorted lists by jumps). EDA 2018*. Tanger.

Zhang, J., Lu, Y., & Daniele, G. (2020). FESIA. A Fast and SIMD-Efficient Set Intersection Approach on Modern CPUs. *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 1465-1476.

Zhou, J., Guo, Q., Jagadish, H. V., Luan, W., & Tung, A. K. H. (2016). Generic Inverted Index on the GPU. Academic Press.

Zukowski, M., Heman, S., Nes, N., & Boncz, P. (2006). Super-scalar ram-cpu cache compression. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE '06)*. IEEE Computer Society. doi:10.1109/ICDE.2006.150

Manseur Faiza is a PhD student at the department of computer sciences at Oran1 Ahmed Ben Bella University. She obtained her master in the field of information retrieval (large-scale system). Currently, she is interested in indexation, big data and parallel computing.

Zekri Lougmiri is assistant professor at the department of computer sciences at Oran1 Ahmed Ben Bella University. He received his PhD at the same university. He works on multi-criteria optimization, information retrieval and big data.

Senouci Mohamed is professor at the department of computer sciences at Oran1 Ahmed Ben Bella University. His current research area includes pattern recognition, deep learning and ad hoc networks.