Exact Set Similarity Joins for Large Datasets in the GPGPU paradigm

Christos Bellas
Aristotle University of Thessaloniki, Greece
chribell@csd.auth.gr

Anastasios Gounaris
Aristotle University of Thessaloniki, Greece
gounaria@csd.auth.gr

ABSTRACT

We investigate the problem of exact set similarity joins using a co-process CPU-GPU scheme. We focus on large instances of the problem, i.e., using datasets of >1M entries, which may take hours to complete if not approached with care, due to the inherent quadratic complexity of the problem. We introduce a novel CPU-GPU co-process scheme, which performs initial filtering and indexing on the CPU and delegates final verification to the GPU. Further, we show that this scheme improves upon the state-of-the-art in both the CPU and GPU standalone solutions in several cases.

CCS CONCEPTS

• **Information systems → Data management systems**;

ACM Reference Format:
Christos Bellas and Anastasios Gounaris. 2019. Exact Set Similarity Joins for Large Datasets in the GPGPU paradigm. In *International Workshop on Data Management on New Hardware (DaMoN'19), July 1, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3329785.3329919

1 INTRODUCTION

In this work, we deal with exact set similarities joins, are used in a range of applications, such as plagiarism detection, web crawling, clustering and data mining and have been the subject of extensive research recently, e.g., [3, 5, 13, 18, 23]. The goal of exact set similarity joins is to find all the pairs of sets that overlap above a user-defined threshold. We focus on datasets that are large, i.e., they contain over 1M sets, which results in billions of pairs that need to be checked in practice. Most of the techniques to date for large datasets target the MapReduce paradigm e.g., [2, 9, 19, 23, 24]. In this work, we explore the case to efficiently employ a GPU to enhance sequential algorithms, as these are investigated in [13, 18].

To date, the problem of exact set similarity on GPUs has been addressed in [22]; however, as we will show later, there are specific cases where this technique is not suitable for large datasets. In the literature, there exist several proposals for approximate set similarity or for similar problems, such as nearest neighbor search, e.g., [7, 14, 16]. Therefore, there is a gap in detailed investigation of exact similarity joins in GPUs tailored to large datasets. The main contribution of this work is to fill this gap and propose an efficient solution after thoroughly investigating several design alternatives.

Our work incorporates a co-process scheme between CPU and GPU in order to efficiently compute the set similarity join: the CPU remains responsible for index building and initial pruning of candidate pairs, whereas the GPU computes the overlap of all remaining pairs. As shown in the real experiments, this may lead to improvements, although the maximum speed-ups is bounded because of Amdahl's law. Moreover, the GPU part comes with several challenges regarding the data serialization and layout, the thread management and the techniques to compare sets of tokens. In our work, we address these challenges and manage to achieve speed-ups up to 1.9X compared to the best between the CPU and the GPU solution in [22].

In summary, the technical contributions of our work are twofold: (i) We provide a detailed description of a co-process framework and we propose alternatives that differ in the workload allocated to each GPU thread. We use the CUDA programming model, which is proprietary to NVIDIA [15] but widespread in practice. [1] (ii) We conduct extensive performance analysis on real world datasets. We compare our findings to the state of art CPU and GPU implementations and point out strengths and weaknesses of each.

Paper outline. Next, we provide details on the main CPU-based algorithms and the respective standalone GPU approach to set similarity joins. We present our solutions and the design alternatives involved in Section 3. The experimental results are in Section 4. We discuss our findings and potential future work in Section 5. Finally, we conclude

DaMoN'19, July 1, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-6801-8/19/07…$15.00
https://doi.org/10.1145/3329785.3329919

[1] the code is publicly available from https://github.com/chribell/gpussjoin

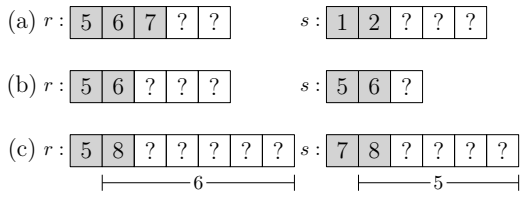

Figure 1: Filters used for candidate pruning: (a) prefix, (b) length, (c) positional

in Section 6. In the appendix, we provide a short CUDA overview and further implementation details.

2 BACKGROUND

The state-of-the-art main memory set-similarity algorithms conform to a common filter-verification framework, as explained in [18], in which seven key representatives[2] are compared using real world datasets. The common idea behind all these algorithms is (i) to avoid comparing all possible set pairs by applying filtering techniques on preprocessed data to prune as much candidate pairs as possible; and (ii) then to proceed to the actual verification of the remaining candidates. We summarize the key points of the work of [18] that are relevant to our research below.

2.1 Data layout.

Every dataset is a collection of multiple sets. Each set consists of elements called *tokens*. The data preprocessing phase involves a tokenization technique and deduplication of tokens. As a result of such a preprocessing phase, all the tokens of a set are unique. The input data tokens are represented by integers and are sorted by their frequency in increasing order, so that infrequent tokens appear first in a set. The sets of a collection are sorted first by their size and then lexicographically within each block of sets of equal size.

2.2 Set Similarity functions.

To measure the similarity between sets, Jaccard, Dice and Cosine normalized similarity functions are typically used. The given normalized threshold t_n is translated to an equivalent overlap t, which defines the minimum number of tokens that need to be shared between two sets to satisfy the threshold (see Table 1). For example, if the Jaccard similarity threshold of two 10-token sets is set to 0.8, this is translated to an overlap threshold of 9 tokens that need to be shared.

2.3 Filters.

The most widely used filter, called *prefix-filter*, exploits the given threshold and similarity function by examining only two subsets, one from each set in the candidate pair, and discards the pair if there is no overlap between the subsets. For example, in Figure 1(a), there is no overlap between the respective set prefixes, thus, even if there is an overlap on the remaining tokens, any overlap threshold set to 4 or higher cannot be reached, and in such cases, the candidate pair (r, s) can be safely pruned.

