gSSJoin: a GPU-based Set Similarity Join Algorithm

Sidney R. Junior[1], **Rafael D. Quirino**[1],
Leonardo Andrade Ribeiro[1], **Wellington S. Martins** [1]

[1]Instituto de Informatica – Universidade Federal de Goias (UFG)
Alameda Palmeiras, Quadra D, Campus Samambaia
CEP 74001-970 - Goiania - GO - Brazil

{sidney,rafael,laribeiro,wellington}@inf.ufg.br

Abstract. *Set similarity join is a core operation for text data integration, cleaning, and mining. Previous research work on improving the performance of set similarity joins mostly focused on sequential, CPU-based algorithms. Main optimizations of such algorithms exploit high threshold values and the underlying data characteristics to derive efficient filters. In this paper, we investigate strategies to accelerate set similarity join using Graphic Processing Units (GPUs). Our approach exploits massive parallelism instead of filtering and, as a result, exhibits much better robustness to variations of threshold values and data distributions. Experimental evaluation shows that we are able to obtain up to 57x speedups over highly optimized CPU-based algorithms.*

1. Introduction

The problem of efficiently answering set similarity join queries has attracted growing attention over the years [Sarawagi and Kirpal , Chaudhuri et al. 2006, Bayardo et al. 2007, Vernica et al. 2010, Xiao et al. 2011, Ribeiro and Härder 2011, Wang et al. 2012, Cruz et al. 2015]. Set similarity join returns all pairs of similar sets from a dataset — two sets are considered similar if the value returned by a set similarity function for them is not less than a given threshold. This operation is of great interest and practical importance both in itself, and as a basic operator for more advanced data processing tasks, including integration, cleaning, and mining of data [Leskovec et al. 2014].

Set similarity join is a popular approach to dealing with text data, whose representation is typically sparse and high-dimensional. Text data can be easily mapped to sets and there is a rich variety of set similarity functions available to capture various notions of similarity. Moreover, predicates involving such functions can be equivalently expressed as a set overlap constraint. As a result, set similarity join is reduced to the problem of identifying set pairs with enough overlap.

The set overlap abstraction provides the basis to several optimizations. *Prefix filtering* [Sarawagi and Kirpal , Chaudhuri et al. 2006] is arguably the most important of such optimizations. In fact, this technique is employed by all state-of-the-art algorithms considered in a recent experimental evaluation [Mann et al. 2016]. Prefix filtering exploits the threshold value to prune dissimilar set pairs by inspecting only a fraction of them. A filter-and-verify framework is typically used in this context, where input sets are *sequentially* processed: with the support of an inverted index, prefix filtering is employed to discard set pairs that cannot meet the overlap constraint; then, the surviving pairs are evaluated and those verified as similar are sent to the output.

Unfortunately, prefix filtering is only effective at high threshold values. Its pruning power drops drastically at lower thresholds, leading to an explosion of the number of set pairs that need to be compared in the verification step. Length-based filtering [Sarawagi and Kirpal], another popular optimization technique, also suffers from the very same limitation. This serious drawback may render set similarity join unsuitable to applications which often require lower thresholds to produce accurate results — important examples are duplicate detection and clustering [Hassanzadeh et al. 2009]. Moreover, filtering effectiveness and, in turn, algorithm performance heavily rely on characteristics of the underlying data distribution [Sidney et al. 2015]. In particular, the filtering effect is severely reduced (or even eliminated) on more uniform data distributions.

One alternative way to tackle the problem of efficiently answering set similarity join queries is to exploit parallelism. Today virtually all processors support parallelism through the use of multiple cores. Multi-core processing is a growing industry trend and it has been followed by the so-called many-core architectures like GPUs (Graphic Processing Units). Many-core processors, also known as accelerators, have a large number of processing units — hundreds or thousands — but in the form of slower and simpler cores. Recent developments and affordability of GPUs have made them attractive to scientists in different areas. GPUs are designed for massive multi-threaded parallelism and are inherently energy efficient, because they are optimized for throughput and performance per Watt. However, GPUs have a different architecture and memory organization and to fully exploit their capabilities it is necessary considerable parallelism (tens of thousands of threads) and an adequate use of its hardware resources. This imposes some constraints in terms of designing appropriate algorithms, requiring the design of novel solutions and new implementation approaches.

