Information Systems 108 (2022) 101992

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

Exploiting GPUs for fast intersection of large sets

Christos Bellas [*], Anastasios Gounaris

Department of Informatics, Aristotle University of Thessaloniki, Greece

ARTICLE INFO

Article history:
Received 19 June 2021
Received in revised form 3 December 2021
Accepted 28 January 2022
Available online 31 January 2022
Recommended by Gottfried Vossen

Keywords:
Set-intersection
Set-similarity join
Set-containment join
Matrix multiplication
GPU computing
CUDA

ABSTRACT

The main focus of this work is on large set intersection, which is a pivotal operation in information retrieval, graph analytics and database systems. We aim to experimentally detect under which conditions, using a single graphics processing unit (GPU) is beneficial over CPU techniques and which exact techniques are capable of yielding improvements. We cover and adapt techniques initially proposed for graph analytics and matrix multiplication, while we investigate new hybrids for completeness. We also explain how we can address set containment joins using the same techniques. The comprehensive evaluation highlights the main characteristics of the techniques examined when both a single pair of two large sets are processed and all pairs in a dataset are examined, while it provides strong evidence that state-of-the-art set containment stands to significantly benefit from advances in GPU-enabled set intersection. Our results reveal that there is no dominant solution but depending on the exact problem and the dataset characteristics, different techniques are the most efficient ones.

1. Introduction

Set intersection is an essential operation in numerous application domains, such as information retrieval for text search engines using an inverted index [1,2], graph analytics for triangle counting and community detection [3–5], and database systems for merging RID-lists [6] and performing fast (potentially bitwise) operations on data in columnar format [7].

Modern GPUs offer a high-level parallel environment at low cost. As a result, over the past years, there has been a considerable research work on improving graph analytics on a GPU, mostly in the context of graph triangle counting, where set intersection dominates the running time [8–13]. The majority of these studies focus on improving the level of parallelism by reducing redundant comparisons and distributing the workload evenly among GPU threads. Set intersection on GPUs has also been examined in the context of set similarity joins [14].

In this work, we compare and evaluate state-of-the-art GPU techniques for set intersection, when the sets are large, i.e., they may contain up to millions of elements. The main rationale behind our high-level approach is to capitalize on existing techniques to the largest possible extent. Therefore, we aim to transfer existing knowledge and this exploitation of existing solutions takes place at two distinct and complementary levels: (i) We transfer GPU-oriented solutions for small set intersection to a setting that sets are large; we also leverage the reduction of the set intersection problem to the more generic problem of matrix multiplication. (ii) We transfer set intersection solutions to solve the set containment problem in both its simpler binary form and in a more generic form, where the degree of containment is reported for every pair of sets checked. More specifically, we gather together techniques belonging to six different rationales, namely intersect path, optimized binary search, hash, bitmap and set similarity join-based solutions, and matrix multiplication, while we include novel promising hybrid solutions that combine existing techniques to better fit into our setting. Moreover, we show that our techniques are not only sufficient to address the set containment problem but they are also superior to the current state-of-the-art as reported in the literature. However, a key observation is that there is no dominant solution and a main contribution of this work is to define under which conditions a specific technique should be employed. Our results complement earlier results, e.g., in [15], which show that a specific technique, such as binary search-based one is a dominant solution in a specific case; in our work, we provide evidence as to which technique to choose in different scenarios after examining a broader set of alternatives and our suggestions are not always aligned with the ones in [15].

We perform experiments both when the intersection of one pair of sets (single-instance problem) and the intersections among all set pairs in a database are computed (multi-instance problem). We derive useful insights, and more specifically we provide experimental evidence about the superiority of (i) two intersection path flavors for the single-instance problem and (ii) bitmap-based or similarity join-based or matrix multiplication-based solutions for the multi-instance problem depending on the size of the sets

[*] Corresponding author.
E-mail addresses: chribell@csd.auth.gr (C. Bellas), gounaria@csd.auth.gr (A. Gounaris).

https://doi.org/10.1016/j.is.2022.101992
0306-4379/© 2022 Elsevier Ltd. All rights reserved.

and the size of the element universe. We also discuss the cases, where CPU–GPU co-processing is beneficial.[1]

A preliminary short version of this work has appeared in [16]; the main points of extensions hereby include the consideration and evaluation of (i) matrix multiplication techniques and (ii) co-processing-based set containment joins. Moreover, we perform main parts of the evaluation from scratch comparing GPU-based techniques against parallel CPU counterparts in order to show the benefits of using a graphics card in a clearer and fair manner. The benefits of using GPU variants are at the order of magnitude.

The rest of the manuscript is organized as follows. Section 2 gives a background on the problem and an overview of the related work. Section 3 describes and evaluates the GPU techniques for set intersection. In Section 4, we present the technical details and the experiments for set containment. Last, in Section 5 we conclude and discuss possible future work.

2. Preliminaries

In this section, we first give a formal notation for the set intersection and the set containment problems. Next, we explain the exact scope of this work and we continue with an overview of the related work about set intersection and set containment.

2.1. Set intersection problem description

Single-instance set intersection (SISI) problem: given (i) a finite universe of elements E, and (ii) a set $A = \{e_1^A, \dots, e_n^A\}$ of size n and a set $B = \{e_1^B, \dots, e_m^B\}$ of size m, where $e_i^{\{A,B\}} \in E$, set intersection $A \cap B$ produces a new set S containing all the common elements among A and B.