Another filter, noted as *length filter*, takes advantage of the normalized similarity functions dependency on set size. Hence, a candidate pair can be pruned if the set size inequality $t_n \cdot |r| \leq |s| \leq |r|/t_n$ is not satisfied. In Figure 1(b), if $t_n = 0.8$, the shown candidate pair (r, s) can be pruned despite the prefix equality because a 6-token r set requires a s set of size $4 \leq |s| \leq 6$.

The last filter used in the examined algorithms is the *positional filter*. Given the first match position, it evaluates if a candidate pair can reach the similarity threshold. As an example, in Figure 1(c), if the threshold implies that at least 6 tokens should be shared, the pair is pruned since the remaining tokens from set s are not enough to reach the similarity threshold.

Table 1: Similarity Functions (adapted from [18])

Similarity function	Definition	Equivalent Overlap
Jaccard	$\frac{\lvert r \cap s \rvert}{\lvert r \cup s \rvert}$	$\lceil \frac{t_n}{1+t_n}(\lvert r \rvert + \lvert s \rvert) \rceil$
Cosine	$\frac{\lvert r \cap s \rvert}{\sqrt{\lvert r \rvert \lvert s \rvert}}$	$\lceil t_n \sqrt{\lvert r \rvert \lvert s \rvert} \rceil$
Dice	$\frac{2\lvert r \cap s \rvert}{\lvert r \rvert + \lvert s \rvert}$	$\lceil \frac{t_n(\lvert r \rvert + \lvert s \rvert)}{2} \rceil$
Overlap	$\lvert r \cap s \rvert$	t

2.4 Algorithm outline.

The set similarity join operation is achieved by executing an index nested loop join consisting of three steps. First, through an index lookup and a length filter application, a preliminary candidate set (pre-candidates) is generated. In the second step, pre-candidates are deduplicated and filtered. The pairs that pass all filters form the final candidates. These two steps compose the *filtering* phase. In the third and final step, also noted as *verification* in the literature, the similarity score for each of the remaining candidate pair is computed and if it exceeds the threshold, the pair is added to the output result.

Most commonly, the set similarity join is a self-join using only a single collection of sets. In that case, a token set is first probed against the current index contents and then added to the index itself. This allows for incremental index building that is interleaved with verification. Also, the fact that a set

[2] AllPairs [3], PPJoin and PPJoin+ [27], MPJoin [21], MPJoin-PEL [17], AdaptJoin [25] and GroupJoin [5].

Table 2: Notation

R, S	Collections of sets to be joined
r_i (resp.s_j)	a token set from R (resp. S)
τ_n	Normalized similarity threshold
$C \subseteq R \times S$	Set of candidate pairs
O	Device output
R_T, S_T	Token arrays
R_O, S_O, C_O	Offset arrays
$\|\|R_T\\|\|$, $\|\|S_T\\|\|$,$\|\|C\\|\|$,$\|\|O\\|\|$ $\|\|R_O\\|\|$, $\|\|S_O\\|\|$,$\|\|C_O\\|\|$	Size of arrays in bytes
H_i	The i^{th} host thread, $i \in 0, 1, 2$
T	Number of device threads
B	Thread block size
M_h	Host memory
M_d	Device memory
$M_c < M_d$	Device memory for candidate pairs

that probes the index is always no shorter than the current indexed sets can be leveraged to speed-up verifications.

2.5 GPU standalone approach

The authors in [20] propose a standalone GPU framework, noted as *fgSSJoin*, in which the complete workload is delegated to the GPU. To be able to conduct the set similarity join, a block partitioning scheme is adopted in order to process collections of arbitrary size. In summary, the input collection is divided into blocks of size n. Through an iterative process, a block is indexed and probed against itself and all its predecessors. Each probe is processed in two separate kernel calls in $O(n^2)$ memory space, (i) a *filtering kernel* in which partial intersection counts between indexed and probed sets prefixes are calculated and stored in global memory; and (ii) a *verification kernel* in which every pair that has a non-zero partial intersection count undergoes full verification, whereas the rest are pruned. In addition, an entire pair of blocks may be pruned before invoking the GPU due to length filter. The output result is stored in linear global memory and is transferred back to main memory via the PCI-E bus.

3 OUR APPROACH

Let R and S be two collections of token sets, $Sim()$ be a similarity function and $\tau_n \in [0, 1]$ the user-defined threshold. The set similarity problem is formally defined as follows.

Definition 3.1. Problem Definition of set similarity joins: Find all pairs $(r, s), r \in R,\ s \in S$ such that $Sim(r, s) \geq \tau_n$.

In a naive solution, the set of candidate pairs C to be checked in the verification phase is all pairs $R \times S$. However, due to the filtering phase, for thresholds not close to 0, C is typically a small subset of the cartesian product. The output of the verification is denoted as O.

In our solution, we assume that the host (resp. device) is equipped with M_h (resp. M_d) memory capacity. The host runs 3 threads ($H_0,\ H_1,\ H_2$), while the device executes T threads in blocks of size B. The collections of token sets are transferred in the device main memory in a linearized form, denoted as R_T and S_T, and are accompanied by offset arrays R_O and S_O in order to distinguish the set boundaries. Table 2 summarizes the notation.

3.1 Main Rationale

As already explained, the CPU-based algorithms solving the set similarity problem efficiently conform to the filter-verification framework. The filtering phase involves probing index structures such as a hashtable. Although there has been some work on implementing hashtables and inverted lists on GPUs [1], which can be employed in exact set similarity joins as in [20], we choose this phase to remain a CPU task. On the other hand, the verification phase is more suitable for parallelization, as it involves a merge-like loop, where the overlap of candidate pairs is computed. As soon as the overlap threshold is met or cannot be reached, the operation terminates. True positives must be verified and the necessary overlap is still computed, while the rejection of pairs in this stage leads to less token comparisons without sacrificing accuracy. Even though the average verification runtime is reported as constant for most datasets, employing the GPU for this part with a view to improving the overall performance is the main goal of this work.