In this paper, we present a parallel algorithm and a GPU-based implementation for the set similarity join problem. Our solution takes advantage of data parallelism by processing individual tokens of a given set in parallel. This operation can be seen as a set similarity search since it finds all sets in a set collection that are similar to a given input set. When performing a self-join, this operation is applied repeatedly, in batch, so that all sets of the collection are compared against all. This greatly improves the similarity join processing and can be easily mapped to modern highly-threaded accelerators like many-core GPUs. The proposed solution, called gSSJoin (GPU-based Set Similarity Join), efficiently implements an inverted index, by using a parallel counting operation followed by a parallel prefix-sum calculation. At search time, this inverted index is used to quickly find sets sharing tokens with the input set. Furthermore, we construct an index for the input set which is used for a load balancing strategy to evenly distribute the similarity calculations among the GPU's threads. Finally, the most similar (above a given threshold) sets are returned to the CPU. In addition to exploiting intra-set parallelism, the solution also deals with inter-set parallelism, which allows the use of modern multi-GPU systems. Our main contribuitions are:

- A fine-grained parallel algorithm for both indexing the data and performing the set similarity joins.
- A highly threaded GPU implementation that takes advantage of intensive occupation, hierarchical memory, and coalesced memory access.
- A scalable multi-GPU implementation that exploits both data parallelism and task parallelism.

- Extensive experimental work with standard real-world textual datasets.

This paper is organized as follows. In Section 2, we introduce the set similarity join problem. Section 3 presents an overview of the architecture and programming models of a GPU. Section 4 describes our solution. Section 5 presents the experimental evaluation. Section 6 covers related work, while Section 7 concludes the paper.

2. Background

2.1. Basic Concepts and Problem Definition

Strings can be mapped to *sets of tokens* in several ways. A well-known method is based on the concept of *q-grams*, i.e., sub-strings of length *q* obtained by "sliding" a window over the characters of the input string. For example, the string *"gSSJoin"* can be mapped to the set of *3*-grams tokens {*'gSS'*, *'SSJ'*, *'SJo'*, *'Joi'*, *'oin'*}.

Given two sets r and s, a set similarity function $sim\,(r, s)$ returns a value in $[0, 1]$ to represent their similarity; a greater value indicates that r and s have higher similarity. We formally define the set similarity join operation as follows.

Definition 1 (Set Similarity Join)**.** Given two set collections $\mathcal{R}$ and $\mathcal{S}$, a set similarity function sim, and a threshold τ, the *set similarity join* between $\mathcal{R}$ and $\mathcal{S}$ returns all scored set pairs $\langle (r, s), \tau\prime \rangle$ s.t. $(r, s) \in \mathcal{R} \times \mathcal{S}$ and $sim\,(r, s) = \tau\prime \geq \tau$.

A popular set similarity function is the well-known *Jaccard similarity*. Given two sets r and s, the Jaccard similarity is defined as $J\,(r, s) = \frac{|r \cap s|}{|r \cup s|}$. A predicate involving the Jaccard similarity and a threshold τ can be equivalently rewritten into a set overlap constraint: $J\,(r, s) \geq \tau \iff |r \cap s| \geq \frac{\tau}{1+\tau}(|r| + |s|)$. In this paper, we focus on the Jaccard similarity, but all concepts and techniques presented henceforth holds for other set similarity functions as well such as Dice and Cosine [Ribeiro and Härder 2011].

Example 1. Consider the sets $r = \{\mathbf{A}, \mathbf{B}, C, \mathbf{D}, \mathbf{E}\}$ and $s = \{\mathbf{A}, \mathbf{B}, \mathbf{D}, \mathbf{E}, F\}$. Thus, we have $|r| = |s| = 5$ and $|r \cap s| = 4$; thus $sim\,(r, s) = \frac{4}{6} \approx 0.66$. For a threshold $\tau = 0.6$, the set overlap constraint is $|r \cap s| \geq \frac{0.6}{1+0.6}(5 + 5) = 3.75$.

Further, we can use the *prefix filtering principle* [Sarawagi and Kirpal , Chaudhuri et al. 2006] to prune dissimilar sets by examining only a subset of them. We first assume that the tokens of all sets are sorted according to a total order. A prefix $r_p \subseteq r$ is the subset of r containing its first p tokens. Given two sets r and s, if $|r \cap s| \geq \alpha$, then $r_{(|r|-\alpha+1)} \cap s_{(|s|-\alpha+1)} \neq \varnothing$. For a threshold τ, we can identify all candidate matches of a given set r using a prefix of length $|r| - \lceil |r| \cdot \tau \rceil + 1$; we denote such prefix by $pref\,(r)$.

The original overlap constraint only needs to be verified on set pairs sharing a prefix token. Finally, we pick the token frequency ordering as total order, thereby sorting the sets by increasing token frequencies in the set collection. Thus, we move lower frequency tokens to prefix positions to minimize the number of verifications.

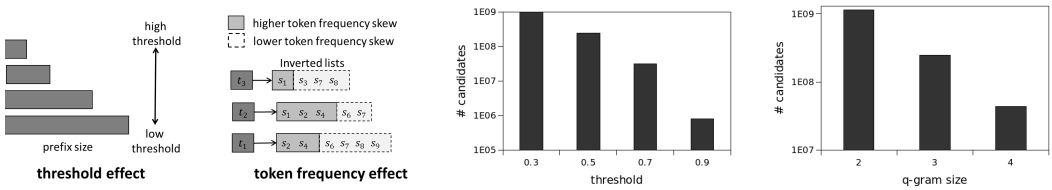
2.2. General Algorithm

Most current set similarity join algorithms for main memory employ an inverted index and follow a filter-and-refine framework [Mann et al. 2016]. A high-level description of this framework presented by Algorithm 1. An inverted list I_t stores all sets containing a token t in their prefix. The input collection C is scanned and, for each set r, its prefix

Algorithm 1: General set similarity join algorithm.