Multi-instance set intersection (MISI) problem: in the multi-instance set intersection problem, we are given a collection of k sets $C = \{S_1, \dots, S_k\}$, and we aim to find each individual set intersection among all $\binom{k}{2}$ of pairs (S_i, S_j), where $0 < i < j \leq k$.

Given the definitions above, the solutions of SISI are of $\Omega(n+m)$ time complexity, whereas MISI solutions are of quadratic complexity in k without the problem definition leaving any space for pair pruning, as, for example, in problems such as set similarity joins [14]. Therefore, the focus is not on evaluating the behavior of algorithms differing in asymptotic complexity, since all techniques are of similar complexity; rather, we aim to assess the impact of different potentially low-level engineering techniques, especially when n and m are at the order of millions and the size of E is orders of magnitude larger. Space complexity differs between techniques; some of them may require additional data structures to operate.

2.2. Set containment problem description

A closely related problem to set intersection is set containment. Essentially, given two sets A and B, set containment indicates if one set is contained within the other, i.e. that one set is a subset of the other. Set containment can be tackled either by union-based methods or intersection-based methods. Since in this work we deal with set intersection techniques, we define the set containment problems in a manner that is consistent with the aforementioned set intersection problems' rationale, as follows.

Single-instance set containment (SISC) problem: given (i) a finite universe of elements E, and (ii) a set $A = \{e_1^A, \dots, e_n^A\}$ of size n and a set $B = \{e_1^B, \dots, e_m^B\}$ of size m, where $e_i^{\{A,B\}} \in E$, set containment holds true if $|A \cap B| = min(n, m)$.

Multi-instance set containment (MISC) problem: this is also known as set containment join and is defined in manner similar to MISI. In the multi-instance set containment problem, we are given a collection of k sets $C = \{S_1, \dots, S_k\}$, and we aim to find, for each set pair among all $\binom{k}{2}$ of pairs (S_i, S_j), where $0 < i < j \leq k$, if the set containment condition holds true.

We make two remarks based on the definitions above. Firstly, contrary to other common MISC definitions of set containment join, e.g., in [17], we do not restrict set containment to refer only to a setting where the contained sets need to belong to a specific collection only, but we always check whether the smaller set is contained into the larger one. The case where the two sets are of equal size is also covered: if, in that case, one set is contained into the other, this implies that the two sets are the same. Secondly, it is trivial to extend the set containment result not to be binary but, instead, to correspond to the *degree of containment* of the smaller set into the larger one: $|A \cap B|/min(n, m) \in [0, 1]$.

All techniques solving the SISI (resp. MISI) problems, can be used to solve the SISC (resp. MISC) problems as well; however the reverse holds only if SISC/MISC solutions compute the degree of containment, i.e., not just establishing whether one set is fully contained in the other one.

2.3. Scope of our work

Over the past years, there has been a lot of research that encapsulates GPU techniques for the set intersection problem, as will be discussed shortly. In the majority of cases, the problem of set intersection is tackled in the context of triangle counting to accelerate graph analytics. Triangle counting is a special case of set intersection, which stems from the need to find quickly intersection counts among vertex adjacency lists of small lengths. As a result, the techniques proposed for triangle counting are tailored to specific algorithmic optimizations. On the other hand, to the best of our knowledge, there is no existing work that investigates set containment on a single GPU either directly or indirectly. In the context of this work, our motivation is twofold. Firstly, we extract, adapt and evaluate set intersection techniques under a more demanding large set intersection scenario performing also a comparison against all methodologies proposed that can address the SISI and MISI problems for large sets. Secondly, we build on top of the evaluated set intersection techniques to conduct set containment in a single GPU setting, and compare our resulting techniques to the state-of-the-art CPU set containment solutions for the MISC problem.

2.4. Overview of existing solutions

In this section, we discuss details of existing set intersection and set containment techniques separately. The former refer to a GPGPU setting, while the latter assumes a traditional CPU-only environment, since to date, set containment has not been examined using GPUs.

2.4.1. Set intersection

We present the relevant techniques mostly in chronological order. Ding et al. [18] were the first to implement a parallel GPU set intersection algorithm. In their proposed solution, they use the so-called *Parallel Merge Find (PMF)* algorithm to compute a set intersection. In essence, given two sets, the shorter one is partitioned into disjoint segments, with each segment assigned to a different GPU thread. Then, the last element of each segment is searched in the longer set to find the corresponding or closest positions for the partitioning of the longer set. As a result, each GPU thread becomes capable of computing its own intersection among segments in parallel. In [19], Wu et al. highlight the inefficiency of the approach followed in [18] for sets of smaller size. Subsequently, they propose a GPU technique for set intersection

[1] The source code is available at https://github.com/chribell/set_intersection.

that takes advantage of the fast on-chip shared memory. First, an empty array of size equal to the size of the shorter set is allocated in shared memory. Next, each GPU thread is assigned a specific number of elements from the shorter set and conducts binary searches in the longer set. If an element is found, its corresponding cell in the shared memory array is set to 1. Finally, a scan operation on the shared memory array is performed along with a stream compaction procedure, so that threads produce the final intersection. In [20], Wu et al. extend their original work described above by introducing heuristic strategies to balance the workload across thread blocks more efficiently. Additionally, the proposal in [21] further extends the original work in [20] by introducing a linear regression approach to reducing the search range of a binary search and a hash segmentation approach as an alternative to binary search.