In summary, we allocate the initial indexing and filtering to the CPU and the verification phase to the GPU. Based on the results both from [18] and our tests, in the following, we focus solely on the best three CPU algorithms: (i) All-Pairs(ALL) [3] (naive, applies prefix and length filters), (ii) PPJoin(PPJ) [27] (applies positional filter as well) and (iii) GroupJoin(GRP) [5] (additionally groups sets with identical prefix). We experiment with 5 real datasets (full details are provided in Section 4). The key observation of our tests is that filtering phase contributes to the total running time significantly. Therefore, due to the Amdahl's law, employing the GPU for the verification in an ideal setting is expected to yield improvements of several times, but lower of an order of magnitude.

When looking for similar sets in a single collection, which is the most common case, instead of looking for similar pairs between two different collections, the whole join process can be done incrementally, i.e. for a probing set, first its candidates are verified and then the algorithm proceeds to the next set. Naively allocating and copying small chunks of data on the GPU through a different kernel invocation per probing set, would incur an enormous overhead penalty. A more efficient alternative is to copy a large chunk of data stored

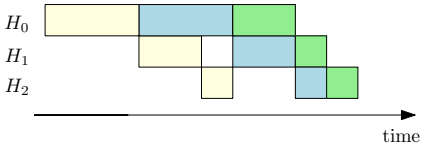

Figure 2: Execution overlap between host (H_0, H_2) and device (H_1)

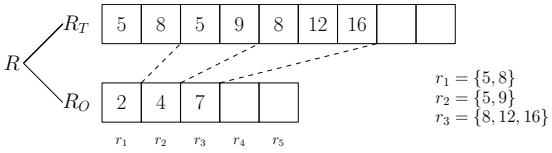

Figure 3: Example layout of a collection of threes sets in the device memory

in linear memory space to the GPU, process it there and copy back the results. Adopting this approach also improves overall runtime as the CPU builds candidate pair collections in waves and feeds them in a non-blocking manner to the GPU, which conducts the verification. Thus, time overlapping between the CPU and GPU tasks can be achieved. More specifically, we propose a multithreaded framework regarding both the host and device tasks. We thoroughly analyze each side below. The most dominant constraining factor is the limited GPU memory. Due to this, the workload must be divided into chunks and the GPU should be invoked several times. The memory limitation mostly relates to the output size. As we have no prior knowledge about the output size, to ensure correctness the most straightforward solution is to allocate enough memory for the worst case, which is $O(|R||S|)$.

3.2 Host Tasks

The host side is responsible for the filtering phase and works as the coordinator. Specifically, the host runs three threads (see Figure 2). The first thread, H_0, conducts any filtering and builds chunks of candidates. When each chunk is built, the second thread, H_1, noted as *device handler*, enacts the verification phase by copying the chunk to device memory and launching the kernel code. Meanwhile, H_0 continues to build the next chunk of candidates. As soon as the device output is copied back to host memory, the third thread H_2 post-process it to form the final pairs result. Note that H_2 may not be invoked if an aggregation is performed on top of the join, i.e., if only the count of pairs is needed instead of the actual pairs. In such a case, the device counts the number of pairs and returns the result to H_1.

3.3 Device Tasks

The device side is responsible for the verification phase. It is invoked when the host prepares a chunk of candidate pairs. We present our data layout approach and discuss its impact on device memory.

There are two levels of concurrency in CUDA, *grid* and *kernel*. Grid level concurrency concerns mostly the overlap between computation and data transfers, while kernel level concurrency refers to how a single task is executed in parallel by many threads [6]. On the grid level, we further divide each input chunk of candidates into smaller chunks, each assigned to a different block. Thus we enhance the overlapping between device computation and host-to-device data transfer. On the kernel level, we summarize our approaches as high-level verification alternatives below.

3.3.1 Data Layout. By default, we pass data to device as arrays stored in consecutive memory space. This is preferred because parallel execution benefits from coalesced global memory accesses. However, due to the nature of the problem, divergence in global memory access patterns is unavoidable, therefore the exploitation of on-chip memories is required to alleviate performance bottlenecks.

According to the linear memory layout, a collection R is physically implemented as the composition of two arrays: tokens R_T and offsets R_O. The former holds every token of every set in the collection in a sequence, while the latter is used to delimit each set boundaries. In practice, this layout has the less memory access overhead. Figure 3 depicts how a collection of sets R is stored in the device memory. Collection S is stored in similar fashion. We transfer any collection of sets once in the beginning of the process. When the device is invoked to perform the verification phase, the host transfers an array of set IDs, noted as C, alongside with an array of offsets (C_O) which indicates the candidate pairs to be evaluated. In addition, an array of equal length to C, noted as O, is allocated on the device and it is used to store the output result. Essentially, O is an array of boolean flags where true indicates that the corresponding candidate pair similarity is equal or greater than the given threshold.

3.3.2 Verification alternatives. We introduce three alternative scenarios which differ in the workload assigned to a single thread. For example, as shown in Figure 4, in alternative A, a single device thread verifies probe set r_3 and all its corresponding candidates $\{s_1, s_3, s_4, s_7\}$. In the slightly altered alternatives B and C, a single device thread block verifies probe set r_3 and all its corresponding candidates. For alternative B, each thread verifies independently a candidate pair, whereas in alternative C threads collaborate to verify a candidate pair.