Input: A sorted set collection $\mathcal{C}$, a threshold τ
Output: A set S containing all pairs (r, s) s.t. $sim\,(r, s) \geq \tau$

1 $I_1, I_2, \ldots I_{|\mathcal{U}|} \leftarrow \varnothing, S \leftarrow \varnothing$
2 **foreach** $r \in \mathcal{C}$ **do**
3 **foreach** $t \in pref_\beta\,(r)$ **do**
4 **foreach** $s \in I_t$ **do**
5 **if not** *filter* (r, s)
6 $S \leftarrow S \cup refine\,(r, s)$
7 $I_t \leftarrow I_t \cup \{r\}$
8 **return** S



(a) Effects of threshold values and token frequency distribution

(b) Number of candidates with varying threshold.

(c) Number of candidates with varying q values.

Figure 1. Limitations of algorithms based on prefix filtering.

tokens are used to find candidate sets in the corresponding inverted lists (lines 2–4). Each candidate s is checked using filters, such as positional [Xiao et al. 2011] and length-based filtering [Sarawagi and Kirpal] (line 5); if the candidate passes through, the actual similarity computation is performed in the refine step and r and s are added to the result if they are similar (line 6). Finally, r is appended to the inverted lists of its prefix tokens (line 7). An important observation is that Algorithm 1 is intrinsically sequential: sets, prefix tokens, and candidate sets are processed sequentially, while the inverted index is dynamically created.

3. Limitations of Current Algorithms

Currently, prefix filtering is the prevalent optimization technique for CPU-based, set similarity join algorithms. As such, all these algorithms benefit from its pruning power, but also suffer from its limitations. In particular, prefix filtering effectiveness is heavily dictated by two factors: threshold value and token frequency distribution [Sidney et al. 2015].

There is a clear correlation between threshold values and similarity join performance. Invariably, execution time increases with lower threshold values. The explanation is that lower threshold values imply larger prefixes as illustrated in Figure 1(a). As a result, more inverted lists have to be scanned (Algorithm 1, lines 3–4) and a larger number of candidate pairs have to be verified. Figure 1(b) shows the number of candidates (in log scale) for decreasing Jaccard thresholds on a 100K sample taken from the DBLP dataset (details about the datasets are given in Section 6). As we decrease the threshold from 0.9 to 0.3, there is an increase of three orders of magnitude in the number of candidates.

Disregarding pruning due to other filters, the frequency of tokens in the prefix determines the number of candidates for a given set. The total token order based on

frequency places rare tokens in the prefixes. As a result, inverted lists are shorter and there is much less prefix overlap between dissimilar sets. Of course, the effectiveness of this strategy depends on the underlying token frequency distribution. For a uniform distribution, it behaves not better than an ordinary lexicographical order. Figure 1(a) depicts the impact of the token frequency skew on size of the lists associated with prefix tokens and 1(c) shows the number of candidates (in log scale) for increasing values q-gram size (larger values of q results in more skewed token distribution). We can observe an increase of one order of magnitude in the number of candidates.

The above observations indicate intrinsic limitations of prefix filtering, and, in turn, current set similarity join algorithms. Next, we provide a general overview of the underlying many-core architecture before present our massively parallel approach, which avoids such drawbacks.

4. GPU Architecture and Programming Model

Next we provide a brief description of a GPU architecture and its corresponding programming model (Cf. [Kirk and Wen-mei 2012] for more details). An abstracted architecture view of a GPU is illustrated in Figure 2. This architecture is common to most GPUs currently available. The GPU architecture has two levels of parallelism, where P streaming multi-processors (SMs) are at the first level and p streaming processors (SPs) withn each multi-processor. Thus a parallel program can be first divided into blocks of computation that can run independently on the P SMs (fat cores), without communicating with each other. These blocks have to be further divided into smaller tasks (threads) that execute on the SPs (thin cores), but with each thread being able to communicate with other threads in the same block. Each of these threads has access to a larger global memory as well as to a small but fast shared memory and registers.