In [8], Green et al. present the *Intersect Path (IP)* algorithm for set intersection, which is a variation of the well established GPU *Merge Path* algorithm for merging lists. In addition, *IP* can be considered as an extension of *PMF* since it encapsulates a similar set partition logic. First, input sets are partitioned into segments that can be intersected independently by multiple thread blocks. Second, for each thread block, workload is distributed in such a way that near equal number of elements to intersect are allocated to threads. In [11], the authors extend the work of [8] by proposing an adaptive load balancing technique that dynamically assigns work to GPU threads based on work estimations. The authors of [22] and [9] follow a similar approach to set intersection. More specifically, their main concept is, for each GPU thread, to sequentially compute a set intersection by using a two-pointers merge algorithm. In the context of triangle counting, such an approach is applicable since the requirement is (i) for each GPU thread to compute a two-set intersection count independently, and (ii) these partial counts to be accumulated to compute the final global count. However, in the setting of set intersection over two large sets, their solution is inferior to GPU-based competitors and similar to a sequential CPU approach.

In [10], Bisson et al. propose a different approach, namely, GPU set intersection to be based on bitmaps and atomic operations. Given two sets, to compute their intersection, the GPU threads create the bitmap representation of the first set in parallel and then, iterate over the elements of the second set to search for the corresponding set bits. Based on the average set size, the workload allocation can be per thread, per warp or per block.

In [12], Hu et al. demonstrate that set intersection is faster though employing efficient binary search-based techniques than *IP*-based techniques, arguing that the latter suffers from non-trivial overhead of partitioning the input sets and non-coalesced memory accesses. On the other hand, their proposed algorithm optimizes binary search at a warp level to achieve coalesced memory access and to alleviate the need for workload balancing. In addition, by caching the first levels of the binary search tree they employ in the shared memory, they can achieve further speedup gains.

In [13], Pandey et al. propose a hash-based technique for set intersection. In brief, first, their algorithm hashes the shorter set into buckets, and then iterates over the larger set and hashes each element to the corresponding bucket. Afterwards, a linear search is conducted within each bucket to find the intersections.

Last, in the context of set similarity join, the authors of [23] propose a technique for conducting set intersections by using a static inverted index and atomic operations. By setting the similarity degree equal to zero, their solution can be used to solve the SISI/SISC and MISI/MISC problems.

As already mentioned, some of these works, such as [8–10,12, 22] are more complete proposals targeting the triangle counting problem; here we narrow down our attention only to the part that is relevant to the SISI and MISI problems.

2.4.2. Set containment

There are several solutions proposed for set containment in the literature. In [24], the authors, based on the set operators that they employ, classify existing solutions into two categories, namely (i) intersection-oriented solutions [17,25–28], and (ii) union-oriented solutions [29,30]. Since no prior GPU solution for set containment exists, we give a brief overview of recent CPU solutions.

Intersection-oriented solutions. The core concept of intersection-oriented solutions is to first build an inverted index on the input collection of sets S, and then, for each set $s \in S$, intersect all the inverted lists corresponding to the elements of s. The main advantage of intersection-oriented solutions is that they are verification-free. However, the main drawback is the dominant cost of the intersection of the inverted lists, especially in the case of long inverted lists. Hence, every intersection-oriented solution proposed aims to decrease this cost.

In [25], the authors propose a solution called *PRETTI*, which employs a prefix tree structure to collectively process input sets. The use of the prefix tree reduces the number of intersections of inverted lists among sets with the same prefix, thus amortizing the computational cost. In [31], the authors extend *PRETTI* by replacing the prefix tree with a more compact tree structure, called Patricia tree, in which all the nodes along a single path are merged into a single node. As a result, the total number of tree nodes traversed is decreased, improving the overall performance.

In [26], Bouros et al. propose a different solution, closer to set similarity join works, called *LIMIT*. Furthermore, *LIMIT* initially builds a shorter prefix tree of height h, considering only the first h elements of each set. After, the set containment join is conducted by using a filtering-verification framework. In brief, in the filtering phase, *LIMIT* either outputs set pairs directly if they share a common prefix and the set size is less or equal to h, or generates candidate pairs that are fully evaluated in the verification phase.

In [27], Kunkel et al. propose a solution, called *PIEJOIN*, in which they transform the prefix tree into linear arrays by using preorder ranks and preorder intervals. In addition, by enriching each tree node with the interval information, they can deduce efficiently whether a certain set element is contained in the corresponding subtree, without traversing it. The authors also introduce a parallel version of *PIEJOIN* using a shared-memory model. In the parallel version, the linearized prefix tree is partitioned based on the order in which tree nodes are stored. In their evaluation, they show that their parallelization efficiency depends heavily on input dataset characteristics.

In [17], Deng et al. propose a solution, called *LCJOIN*, in which they demonstrate a novel method to intersect all inverted lists simultaneously while traversing a prefix tree, and thus reducing the computational cost significantly. Furthermore, they introduce a data partitioning method in which they determine whether partitions should use the complete inverted index or construct a local index on-the-fly. In their evaluation, the authors show that *LCJOIN* outperforms most of the state-of-the-art set containment join solutions[2]; in this work, we directly compare our solutions against an implementation of *LCJOIN*.