In order to analyze the design choices for each alternative, we define three main kernel-level concurrency layers or dimensions of our problem as follows: *grid layout* (**GL**), which corresponds to thread execution; *memory hierarchy* (**MH**),

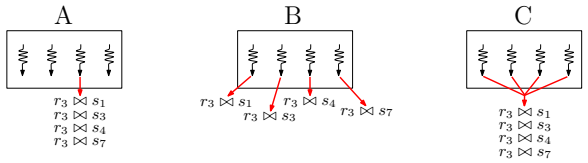

Figure 4: Thread workload per alternative

which corresponds to efficient exploitation of the fastest on-chip memories; and last *output writing* (**OW**), which deals with result output. The latter is also distinguished into two cases depending on the type of querying being performed: *output count* (**OC**) for an aggregate query and *output select* (**OS**) for a full select of the similar pairs query. These layers are tightly coupled and often intertwined, which means that certain options on a layer can rule out available options on the next ones. We further present each verification alternative below.

Alternative A. In our first alternative, the workload we assign to each thread is a probing set and the evaluation of all its corresponding candidate pairs, i.e., a single thread becomes responsible for the verification of all candidate pairs involving a specific probing set.

GL: We launch a 1D grid of 1D blocks, with the overall number of threads executed across all blocks (T) being equal to the input set collection size (R). Each thread is responsible for a probing set r_i and conducts all the joins with the corresponding candidates s_j.

MH: In this alternative, we do not use the shared memory. Since there is no fixed set size, blocks which handle sets of thousand tokens require larger amount of shared memory. For example, a thread block of 32 threads and average probing set size equal to 1000, would require $32 \times 1000 \times 4 = 128$KB of shared memory which exceeds the maximum of 48KB that modern GPUs can support.

OC: As every thread verifies its own candidate pairs independently, it can also count the amount of pairs satisfying the threshold using a register. After finishing the verification, each counter can be stored in shared memory in order for a fast reduction on block level to be performed. The result of each block is then stored in global memory for a grid level reduction to output the global count. The amount of shared memory required depends on the block size, but it is small (e.g., for 32 threads per block we need 128 bytes).

OS: Having allocated the memory required for output array O, a device thread updates specific cells of the array. Incorporating the shared memory in this output is not straightforward because each thread does not know beforehand the length of its output pairs and in the worst case it may exceed the maximum allowed space. As a result, the shared memory cannot be employed to speed-up the output generation.

Alternative B. In this technique, we allocate less work to each thread by shifting the workload of a single probing set from a single thread to a single thread block. By assigning the comparisons referring to a probing set to a thread block, threads evaluate only a portion of the candidate pairs in parallel. The main benefit of this alternative is that the workload of threads within a block is more evenly distributed.

GL: We launch a 1D grid of 1D blocks, with the number of blocks being equal to the input set collection size. Each thread block is responsible for a probing set and each thread is assigned with a portion of candidate pairs to verify.

MH: First, the block threads load the corresponding probing set r_i to shared memory, then each thread verifies a portion of candidate pairs by accessing the corresponding candidate sets s_j from global memory. Because we use shared memory for one probing set per block, unlike alternative A, the maximum supported probing set size also increases.

OC/OS: Same as Alternative A.

Alternative C. In our previous scenario, we try to improve performance on the warp level. We further extend the rationale of alternative B, and more specifically, each block is assigned with a probing set but with the difference that all the block threads cooperate to evaluate a candidate pair using the intersect path algorithm proposed in [12]. This further mitigates the problem of balancing, since the threads do not only become responsible for an equal number of candidates, but also perform a roughly equal number of operations.

GL: We launch a 1D grid of 1D blocks, with a number of thread blocks equal to the input set collection size. Each thread block is responsible for a probing set and multiple threads contribute to each candidate pair verification. A control thread outputs the result to global memory.

MH: Extending alternative B, due to thread cooperation, by default we also load candidate sets to shared memory. If there is not enough space to hold all candidates, we load data in chunks, perform the verification, and then proceed to the next chunk.

OC: Since all threads within a block cooperate to verify a candidate pair, we assign only a single thread to increment the block's counter. Thus there is no need for a thread block counter reduction. However, grid level reduction is still required.

OS: The same applies for updating the output array O. If a candidate pair meets the threshold, only a single thread updates the corresponding array cell.

In the appendix, we provide the full implementation details covering aspects such as candidate serialization, thread and memory management, block size, merge and intersect path computation, set intersection count and count reduction. We also explain that in practice, it is beneficial to employ alternative B for sets of small size in terms of the tokens they include, and alternative C otherwise.

Table 3: Datasets characteristics

Dataset	Cardinality	Avg set size	# diff tokens
AOL	$1.0 \cdot 10^7$	3	$3.9 \cdot 10^6$
DBLP (2-grams)	$6.5 \cdot 10^6$	88.5	$2.8 \cdot 10^4$
DBLP (4-grams)	$6.5 \cdot 10^6$	90.5	$8.7 \cdot 10^5$
DBLP (6-grams)	$6.5 \cdot 10^6$	92.5	$9.1 \cdot 10^6$
DBLP (8-grams)	$6.5 \cdot 10^6$	94.5	$3.7 \cdot 10^7$
LIVEJOURNAL	$3.1 \cdot 10^6$	36.5	$7.5 \cdot 10^6$
ORKUT	$2.7 \cdot 10^6$	120	$8.7 \cdot 10^6$
TWITTER	$1.6 \cdot 10^6$	75	$3.7 \cdot 10^4$

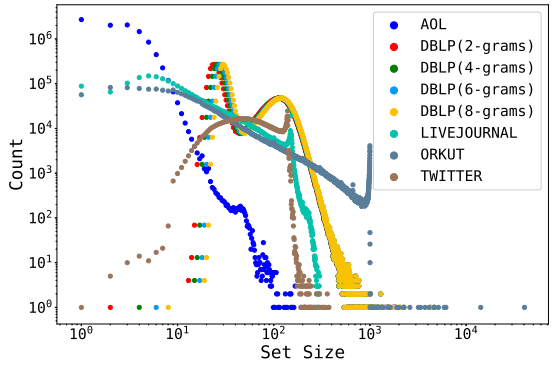

Figure 5: Datasets set size distribution

4 EVALUATION

The goals of our experiments are threefold: (i) to compare our co-process scheme against the state of the art standalone implementations; (ii) to give concrete evidence about the speed-ups achieved in practice, and (iii) to provide explanations about the observed behavior.