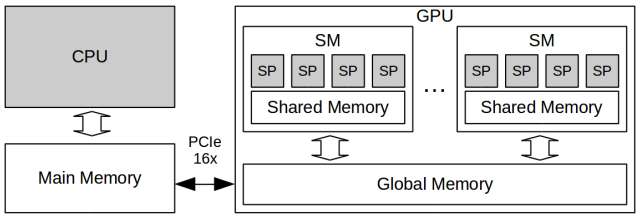


Figure 2. *Overview of a GPU architecture*

The GPU supports thousands of light-weight concurrent threads and, unlike the CPU threads, the overhead of creation and switching is negligible. To hide the global memory's high latency, it is important to have more threads than the number of SPs and to have threads accessing consecutive memory addresses that can be easily coalesced. Another important data movement channel is the PCIExpress connection, whereby CPU and GPU can exchange data between each one's address space but in a much slower speed. The GPU programming model requires that part of the application runs on the CPU while the computationally-intensive part is accelerated by the GPU. A GPU program exposes parallelism through a data-parallel special function, called kernel function. During implementation, the programmer can configure the number of threads to be used. Threads execute data parallel computations of the kernel and are organized in groups (thread blocks)

that are further organized into a grid structure. When a kernel is launched, the blocks within a grid are distributed on idle SMs while the threads are mapped to the SPs.

5. The gSSJoin Algorithm

5.1. Pre-processing

Before performing the set similarity join, we map input strings to sets of tokens using q-grams and create an inverted index in the GPU memory, assuming the input collection fits in memory and is static. Let $\mathcal{S}$ be the set collection and $\mathcal{V}$ be the vocabulary of the collection, that is, the set of distinct tokens of the input collection. The input collection is the set $\mathcal{E}$ of distinct token-sets (t, s) pairs occurring in the original collection, with $t \in \mathcal{V}$ and $s \in \mathcal{S}$. An array of size $|\mathcal{E}|$ is used to store the inverted index. Once the set $\mathcal{E}$ has been moved to the GPU memory, each pair in it is examined in parallel, so that each time a token is visited the number of sets where it appears is incremented and stored in the array $count$ of size $|\mathcal{V}|$. A parallel prefix-sum is executed on the $count$ array by mapping each element to the sum of all tokens before it and storing the results in the $index$ array. Thus, each element of the $index$ array points to the position of the corresponding first element in the $invertedIndex$, where all (t, s) pairs will be stored ordered by token. The cardinality of all sets is computed in parallel with each processor contributing partially (incrementing) in the respective position of the array $cardinality$ of size $|\mathcal{S}|$. Algorithm 2 depicts the data indexing process.

Algorithm 2: $DataIndexing(E)$

input : token-set pairs in $E[\,0\,..\,|\mathcal{E}|-1\,]$.
output: $count$, $index$, $cardinality$, $invertedIndex$.

1 array of integers $count[\,0\,..\,|\mathcal{V}|-1\,]$ `// count array, initialized with zeros.`
2 array of integers $index[\,0\,..\,|\mathcal{V}|-1\,]$.
3 array of integers $cardinality[\,0\,..\,|\mathcal{S}|-1\,]$.
4 $invertedIndex[\,0\,..\,|\mathcal{E}|-1\,]$ `// the inverted index`
5 Count the occurrences of each token in parallel on the input and accumulates in $count$.
6 Perform an exclusive parallel prefix sum on $count$ and stores the result in $index$.
7 Access in parallel the pairs in E, with each processor performing the following tasks:
8 **begin**
9 Contribute on the cardinality computation of each set in $cardinality[s]$.
10 Store in $invertedIndex$ the entries corresponding to pairs in E, according to $index$.
11 **end**
12 **Return** the arrays: $count$, $index$, $cardinality$ and $invertedIndex$.

5.2. Set Similarity Search

After inverted index construction, our algorithm can be viewed as a batch of set similarity search operations. Given an input set, the set similarity search consists of two steps. First, the Jaccard similarity of the input set s to all sets have to be computed. Then, the sets most similar (above the threshold) to the input set, are selected. The Jaccard similarity computation takes advantage of the inverted index model, because only the similarities between the input set s and those sets in $\mathcal{S}$ that have tokens in common with s have to be computed. These sets are the elements of the $invertedIndex$ pointed to by the entries of the $index$ array associated with the tokens occurring in the input set s.

The obvious solution to compute the Jaccard similarity is to distribute the tokens of input set s evenly among the processors and let each processor p access the inverted

lists corresponding to tokens allocated to it. However, the distribution of tokens in sets of the collections follows approximately the Zipf's Law. This means that few tokens occur in a large amount of sets and most tokens occur in only a few sets. Consequently, the size of the inverted lists also varies according to the Zipf's Law, thus distributing the workload according to the tokens of s could cause a great imbalance of the work among the processors.