Last, in [28], Deep et al. investigate the use of matrix multiplication to accelerate the set containment join problem. Their solution, called *MMJoin*, first employs a data partitioning scheme in which input sets are classified as light or heavy ones based on their size. Similarly, set elements are also classified into light or heavy ones based on the inverted list sizes. Next, light and heavy sets with light set elements are directly processed using the inverted index, whereas heavy sets with heavy set elements

[2] There is no published comparison between [17] and [28].

are processed using matrix multiplication. In their evaluation, the authors show that matrix multiplication is highly conducive to achieving higher performance on dense datasets. As will be discussed in the next section, we leverage this approach motivated also by the fact that matrix multiplication naturally lends itself to efficient parallelization on a GPU. In addition, we also apply the same concept to address set intersection.

Union-oriented solutions. The main rationale of union-oriented solutions is to generate signatures alongside the inverted index construction on the input collection of sets S. Then, for each set $s \in S$, the corresponding techniques generate candidate sets for evaluation by finding the union of the inverted lists for the relevant signatures. Finally, candidate set pairs undergo full verification to form the final output. The two dominant costs of union-oriented solutions are the cost of the union of inverted lists and the cost of the verification phase. As shown in [26] and [29], union-oriented solutions [32,33] are outperformed by intersection-oriented solutions due to these costs. However, progress on improving union-oriented solutions, by integrating perks from intersection-oriented solutions, has been made recently [29,30].

3. Techniques for set intersection

In the previous section, we have identified six different methodologies, based on (1) *IP*, (2) optimized binary search, (3) bitmap operations, (4) hashing, (5) set similarity joins and (6) matrix multiplication, respectively. Here, we start with presenting five different state-of-the-art GPU techniques for set intersection, one for each of the first five methodologies, in more detail. In addition, we discuss some modifications to these techniques to target large set intersection yielding two novel hybrid solutions. We do not consider works such as [9,22] that mostly rely on the preprocessing of input data and conduct intersection with a simple merge fashion algorithm. By contrast, we present and evaluate techniques that are more adaptable in a general set intersection scenario. Next, we explain how we can leverage matrix multiplication for set intersection and we emphasize the memory constraints imposed. Overall, we give a concise presentation for each evaluated technique below followed by extensive experiments and discussion on the characteristics of the techniques based on the results.

3.1. GPU implementations of the different methodologies

3.1.1. Intersect Path (IP) [8]

Given two ordered sets A and B, *IP* considers the traversal of a grid, noted as *Intersect Matrix*, of size $|A| \times |B|$. Beginning from the top left corner of the grid, the path can move downwards if $A[i] > B[j]$, to the right if $A[i] < B[j]$, and diagonally if $A[i] = B[j]$, until it eventually reaches the bottom right corner. There are two partitioning stages, one on the kernel grid level and the other on the block level. On the grid level, equidistant cross diagonals are placed on the Intersect Matrix. The number of diagonals is equal to the number of thread blocks plus one, in order to delimit the boundaries of each block. Using binary search, the point of intersection between a cross diagonal and the path is found. As a result, each thread block is assigned to intersect disjoint subsets of the input. In case a cross diagonal intersects with the path inside a matrix cell, a count augmentation is required beforehand, since otherwise, this intersection would not be assigned to any thread block. An example of this scenario is shown in Fig. 1. On the block level, the same cross diagonal approach applies in order to distribute workload among threads. Each thread may conduct a serial or binary search-based intersection.

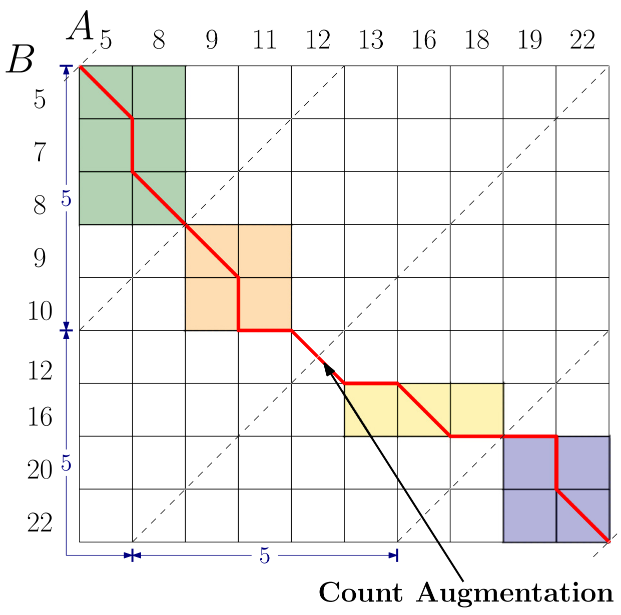

Fig. 1. Intersect Path example using 4 thread blocks.

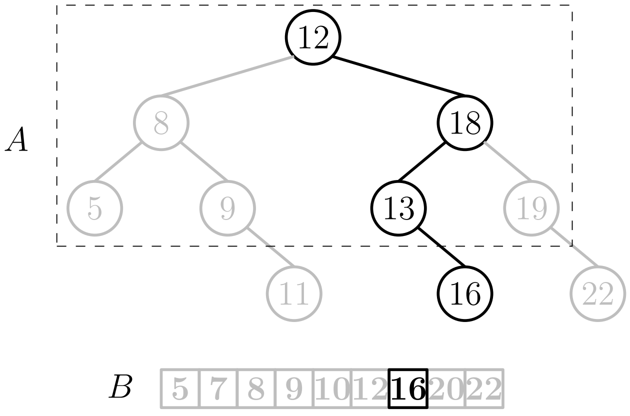

Fig. 2. Optimized Binary Search example.