4.1 Experiment setting.

The experiments were conducted on a machine with an Intel i7 5820k clocked at 3.3GHz, 32 GB RAM at 2400MHz and an NVIDIA Titan XP. This GPU has 3840 CUDA cores, 12 GB of global memory and a 384-bit memory bus width.

The overall runtime is the composition of candidate generation and serialization performed by H_0 host thread, and the verification conducted by the device. We do not include any data preprocessing time spent for tokenization and deduplication, which are perfomed exactly as in [18] and are considered not relevant to the join time. We conduct experiments for all datasets using five thresholds in the range [0.5, 0.9]. We focus on self-joins using the Jaccard similarity and perform an aggregation on top of the set similarity join. The reported time for each experiment is an average over 3 independent runs (no significant deviation was observed). We measure the *index/filtering* and total *join* time with the *std::chrono* library. For the *verification* time we use the CUDA event API. The times for allocating device memory and transferring chunks of candidates to the device were negligible for all the experiments and furthermore were completely hidden due to the execution overlap.

We experiment with five real world datasets; four of them were also employed in [18] and we also employed the TWITTER dataset found in [10]. We further process the DBLP dataset to measure the performance impact of the number of different tokens. Table 3 shows an overview of each dataset characteristics. Some datasets follow a Zipf-like distribution of set sizes, as shown in Figure 5, but in general the distribution types differ.

4.2 Main Experiments.

We compare the state-of-the art CPU standalone implementation of Mann [18] against its GPU-accelerated version presented in this work, noted as *CPU-GPU*, and the best GPU standalone solution described in [20]. As can be observed in Table 4, there is not a dominant technique. For high threshold values ([0.8, 0.9]), the CPU remains very competent due to the effective filtering, while GPU-based solutions seem beneficial for lower threshold values ([0.5- 0.7]), where billions of candidate pairs are evaluated. Nevertheless, there is no clear pattern even when the threshold are in [0.5- 0.7]. For AOL and DBLP (6, 8 grams), our CPU-GPU solution is faster. The GPU standalone solution performs better for the rest of datasets. We explain that later. In general, when CPU-GPU is faster, the maximum observed speed-up is 1.9X compared to the best between the CPU and the GPU solution in [22]. When compared solely against *fgSSJoin*, the speed-ups can be up to 1-2 orders of magnitude, e.g., for the AOL dataset.

4.3 Performance Analysis.

Motivated by the fact that neither technique is the most dominant one, we analyze the impact of three key dataset characteristics, (i) set size distribution; (ii) average set size and (iii) number of different tokens, on the GPU-enabled techniques' performance.

4.3.1 Set size distribution. For datasets with a high proportion of similar set sizes, such as AOL as shown in Figure 5, length filtering on block probe level is ineffective. When applying *fgSSJoin* on AOL, this results in a higher number of GPU calls, which, in turn, increases the clear operation overhead of the quadratic memory space. The bottom line is that in large datasets, especially when, due to the dataset size and high number of similar set sizes, the length filter cannot prune many pairs, the overhead associated with the

Table 4: Comparison between the best times for different thresholds (in seconds)

	τ_n														
	0.5			0.6			0.7			0.8			0.9		
	CPU	CPU-GPU	fgSS	CPU	CPU-GPU	fgSS	CPU	CPU-GPU	fgSS	CPU	CPU-GPU	fgSS	CPU	CPU-GPU	fgSS
AOL	276	**207**	694	69	**50**	466	10	**7**	311	3.1	**2.7**	240	**1**	1.1	202
DBLP (2-grams)	many hours			many hours			many hours			91049	36358	**7437**	8117	3152	**241**
DBLP (4-grams)	51069	21005	**2915**	19398	8750	**1571**	5998	2877	**576**	1292	686	**231**	103	135	**80**
DBLP (6-grams)	2776	**1334**	1449	934	**583**	658	279	**147**	301	**73**	76	175	**14**	15	75
DBLP (8-grams)	441	**287**	out of memory	149	**145**	581	59	**58**	276	**21**	22	170	**5.8**	6.2	78
LIVEJOURNAL	277	188	**65**	70	49	**37**	17	**13**	21	5	**4**	13	**1.35**	1.36	7
ORKUT	161	129	**78**	59	57	**39**	24.1	23.7	**19**	**9**	10	10	**3**	3.3	5
TWITTER	36318	24401	**838**	15154	9857	**371**	4698	3002	**146**	898	557	**48**	63	46	**10**

quadratic space complexity is not outweighed by any benefits compared to the CPU solution, and, overall *fgSSJoin* is not the optimal GPU-enabled technique. In contrast, our CPU-GPU solution does not entail this kind of overhead and performs better, especially in lower thresholds where execution overlap between both ends becomes prominent. However, the improvements on a CPU-based solution are up to 1.39X only.

4.3.2 Average set size. For datasets with small average set size (≤ 10) such as AOL the global memory access footprint is also small, which renders alternatives A and B of CPU-GPU competitive. However, as the average set size increases (in DBLP, LIVEJOURNAL, ORKUT, TWITTER), alternative C is preferred to minimize the global memory access bottleneck. Thus, it is more beneficial to employ alternative B for sets of small size, and alternative C otherwise. On the other hand, *fgSSJoin* remains robust against varying average set sizes.