Thus, we propose a load balance method to distribute the sets evenly among the processors so that each processor computes approximately the same number of Jaccard similarities. In order to facilitate the explanation of this method, suppose that we concatenate all the inverted lists corresponding to tokens in s in a logical vector $E_s = [\,0\,..\,|E_s| - 1\,]$, where $|E_s|$ is the sum of the sizes of all inverted lists of tokens in s. Given a set of processors $\mathcal{P} = \{p_0, \cdots p_{|\mathcal{P}|-1}\}$, the load balance method should allocate elements of E_s in intervals of approximately the same size, that is, each processor $p_i \in \mathcal{P}$ should process elements of E_s in the interval $[i\lceil\frac{|E_s|}{|\mathcal{P}|}\rceil, min((i+1)\lceil\frac{|E_s|}{|\mathcal{P}|}\rceil - 1, |E_s| - 1)]$. Since each processor knows the interval of the indexes of the logical vector E_s it has to process, all that is necessary to execute the load balancing is a mapping of the logical indexes of E_s to the appropriate indexes in the inverted index (array $invertedIndex$). Each processor executes the mapping for the indexes in the interval corresponding to it and finds the corresponding elements in the $invertedIndex$ array for which it has to compute the Jaccard similarity to the input set.

Let $\mathcal{V}_s \subset \mathcal{V}$ be the vocabulary of the input set s. The mapping proposed in this paper uses three auxiliary arrays: $count_s[\,0\,..\,|\mathcal{V}_s| - 1\,]$, $start_s[\,0\,..\,|\mathcal{V}_s| - 1\,]]$ and $index_s[\,0\,..\,|\mathcal{V}_s| - 1\,]$. The arrays $count_s$ and $start_s$ are obtained together by copying in parallel $count[t_i]$ to $count_s[t_i]$ and $index[t_i]$ to $start_s[t_i]$, respectively, for each token t_i in the input set s. Once the $count_s$ is obtained, an inclusive parallel prefix sum on $count_s$ is performed and the results are stored in $index_s$.

Algorithm 3 shows the pseudo-code for the complete similarity search. In lines 3–6, the arrays $count_s$ and $start_s$ are obtained. In line 8, the array $index_s$ is obtained by applying a parallel prefix sum on array $count_s$. Next, each processor executes a mapping of each position x in the interval of indexes of E_s associated to it to the appropriate position of the $invertedIndex$. This mapping is described in lines 10-17 of the algorithm. Then, the mapped entries of the inverted index are used to compute the intersection between each set associated with these entries and the input set. The intersections are computed partially by each processor, but the complete intersections are available when all processors have finished this phase. Lines 20-24 show how the Jaccard similarities are computed and compacted. Each processor is responsible for $\frac{|S|}{|\mathcal{P}|}$ similarities. Each Jaccard similarity calculation uses the intersection (calculated in the previous phase) and the $cardinality$ of the sets. Similarities above the given $threshold$ are flaged and filtered by an exclusive parallel prefix sum. Thus only those similarities are returned (line 22) to the CPU.

5.3. Multi-GPU Similarity Join

The gSSJoin algorithm was designed having in mind a many-core architecture (accelerator) with (global) shared memory. It exploits data parallelism when processing a single input set and uses a single accelerator. It does that by making use of thousands of threads to index the dataset and to find the most similar sets to a given input set. However, when

Algorithm 3: $SimilaritySearch(invertedIndex, s)$

input : $invertedIndex$, $count$, $index$, $cardinality$, $threshold$, $inputset$ $s[\,0\,..\,|\mathcal{V}_s|-1\,]$.
output: Jaccard similarity array $jac\_sim[\,0\,..\,|\mathcal{S}|-1\,]$ initialized with zeros.

1 array of integers $count_s[\,0\,..\,|\mathcal{V}_s|-1\,]$ initialized with zeros
2 array of integers $index_s[\,0\,..\,|\mathcal{V}_s|-1\,]$
3 **for** *each token* $t_i \in s$, *in parallel* **do**
4 $\quad count_s[i] = count[t_i]$;
5 $\quad index_s[i] = index[t_i]$;
6 **end**
7 Perform an inclusive parallel prefix sum on $count_s$ and stores the results in $index_s$
8 **foreach** *processor* $p_i \in \mathcal{P}$ **do**
9 $\quad$ **for** $x \in [i\lceil\frac{|E_s|}{|\mathcal{P}|}\rceil, min((i+1)\lceil\frac{|E_s|}{|\mathcal{P}|}\rceil - 1, |E_S|-1)]$ **do**
$\quad\quad$ `// Map` x `to the correct position` $indInvPos$ `of the` $invertedIndex$
10 $\quad\quad pos = min(i : index_s[i] > x)$;
11 $\quad\quad$ **if** $pos = 0$ **then**
12 $\quad\quad\quad p = 0; offset = x$;
13 $\quad\quad$ **else**
14 $\quad\quad\quad p = index_s[pos-1]; offset = x - p$;
15 $\quad\quad$ **end**
16 $\quad\quad indInvPos = start_s[pos] + offset$
17 $\quad\quad$ uses $s[pos]$ and $invertedIndex[indInvPos]$ in the partial computation of the intersection between s and the set associated to $invertedIndex[indInvPos]$
18 $\quad$ **end**
19 $\quad$ **for** $x \in [i\lceil\frac{|S|}{|\mathcal{P}|}\rceil, min((i+1)\lceil\frac{|S|}{|\mathcal{P}|}\rceil - 1, |S|-1)]$ **do**
$\quad\quad$ `// Jaccard similarity calculation for` $\mathcal{S}$ `sets using` $\mathcal{P}$ `processors`
20 $\quad\quad$ uses intersection and union (through $cardinality$) to compute the Jaccard similarity
21 $\quad\quad$ flag sets with Jaccard similarity above the $threshold$
22 $\quad$ **end**
23 **end**
24 Perform an exclusive parallel prefix sum on the flagged sets to compact the selected sets
25 **Return** the array: jac_sim with the selected jaccard similarities.