3.1.2. Optimized Binary Search (OBS) [12]

Given two ordered sets A and B, *OBS* caches the first levels of the binary search tree of the larger set in shared memory to reduce expensive global memory reads. For example, as shown in Fig. 2, the higher level nodes of the binary tree reside in shared memory, whereas the leafs are located in global memory. The smaller set, which is B in the example, is used for lookup and, as a result, there are $|B|$ total binary search lookups. Each thread is assigned a lookup and, in each iteration, up to 32 (i.e., the size of warp) lookups are executed simultaneously. For the multi-instance problem, a simple optimization is to cache each set to shared memory one at a time, and then iterate over every subsequent set to perform the intersection for each pair. This requires sets to be sorted by their size.

3.1.3. Hash-based Intersection (HI) [13]

Given two sets A and B, *HI* first hashes the shorter set and constructs buckets in parallel, and then, iterates and hashes every element of the larger set into the corresponding bucket as already described in Section 2. In case of a collision, a linear search is conducted to find whether an entry with equal value exists in the bucket. An example of *HI* is shown in Fig. 3. The initial hashing of the smaller set is preferred in order to reduce the number

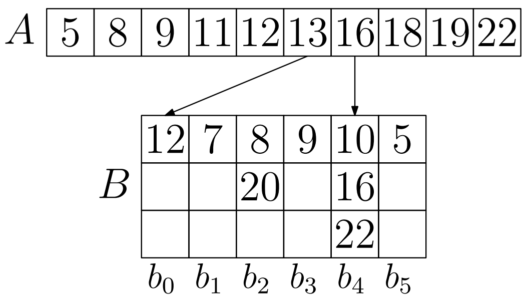

Fig. 3. Hash-based Intersection with six buckets of size three each and $h(x) = x\%6$.

of collisions. Buckets are statically allocated once in the global memory space. The size of each bucket, i.e. the number of entries from the shorter set in the bucket, is stored in shared memory. Thus, to ensure correctness for the MISI problem, we only need to clear the buckets sizes in shared memory, and let every next short set hashing to overwrite the previous one.

3.1.4. Bitmap-based Intersection (BI)

Given two sets A and B, *BI* conducts the set intersection on their bitmap representations, with each bitmap requiring $|E|/8$ bytes of memory regardless of the set sizes. More specifically, there are two flavors of *BI*, namely (i) naive, where all bitmaps fit in global memory, and (ii) dynamic, where bitmaps are built on the fly in the case that we cannot store all of them in the available global memory. The latter is similar to the work of [10]. Bitmaps are constructed in parallel using the `atomicOr` function. In the naive scenario, each GPU thread fetches two bitmap words, one from each set, and conducts a `popcount` operation on the resulting logical `AND` word to compute the subset intersection. An example of the naive scenario is shown in Fig. 4. In the dynamic scenario, each GPU thread block first constructs the bitmap representation of the current set, and then, for every next set, the threads iterate over its elements and check whether the respective bits are set (see the example in Fig. 5).

3.1.5. Static-index Intersection (sf-gssjoin) [23]

Given k sets, *sf-gssjoin* first constructs a static inverted index over all set elements. For each set, the inverted lists for all its elements are concatenated in a logical vector. Next, this vector is partitioned evenly with each partition assigned to a specific GPU thread. Thus, each thread processes independently its corresponding partition and contributes to the intersections of the current set in cooperation with other threads. To ensure correctness, threads increment the intersection counts by using atomic operations. An example of *sf-gssjoin* is shown in Fig. 6.

3.1.6. Discussion and hybrids

All of the presented techniques with the exception of *BI-naive* and *IP*, conduct a single instance set intersection on a single block or warp. This results in severe GPU underutilization, especially for sets of size in the order of millions, since a single execution unit is assigned with the complete workload. To tackle this issue, we investigate the integration of the kernel grid level partitioning of *IP* with *OBS* and *HI*, which have not been proposed previously in the literature. We denote these novel hybrid techniques as *IP-OBS* and *IP-HI* respectively.

In addition, for the single instance scenario, in *BI-dynamic*, we modify the algorithm by constructing the bitmap of the first set across multiple blocks and then we partition evenly the second set into disjoint subsets with each one assigned to a specific block.

Finally, a common problem encountered in GPGPU computing is the need to process data that do not fit into GPU memory. In the setting of MISI, we need to compute and store $\binom{k}{2}$ intersections. In addition to the space required to store the intersections, there is also the prerequisite of sufficient memory space so that each technique can store its auxiliary data structures. Therefore, it becomes apparent that for large values of k the complete result cannot be computed in a single GPU pass. In such cases, the simplest approach to process the multi-instance scenario is to partition the output into tiles and conduct set intersection incrementally by invoking the GPU for each tile. An illustration of this approach is depicted in Fig. 7.