4.3.3 Number of different tokens. *fgSSJoin* replicates input tokens to perform the verification phase faster. In case of lower thresholds and high number of different tokens the required memory space to store the replicated data increases and as a result the available free memory space to process input sets is decreased (in particular DBLP 8-grams, with $\tau_n = 0.5$ cannot run with 12GB of global memory). This leads to smaller partition blocks, and thus to more GPU calls with the subsequent overhead costs. In contrary, the GPU phase of the CPU-GPU solution is independent of the number of different tokens. In general, CPU-GPU benefits from the combination of large number of different tokens and high cardinality. Such a combination appears in DBLP (6, 8 grams). When the number of different tokens is high but the cardinality is lower, such as in LIVEJOURNAL and ORKUT, CPU-GPU does not always outperform *fgSSJoin*.

5 DISCUSSION

As presented in our evaluation section, the proposed CPU-GPU hybrid approach performs better than the respective standalone solutions in several cases. However, in all of the examined techniques the *prefix*-based filtering phase dominates the overall runtime and therefore faster candidate generation techniques should be investigated as indicated in [18].

Another filtering approach in [8], noted as *partition*-based, advocates partitioning sets in disjoint subsets. In that work, two sets are considered as candidates, only if the share a common subset. In [26], Wang et al. point out the redundant computational cost due to the incrementally join process. Thus, by leveraging the relations among sets, they propose a technique for faster candidate generation, in which inverted lists are rearranged in order to be able for candidates to be skipped on lookup. Both [8, 26] are yet to be explored in the GPGPU paradigm.

Finally, the verification phase should be also further investigated for additional improvements, e.g., through reducing idle times for many GPU cores.

6 CONCLUSION

This work describes a thorough investigation regarding the design alternatives for the verification phase in exact set similarity joins using a GPU. We conform to the established filter-verification framework, and we transfer verification to the GPU. Using real datasets, we show that our co-process scheme performs better than the state-of-the-art CPU and GPU approaches in several cases when the datasets are large. However, there is still room for additional and perhaps hybrid techniques that combine the strengths of all the techniques considered in this work to develop a globally better solution.

Acknowledgement. The authors gratefully acknowledge the support of NVIDIA Corporation through the donation of the GPU used.

REFERENCES

[1] Saman Ashkiani, Martin Farach-Colton, and John D Owens. 2018. A dynamic hash table for the GPU. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 419–429.

[2] Ranieri Baraglia, Gianmarco De Francisci Morales, and Claudio Lucchese. 2010. Document Similarity Self-Join with MapReduce. In *ICDM*. 731–736.

[3] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. 2007. Scaling up all pairs similarity search. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*. 131–140.

[4] Christos Bellas and Anastasios Gounaris. 2017. GPU processing of theta-joins. *Concurrency and Computation: Practice and Experience* 29, 18 (2017).

[5] Panagiotis Bouros, Shen Ge, and Nikos Mamoulis. 2012. Spatio-textual similarity joins. *PVLDB* 6, 1 (2012), 1–12.

[6] John Cheng, Max Grossman, and Ty McKercher. 2014. *Professional Cuda C Programming*. John Wiley & Sons.

[7] Mateus SH Cruz, Yusuke Kozawa, Toshiyuki Amagasa, and Hiroyuki Kitagawa. 2015. GPU acceleration of set similarity joins. In *International Conference on Database and Expert Systems Applications*. Springer, 384–398.

[8] Dong Deng, Guoliang Li, He Wen, and Jianhua Feng. 2015. An efficient partition based method for exact set similarity joins. *Proceedings of the VLDB Endowment* 9, 4 (2015), 360–371.

[9] Fabian Fier, Nikolaus Augsten, Panagiotis Bouros, Ulf Leser, and Johann-Christoph Freytag. 2018. Set similarity joins on MapReduce: an experimental survey. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1110–1122.

[10] Alec Go, Richa Bhayani, and Lei Huang. 2009. Twitter sentiment classification using distant supervision. *CS224N Project Report, Stanford* 1, 12 (2009).

[11] Oded Green, Robert McColl, and David A Bader. 2012. GPU merge path: a GPU merging algorithm. In *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 331–340.

[12] Oded Green, Pavan Yalamanchili, and Lluís-Miquel Munguía. 2014. Fast triangle counting on the GPU. In *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*. IEEE Press, 1–8.

[13] Yu Jiang, Guoliang Li, Jianhua Feng, and Wen-Syan Li. 2014. String Similarity Joins: An Experimental Evaluation. *PVLDB* 7, 8 (2014), 625–636.

[14] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2017. Billion-scale similarity search with GPUs. *arXiv preprint arXiv:1702.08734* (2017).

[15] David Blair Kirk and Wen-mei W. Hwu. 2013. *Programming Massively Parallel Processors - A Hands-on Approach, 2nd Ed.* Morgan Kaufmann.

[16] Michael D Lieberman, Jagan Sankaranarayanan, and Hanan Samet. 2008. A fast similarity join algorithm using graphics processing units. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*. IEEE, 1111–1120.

[17] Willi Mann and Nikolaus Augsten. 2014. PEL: Position-Enhanced Length Filter for Set Similarity Joins. In *Proceedings of the 26th GI-Workshop Grundlagen von Datenbanken*. 89–94.

[18] Willi Mann, Nikolaus Augsten, and Panagiotis Bouros. 2016. An Empirical Evaluation of Set Similarity Join Techniques. *Proceedings of the VLDB Endowment* 9, 9 (2016), 636–647. http://www.vldb.org/pvldb/vol9/p636-mann.pdf

[19] Ahmed Metwally and Christos Faloutsos. 2012. V-SMART-Join: A Scalable MapReduce Framework for All-Pair Similarity Joins of Multisets and Vectors. *PVLDB* 5, 8 (2012), 704–715.

[20] Rafael David Quirino, Sidney Ribeiro-Junior, Leonardo Andrade Ribeiro, and Wellington Santos Martins. 2017. Efficient Filter-Based Algorithms for Exact Set Similarity Join on GPUs. In *International Conference on Enterprise Information Systems*. Springer, 74–95.