dealing with self-joins, the gSSJoin search has to be invoked repeatedly, to deal with the processing of many input sets. The gSSJoin algorithm can handle that by processing the queries one after another, once the input data has been indexed. This streaming operation requires that the most similar sets are returned before another input set can be processed. In addition, a set-specific memory allocation is needed for every set. Moreover, machines with more than one accelerator (GPU) can not take advantage of the extra computing power for the gSSJoin search. These observations have motivated us to extend the gSSJoin to deal with multiple sets in a multi-GPU platform.

In the multi-GPU version, called *mgSSJoin*, task parallelism is exploited in addition to data parallelism. The data indexing step is performed by replicating the input data in each of the g available GPUs and then, in parallel, creating g copies of the inverted index. Thus, each GPU receives the same task and they all produce the same inverted index in their memory. Next, the gSSJoin search proceeds by partitioning the m sets (tasks) into the g GPUs. Each GPU then receives m/g tasks. This is possible since the tasks (sets) are completely independent of each other. Since the sets are of different size, we pre-allocate memory based on the biggest set, i.e., the set with the largest number of tokens). This saves us a lot of time since GPU memory allocation can be very costly. Algorithm 4 shows the pseudo-code for the mgSSJoin. Note that this algorithm, differently from the previous ones, exploits task parallelism and runs on the CPU. The GPU function (kernel) calls (invocations), for data indexing and gSSJoin search, take place in lines 3 and 9 respectively.

Algorithm 4: $MultiGPUSearch(E)$

input : token-set pairs in $E[0..|\mathcal{E}|-1]$.
output: A list of the most similar, one for each set.

```
1  for each i ∈ g, in parallel do
2      set.gpu.device(i);
3      DataIndexing(E);
4  end
5  Allocate memory for the biggest set;
6  for each j ∈ g, in parallel do
7      set.gpu.device(j);
8      for each set s ∈ (m/g) in parallel do
9          SimilaritySearch(invertedIndex, s);
10     end
11 end
12 Return A list of the most similar sets, one for each set.
```

6. Experiments

6.1. Experimental Setup

We used two publicly available, real-world datasets: DBLP [1] contains information about Computer Science publications and Netflix [2] contains information about movies. For DBLP, we randomly selected 20k publications and extracted their title; then, we generated 4 additional "dirty" copies of each string, i.e., duplicates to which we injected textual modifications consisting of 1–5 character-level modifications (insertions, deletions, and substitutions). For both datasets, we converted all strings to upper-case letters and eliminated repeated white spaces. Statistics about the two datasets are shown in Table 6.1. Finally, we tokenized all strings into sets of 2-grams and 3-grams and sorted the sets as described in Section 2.1.

We compared gSSJoin against ppjoin and AllPairs [Xiao et al. 2011], two of three best-performing algorithms according to the evaluation in [Mann et al. 2016]. We used the binaries provided by the authors[3]. We implemented gSSJoin using the CUDA Toolkit version 7.5. The experiments were conducted on a machine running CentOs 7.2.1511 64-bits, with 24 Intel Xeon E5-2620, 16GB of ECC RAM, and four GeForce Zotac Nvidia GTX Titan Black, with 6GB of RAM and 2,880 CUDA cores each.