3.2. Leveraging matrix multiplication for set intersection

A different approach to evaluating set intersection involves the use of matrix multiplication. Essentially, by representing a collection of k-sets S as a binary $k \times |E|$ matrix M_S, we can compute the intersections for the MISI problem, by multiplying M_S with its transpose matrix M_S^T. An example of using matrix multiplication for the MISI problem with $|S| = 5$ and $|E| = 6$ is shown in Fig. 8.

In general, multiplying two matrices of size $n \times n$ naively, requires $O(n^3)$ time. There has been a plethora of research on fast matrix multiplication. Due to their massive parallelism, assisted by the simplicity of the numerical computations of the problem, GPUs excel at matrix multiplication, achieving significant speedups over CPU implementations [34].

However, in the context of MISI, certain restrictions must be met in order to be able to conduct set intersection as a matrix multiplication operation efficiently. The most prominent concerns the required $O(k|E|)$ memory to represent S as a binary matrix M_S. For example, consider a set collection S with $k = 10^3$ and $|E| = 10^7$. The memory required to store M_S as a matrix of floats is approximately 40 GB which exceeds greatly any GPU memory. In addition, the required $O(k^2)$ memory to store the output must also be taken into account, which imposes further restrictions on the datasets that matrix multiplication-based set intersection can process on a GPU. The latter restriction is addressed through invoking multiple kernel calls, as explained previously. Orthogonally to multiple kernel calls, we could split the binary matrix into sub-matrices in case that it does not fit into the main memory; however the memory allocation and clearing overheads associated outweigh any benefits, compared also to other solutions, and we do not employ such a solution in practice. We further discuss this in the subsequent evaluation section.

The main consequence of this approach is that matrix multiplication ceases to be applicable in datasets with very large element universe size. Overall, the time complexity for matrix multiplication (*MM*) in the MISI scenario is $O(k^2|E|)$, whereas the required memory is $O(k^2 + k|E|)$. Hence, the factors of k and $|E|$ dictate if we can employ *MM* for set intersection in a single run, whereas, if M_S can fit into the GPU's main memory, large k values result in multiple kernel calls.

A final note is that, in our implementations, since we operate on binary matrices, we use the SGEMM operation provided by cuBLAS library [35] after performing all adaptations required to incorporate it into our framework.

3.3. Evaluation

In this section, we evaluate the techniques presented in the previous sections, while, in our experiments, we also include CPU variants for completeness. More specifically, we compare the GPU techniques against three CPU alternatives, namely, (i) *SIMD*, which performs intersection on sorted integers using SIMD

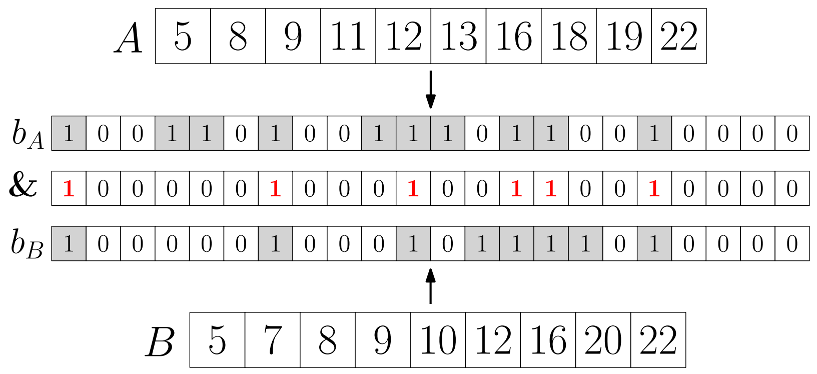

Fig. 4. Naive Bitmap-based Intersection example.

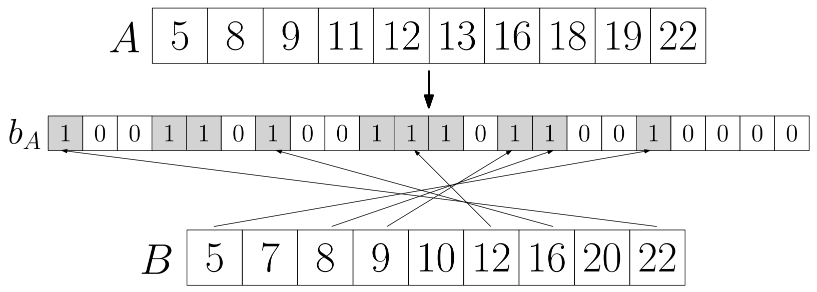

Fig. 5. Dynamic Bitmap-based Intersection example.