[21] Leonardo Andrade Ribeiro and Theo Härder. 2011. prefix filtering to improve set similarity joins. *Information Systems* 36, 1 (2011), 62 – 78.

[22] Sidney Ribeiro-Junior, Rafael David Quirino, Leonardo Andrade Ribeiro, and Wellington Santos Martins. 2017. Fast parallel set similarity joins on many-core architectures. *Journal of Information and Data Management* 8, 3 (2017), 255.

[23] Akash Das Sarma, Yeye He, and Surajit Chaudhuri. 2014. ClusterJoin: A Similarity Joins Framework using Map-Reduce. *PVLDB* 7, 12 (2014), 1059–1070.

[24] Rares Vernica, Michael J. Carey, and Chen Li. 2010. Efficient parallel set-similarity joins using MapReduce. In *SIGMOD Conference*. 495–506.

[25] Jiannan Wang, Guoliang Li, and Jianhua Feng. 2012. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 85–96.

[26] Xubo Wang, Lu Qin, Xuemin Lin, Ying Zhang, and Lijun Chang. 2017. Leveraging set relations in exact set similarity join. *Proceedings of the VLDB Endowment* 10, 9 (2017), 925–936.

[27] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. 2011. Efficient similarity joins for near-duplicate detection. *ACM Trans. Database Syst.* 36, 3 (2011), 15:1–15:41.

APPENDIX

A CUDA OVERVIEW

In CUDA terminology, CPU and the main memory are referred to as *host*, while the GPU and its own memory are referred to as *device*. In this work, we use the terms CPU and host (resp. GPU and device) interchangeably.

A.1 Architecture.

CUDA-enabled GPUs have many cores called *Streaming Processors (SPs)*, which are divided into groups called *Streaming Multiprocessors (SMs)*. Each SM includes also other units such as ALUs, instruction units, memory caches for load/store operations, and follows the *Single Instruction Multiple Data (SIMD)* parallel processing paradigm.

A.2 Thread Organization.

Threads are organized in logical blocks called *thread blocks*. A thread block is scheduled and executed in its entirety on a SM in groups of 32 threads called *warps*. Threads within a warp are called *lanes* and share the same instruction counter, thus they are executed simultaneously in a SIMD manner. There are two cases of thread divergence, which degrade performance, namely *inter-warp*, when concurrent warps run unevenly and *intra-warp*, when warp lanes take different execution paths. The latter is also simply referred to as *warp divergence*.

A.3 Memory hierarchy.

There are several memory types on CUDA-enabled GPUs. They are divided into on-chip and off-chip ones. Off-chip memories include the *global*, *constant*, *texture* and *local* memory. The global memory is the largest (in the order of GBs) but slowest memory. Data transferred from the host to device resides in global memory and it is visible to all threads. The constant memory is read-only and much smaller (in

the order of KBs). It is used for short access times on immutable data throughout the execution. The texture memory is essentially a read-only global memory and is preferred when 2-dimensional spatial locality occurs in memory access patterns. The local memory is part of the global memory and is used when the registers needed for a thread are fully occupied or cannot hold the required data. This is called *register spilling*. On-chip memories include the *caches*, the *shared* memory and the *registers*. For data reuse, there are caches per SM and the L2 cache is shared across all SMs. The shared memory is the second fastest memory type. Each SM has its own shared memory. Data stored in shared memory can be accessed by all threads within the same block, thus threads of the same block are allowed to inter-communicate via shared-memory. The registers are the fastest memory type and contain the instructions of a single thread and the local variables during the lifetime of that thread.

A.4 Kernel grid.

Every function to be processed in parallel by the GPU is called a *kernel*. Each kernel is executed by multiple thread blocks, which form the kernel *grid*. The grid can be regarded as an array of blocks with up to two dimensions. Each block is, in turn, an array of threads with up to three dimensions. CUDA can schedule blocks to run concurrently on a SM depending on the shared memory and registers used per block. Increasing either of these factors can lead to limited concurrent block execution, which results in low occupancy. *Occupancy* is defined as the ratio of active warps on a SM to the maximum allowed active warps per SM. Maximizing occupancy is a good heuristic approach but it does not always guarantee performance gain. On the contrary, maintaining high warp *execution efficiency*, i.e. the average percentage of active threads in each executed warp, is a more robust approach for data-management tasks, as shown in executing generic theta-joins on GPUs [4].

B IMPLEMENTATION ISSUES

B.1 Host Details

Our framework leverages the work of Mann [18]. The main difference is that we migrate the verification phase from the host to the device side via a multi-threaded implementation, which gives rise to the issues discussed in this section.

B.1.1 Candidate Serialization. The need to transfer candidate pairs to device memory highlights the necessity of efficient serialization methods. Our goal is for the device to avoid complex global memory accesses. Therefore, the host is responsible for storing the candidates of a probe set in successive memory addresses. We list our options for serializing candidates C as follows:

(1) Use a sequence container such as *std::vector* and push back every new candidate. The main drawback is the extra memory checks on insertion to determine if a reallocation is required.
(2) Prior reserve memory space for *std::vector* to avoid memory checks.
(3) Use primitive arrays and handle memory operations manually.
(4) Use a map structure where a key is an integer, i.e. the probe set ID, and its value is a *std::vector* containing the corresponding candidates IDs.

As a complement to C, a separate array C_O to delimit candidate pairs is required. Moreover, the tokens that we insert to C_O are pairs consisting of a probe set ID and its corresponding offset on C. Omitting the probe set ID and inserting the candidate offset by itself, thus reducing the C_O size, implies that the probe set ID should be capable to be *extracted* from the index of C_O. For that to be possible, continuous probe sets IDs in ascending order must be processed, which might not always be the case.

The use of map is an intermediate stage to group together in memory probe set candidates. Before invoking the device, we must iterate the map and serialize every candidates list to construct the final C, and also update C_O per iteration.