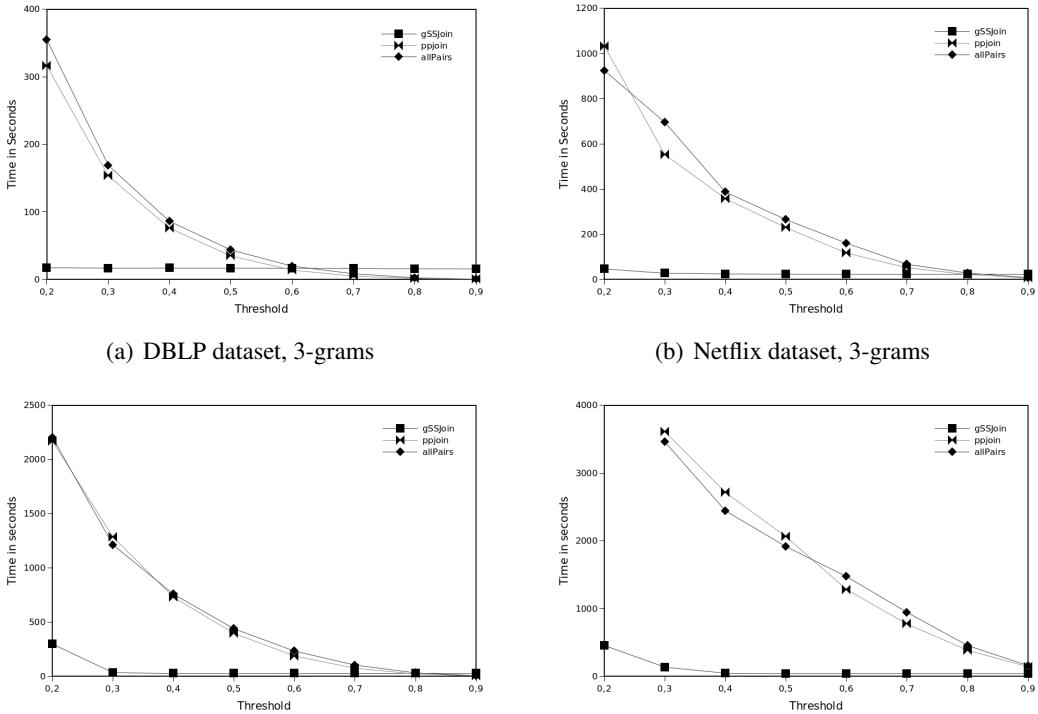
Table 1. Dataset statistics.

Database	Number of sets	Max length	Mean length	Standard deviation
DBLP	100k	205	70	23.9
Netflix	200k	54	30.54	8.2

6.2. Performance Results

We now analyze and compare the efficiency of gSSJoin against ppjoin and AllPairs. To this end, we measured the runtime performance with varying threshold parameters and for q-grams of size 2 and 3. Figure 3 shows the results. As a general trend, all algorithms exhibit similar performance, with some advantage to the latter, at high threshold values

[1] http://dblp.uni-trier.de/
[2] http://www.cs.uic.edu/l̃iub/Netflix-KDD-Cup-2007.html
[3] http://www.cse.unsw.edu.au/ weiw/project/simjoin.html



(a) DBLP dataset, 3-grams

(b) Netflix dataset, 3-grams

(c) DBLP dataset, 2-grams

(d) Netflix dataset, 2-grams

Figure 3. Performance results using varying thresholds.

(DBLP dataset with 3-grams, in particular). However, as the threshold decreases, the performance of ppjoin and AllPairs drops dramatically. As discussed in Section 3, ppjoin and AllPairs, as most set similarity join algorithms, heavily depends on prefix filtering effectiveness to obtain performance. At lower thresholds, filtering becomes ineffective and verification workload skyrockets. In contrast, gSSJoin maintains nearly constant runtimes, thereby achieving increasing speedups over ppjoin.

Furthermore, the performance advantage of gSSJoin over ppjoin and AllPairs is much greater on the Netflix dataset. The token frequency is much less skewed as compared to DBLP, which leads to prefix tokens with higher frequency. As for treshold values, gSSJoin exhibits great robustness to data distribution variations. An identical trend is observed for q-grams of size 2 for the same reason. In fact, the peak speedup gain for gSSJoin is obtained on the Netflix dataset with 2-grams and 0.3 as threshold value: in this setting gSSJoin is 57x faster than the other algorithms. Finally, gSSJoin suffers a drop in performance for threshold value at 0.2 because of the huge number of set pairs in the result that have to be copied to the CPU memory.

Table 2 shows the execution time using multiple GPUs. The use of multiple GPUs showed an almost linear speedup increase. The data is easily divided between the GPUs, making our solution capable of dealing with big datasets.

Table 2. Execution Time (secs) Using Multiple GPUs, Threshold = 0.5

Number of GPUs:	1	2	3	4
DBLP 2gram	27.56	14.37	10.4	7.2
DBLP 3gram	16.73	8.77	6	4.47
Netflix 2gram	41.10	21.63	14.37	11.2
Netflix 3gram	23.52	12.69	8.5	6.5

7. Related Work

There is a substantial body of literature on the efficient computation of set similarity joins. Here, we focus on key related work and refer the reader to [Mann et al. 2016] for a recent experimental evaluation of several state-of-the-art set similarity join algorithms.

Most set similarity join algorithms proposed so far are intrinsically sequential [Sarawagi and Kirpal , Chaudhuri et al. 2006, Bayardo et al. 2007, Xiao et al. 2011, Ribeiro and Härder 2011, Wang et al. 2012] and, therefore, cannot fully exploit the modern computer architectures. Main optimizations of such algorithms exploit the threshold and the underlying data characteristics to derive efficient filters [Sidney et al. 2015] — prefix filtering as prime example.