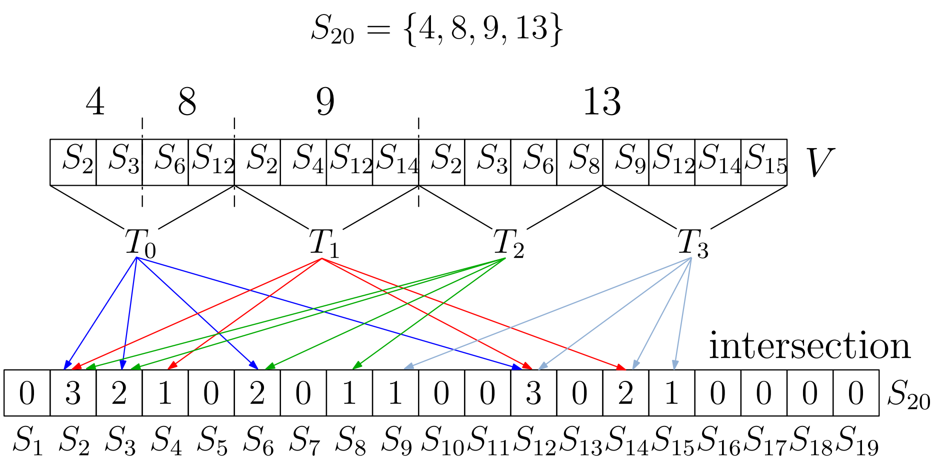
$S_{20} = \{4, 8, 9, 13\}$
4
8
9
13
S_2 S_3 S_6 S_{12} S_2 S_4 S_{12} S_{14} S_2 S_3 S_6 S_8 S_9 S_{12} S_{14} S_{15}
V
T_0
T_1
T_2
T_3
intersection
0 3 2 1 0 2 0 1 1 0 0 3 0 2 1 0 0 0 0
S_{20}
S_1 S_2 S_3 S_4 S_5 S_6 S_7 S_8 S_9 S_{10} S_{11} S_{12} S_{13} S_{14} S_{15} S_{16} S_{17} S_{18} S_{19}

Fig. 6. Static-index Intersection example using 4 GPU threads.
Source: Adapted from [14]

instructions [36], (ii) *std::set_intersection*, which uses the C++ standard library and (iii) *boost::dynamic_bitset* which uses the Boost library in a similar fashion as *BI-dynamic*. In addition, for the multi-instance problem, we run the CPU solutions in parallel by using OpenMP with 12 threads.[3] We measure the total intersection time for the CPU techniques using the *std::chrono* library. For the GPU operations, we measure the total time, including transfers and memory allocations, by using the CUDA event API. We do not measure any preprocessing time since it is the same for all the evaluated techniques. Each number presented is the average of 3 runs; the standard deviation across runs was negligible, therefore there was no need to run more than 3 times each experiment.

The experiments were conducted on a machine with an Intel i7 5820k clocked at 3.3 GHz, 72 GB RAM at 2400 MHz and an NVIDIA Titan XP on CUDA 11.0. This GPU has 30 Streaming Multiprocessors, with a total of 3840 cores, 12 GB of global memory and a 384-bit memory bus width.

We first experiment with artificial datasets, where set elements follow the normal, uniform and zipf like distribution. To distinguish each dataset, we use the notation *Distribution-Element universe-Average set size*. We vary the element universe from 10^8 up to 10^9. Respectively, we vary the average set size from 10^6 up to 10^7. For the multi-instance scenario, we also experiment with four real world datasets previously employed in [14] and one from [37]. More specifically, for each sorted dataset we extract the $k \in \{10\,000, 50\,000, 100\,000\}$ largest sets. Table 1 gives an overview of the real-world datasets' characteristics.

3.3.1. SISI experiments

We examine the single instance scenario using twelve artificial datasets (combinations of 3 distributions, 2 element universe

[3] Due to employing CPU multithreading, all MISI experiments presented hereby differ from those reported in [16].

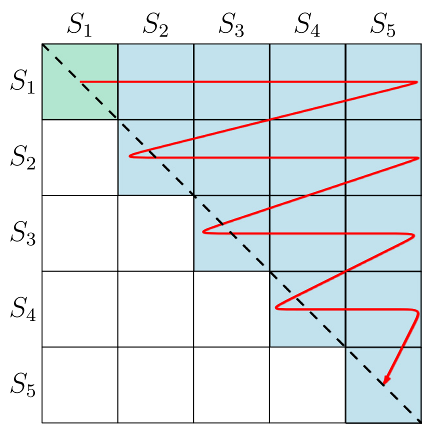

Fig. 7. Process the multi-instance scenario incrementally by using a tile-based approach.

Table 1
Real world dataset characteristics.

Dataset	Cardinality	Avg set size	Element universe
BMS-10K	$1 \cdot 10^4$	35	
BMS-50K	$5 \cdot 10^4$	22	$1.6 \cdot 10^3$
BMS-100K	$1 \cdot 10^5$	17	
ENRON-10K	$1 \cdot 10^4$	794	
ENRON-50K	$5 \cdot 10^4$	374	$1.1 \cdot 10^6$
ENRON-100K	$1 \cdot 10^5$	258	
ORKUT-10K	$1 \cdot 10^4$	1001	
ORKUT-50K	$5 \cdot 10^4$	984	$8.7 \cdot 10^6$
ORKUT-100K	$1 \cdot 10^5$	905	
TWITTER-10K	$1 \cdot 10^4$	150	
TWITTER-50K	$5 \cdot 10^4$	142	$3.7 \cdot 10^4$
TWITTER-100K	$1 \cdot 10^5$	140	
WORDS-10K	$1 \cdot 10^4$	3494	
WORDS-50K	$5 \cdot 10^4$	2396	$1.1 \cdot 10^4$
WORDS-100K	$1 \cdot 10^5$	2021	

sizes and 2 set sizes). We have also experimented with several block sizes and present the results of the best configuration for each technique.