As will be shown in our experiments, primitive arrays, i.e., the third option listed above, perform better than *std::vector*, i.e., the first two options, and are adequate for ALL and PPJoin that produce the full candidate set for a single probing set in a single phase. For GRP, which employs two phases during candidate generation, a map structure is necessary if the full verification phase is delegated to the GPU.

B.1.2 Thread & Memory Management. As stated in Section 3, our framework consists of three threads. The indexing/filtering thread H_0 reserves memory space beforehand and serializes candidates. When the device maximum memory for candidate pairs M_c is filled, H_0 triggers H_1, the device handler thread, and assigns a pointer to the current candidates to it. In parallel, H_1 allocates the required device memory space, copies the candidates array to device and launches the join kernel code. Meanwhile, H_0 continues to build the next chunk of candidates. As soon as the join kernel finishes, if an aggregation on top of the join is requested, H_1 launches immediately a separate kernel to perform a count reduction. In case of output the actual pairs, H_1 starts the H_2, which post-process O. When H_2 finishes, the memory reserved for the respective candidates is freed. The same steps are repeated until no candidate pairs are left to be verified.

B.1.3 GroupJoin Work Split. The three best-performing algorithms examined can be divided into two categories: those which generate every candidate pair in one phase, i.e.

ALL and PPJ, and the one, GRP, which requires an extra phase (group expanding) to output all candidate pairs.

For each probe set in ALL and PPJ, candidates are guaranteed to be stored in successive memory addresses. Thus, we serialize candidates using primitive arrays. In contrast, GRP generates candidates that are intertwined due to the expanding phase. Therefore storing candidates in a map structure is required. However, in our testings serializing the map adds extra overhead, rendering this option unfeasible.

We choose to split the work for GRP, and for this technique, allocate part of the verification to the CPU as well. We assign the verification of every candidate pair generated in the first phase to the device. Hence, we serialize candidates from this phase in primitive arrays and transfer them to the device. Every candidate pair that emerges from the second phase, i.e. group expanding, is left to be verified in the host side by H_0.

By splitting the work, we alleviate overall performance as shown in Section 4. In spite of this gain, transferring the whole GRP verification workload to device remains a challenge and would further improve the performance.

B.2 Device Details

B.2.1 Block size. We launch grids composed of 1D blocks. The block size B must be a power of 2 for reduction on the shared memory to work properly. We prefer $B = 32$ since a *warp* can be considered as the CPU thread equivalent. However in our evaluation with different block sizes, we show a correlation between block size and set size, and increasing the block size should be considered when the third verification alternative is the best performing one.

B.2.2 Merge and Intersect Path. Computing a list intersection can be derived from a list merge operation. An efficient parallel merging algorithm for GPUs is Merge Path[11]. Given two sorted lists A and B, Merge Path considers the order in which elements are merged, which is equivalent to the traversal of a grid, noted as *Merge Matrix*, of size $|A| \times |B|$. Beginning from the top left corner of the grid, the path can only move to the right if $A[i] \geq B[j]$ or downwards otherwise, until it eventually reaches the bottom right corner.

There are two partitioning stages, one on kernel grid level and the other on block level. On grid level, equidistant cross diagonals are placed on the Merge Matrix. Using binary search, the point of intersection for a cross diagonal and the path is found. As a result, each SM is assigned to merge non-overlapping portions of the input. On block level, threads cooperate in loading the required list portions on shared memory and then merge them in global memory.

By modifying Merge Path in [12], the authors propose a fast list intersection algorithm, called Intersect Path. They introduce a new diagonal path move, if $A[i] = B[j]$. The same

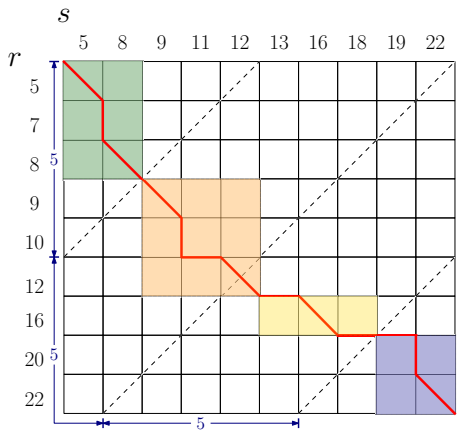

Figure 6: Intersect path example.

partitioning stages still hold. Each SM outputs a portion of the intersection to global memory.

Because in our approach thread blocks verify whole candidate pairs, we have to modify Intersect Path accordingly. Thus we perform both partitioning stages on block level.

B.2.3 Set Intersection Count. In order to verify a candidate pair, threads must calculate the intersection of two sets. Each thread in alternatives A and B, independently performs a merge-like loop and counts the number of intersects. On the other hand, in alternative C, threads collaborate to output the intersection. Since our problem can be reduced to finding the intersection count of two sets, we use a modified Intersect Path algorithm to divide the workload between block threads, as mentioned above.

Cross diagonals are equally placed apart at $\lceil \frac{|r_i|+|s_j|}{B} \rceil$ [12]. Each thread is assigned with a partition with starting point the intersection of the path and the corresponding diagonal. In Figure 6, we show an example of Intersect Path using $B = 4$ threads (each color corresponds to a different thread). Starting from the top left corner, cross diagonals are placed apart 5-hops away, where hops are along the axes. Each thread calculates independently its partition intersection count and stores it in registers memory. When finished, intersection counts are copied in shared memory for a fast reduction to output the overall intersection.

B.2.4 Count Reduction. Our primary focus is to minimize the global memory transactions. Whenever possible, we use the registers to store counts per thread. Similarly, we store every thread count to shared memory and use warp shuffle functions for fast reduction. Hence, when performing an aggregation on top of the join, only a single write to global memory is required per block. To output the final count, we use *thrust::reduce* on the global memory-resident intermediate counts.