The processing of set similarity joins can easily become prohibitive for large datasets and high-dimensional space. Thus, some parallel solutions have been proposed to speedup this process. For example, a recent work proposed to perform similarity joins on a distributed memory MapReduce framework [Vernica et al. 2010]. The authors report some performance gains, but the high communication costs limit their solution. Other works use a fine-grained parallel approach on shared-memory architectures.

Another form of optimization appears in *approximate set similarity joins* that use some sort of data reduction technique to speed up processing at the cost of missing some valid results. Locality Sensitive Hashing (LSH) is the most popular technique for approximate set similarity joins [Indyk and Motwani 1998]. The work in [Cruz et al. 2015] proposes an approximate set similarity join algorithm designed for a many-core architecture (GPU). The authors estimate the Jaccard similarity between two sets using MinHash [Broder et al. 1998], an LSH scheme for Jaccard. Approximate approaches can obtain good speedups, but at the cost of missing some valid output set pairs.

Our proposal differs from the above mentioned work in many aspects. First, our proposal finds *exact* answers to set similarity join algorithms. However, our techniques are orthogonal to the approximated solutions and can be combined with LSH (used in a pre-processing step) for better performance. Second, our approach exploits massive parallelism instead of filtering; as observed in our experimental results, this approach exhibited better robustness to variations of threshold and data distributions.

8. Conclusions and Future Works

In this paper, we proposed the gSSJoin algorithm, a GPU-based solution to the set similarity join problem. Our solution exploits intense GPU occupancy, hierarchical memory, and coalesced memory access to achieve much better performance than the-state-of-the-art CPU-based algorithms in most settings. We also proposed a multi-GPU variant to further exploit task parallelism in addition to data parallelism. In future work, we plan

to investigate the integration of GPU-based algorithms into a distributed framework and incorporate candidate filtering techniques to obtain even greater speedups.

Acknowledgments This research was partially supported by CAPES, CNPq, FAPEG, and FINEP. We thank NVIDIA for equipment donations.

References

[Bayardo et al. 2007] Bayardo, R. J., Ma, Y., and Srikant, R. (2007). Scaling up All Pairs Similarity Search. In *WWW*, pages 131–140.

[Broder et al. 1998] Broder, A. Z., Charikar, M., Frieze, A. M., and Mitzenmacher, M. (1998). Min-Wise Independent Permutations (Extended Abstract). In *STOC*, pages 327–336.

[Chaudhuri et al. 2006] Chaudhuri, S., Ganti, V., and Kaushik, R. (2006). A primitive operator for similarity joins in data cleaning. In *ICDE*, page 5.

[Cruz et al. 2015] Cruz, M. S. H., Kozawa, Y., Amagasa, T., and Kitagawa, H. (2015). GPU Acceleration of Set Similarity Joins. In *DEXA*, pages 384–398.

[Hassanzadeh et al. 2009] Hassanzadeh, O., Chiang, F., Miller, R. J., and Lee, H. C. (2009). Framework for Evaluating Clustering Algorithms in Duplicate Detection. *PVLDB*, 2(1):1282–1293.

[Indyk and Motwani 1998] Indyk, P. and Motwani, R. (1998). Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *STOC*, pages 604–613.

[Kirk and Wen-mei 2012] Kirk, D. B. and Wen-mei, W. H. (2012). *Programming massively parallel processors: a hands-on approach*. Newnes.

[Leskovec et al. 2014] Leskovec, J., Rajaraman, A., and Ullman, J. D. (2014). *Mining of Massive Datasets, 2nd Ed*. Cambridge University Press.

[Mann et al. 2016] Mann, W., Augsten, N., and Bouros, P. (2016). An Empirical Evaluation of Set Similarity Join Techniques. *PVLDB*, 9(9):636–647.

[Ribeiro and Härder 2011] Ribeiro, L. A. and Härder, T. (2011). Generalizing Prefix Filtering to Improve Set Similarity Joins. *Information Systems*, 36(1):62–78.

[Sarawagi and Kirpal] Sarawagi, S. and Kirpal, A. Efficient Set Joins on Similarity Predicates. In *SIGMOD*.

[Sidney et al. 2015] Sidney, C. F., Mendes, D. S., Ribeiro, L. A., and Härder, T. (2015). Performance Prediction for Set Similarity Joins. In *SAC*, pages 967–972.

[Vernica et al. 2010] Vernica, R., Carey, M. J., and Li, C. (2010). Efficient Parallel Set-similarity Joins using MapReduce. In *SIGMOD*, pages 495–506.

[Wang et al. 2012] Wang, J., Li, G., and Feng, J. (2012). Can We Beat the Prefix Filtering?: An Adaptive Framework for Similarity Join and Search. In *SIGMOD*, pages 85–96.

[Xiao et al. 2011] Xiao, C., Wang, W., Lin, X., Yu, J. X., and Wang, G. (2011). Efficient Similarity Joins for Near-duplicate Detection. *TODS*, 36(3):15.