As shown in Fig. 9, *IP* and *IP-OBS* are the clear winners for the SISI experiments, in the sense that one of them is the best overall performing technique across every dataset. Moreover, the differences between their performance is relatively small: up to 12% for the normal and zipf distribution, and up to 35% for the uniform distribution. Overall, these techniques achieve average 2.3X speedup compared to the best performing CPU technique; the best performing CPU technique differs between the cases, but in average, *SIMD* is the most robust CPU technique. In general, the speedup is similar for all distributions types and increases with higher universe and average set size, reaching 2.92X. The speedup of *IP* or *IP-OBS* over the worst performing CPU or GPU technique exceeds an order of magnitude; we have observed speedups up to 48.67X over a CPU technique and up to 10.97X over a GPU technique.

In addition, *IP-HI* exhibits the worst performance regarding GPU techniques and seems unable to perform better than CPU for many settings; this shows that simply relying on the *IP* workload allocation rationale is not adequate. Finally, *BI*-based techniques behave better for smaller universe sizes and non-very sparse bitmap representations; as the sparsity in the bitmap representation increases, solutions like *IP* or *IP-OBS* that operate directly on set elements instead of bits are dominant by a large margin.

We have conducted extensive profiling using the official tool provided by NVIDIA, `nvprof`, to explain the observations above.[4] The higher performance of *IP-OBS* especially when compared to *BI* is mainly attributed to the higher significance of IPC, which stands for Instructions per cycle, (and related metrics) in our problem as shown in Fig. 10. Our problem is not arithmetic operation-intensive but involves several complex interdependencies between computations and accesses to memory spaces; i.e., as a GPU program, it is mostly latency-bounded and as such, traditional efficiency metrics like flops do not apply. *BI*-based solutions have a larger memory footprint because they create the bitmap signatures for both sets, which incurs an overhead. However, IPC alone cannot explain the similar performance between *IP* and *IP-OBS*. Fig. 11 reveals that both these techniques excel at atomic transactions per request. According to the documentation,[5] the optimal target ranges between 1–2 transactions per request and this is achieved by these two techniques only. Finally, *IP-HI* suffers, except the higher number of atomic transactions per request, from a lot of bank conflicts in shared memory and minor register spilling due to collisions and increased number of random memory accesses in larger sets; this explains that it behaves worse than the other *IP*-based variants.

The main lesson learnt is that crafting efficient GPU techniques for the SISI problem is a complex task that requires deep understanding of the interplay between all latency aspects in GPU programming. Counter-intuitively, commonly used metrics may be misleading. For example, *IP-OBS* has a much higher number of instructions issued, but these instructions manage to execute fast; similarly, metrics such as memory throughput and utilization metrics are insufficient to explain the behavior of our techniques. Overall, what matters most is finding a beneficial equilibrium between the amount and type of computations and memory-related bottlenecks that these computations may trigger.

3.3.2. MISI experiments

For the multi-instance evaluation, we conduct two sets of experiments. In the first one, we use two artificially generated datasets consisting of large sets that follow the zipf distribution, which is the closest to real world. In the second one, we use real world datasets and evaluate the intersection techniques on smaller set sizes.

We first discuss the behavior using the two artificial datasets. As shown in Fig. 12, in both datasets examined with $k = 1000$, *BI-dynamic* is the best performing technique and achieves average 10X speedup compared to the best performing CPU technique. Furthermore, *sf-gssjoin* is the second best performing technique, when manages to launch, i.e. when the static index fits in the global memory, which is not the case when the element universe is 10^9 in our experiments. On the other hand, *IP-OBS* is more robust and its performance is the closest to *BI-dynamic* across both datasets. In addition, we observe a 80% performance increase for the hybrid *IP-OBS* technique over the standalone *OBS*. Furthermore, even though *IP* and *IP-HI* are the worst performing GPU techniques, they manage to achieve 1.57X speedup compared to the best performing CPU technique on average. Last, we also note the inability to launch *MM* due to the large element universe, which requires an excessive amount of memory; in such a case, if we were to perform matrix multiplication through splitting the binary matrix into sub-matrices, the associated overhead of allocating GPU memory is several orders of magnitude higher than the slowest CPU technique.

In Fig. 13 we present some additional profiling information referring to the right part of Fig. 12. We can observe that the

[4] The full profiling data are provided along with the source code.
[5] https://docs.nvidia.com/cuda/profiler-users-guide/index.html

$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline s_1 & 0 & 0 & 1 & 1 & 0 & 0 \\ s_2 & 1 & 0 & 1 & 0 & 1 & 0 \\ s_3 & 0 & 1 & 0 & 0 & 1 & 1 \\ s_4 & 0 & 0 & 0 & 1 & 1 & 1 \\ s_5 & 1 & 0 & 1 & 1 & 0 & 1 \end{array} \times \begin{array}{c|ccccc} & s_1 & s_2 & s_3 & s_4 & s_5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 0 & 0 & 1 & 0 & 0 \\ 3 & 1 & 1 & 0 & 0 & 1 \\ 4 & 1 & 0 & 0 & 1 & 1 \\ 5 & 0 & 1 & 1 & 1 & 0 \\ 6 & 0 & 0 & 1 & 1 & 1 \end{array} = \begin{array}{c|ccccc} & s_1 & s_2 & s_3 & s_4 & s_5 \\ \hline s_1 & 2 & 1 & 0 & 1 & 2 \\ s_2 & 1 & 3 & 1 & 1 & 2 \\ s_3 & 0 & 1 & 3 & 2 & 1 \\ s_4 & 1 & 1 & 2 & 3 & 2 \\ s_5 & 2 & 2 & 1 & 2 & 4 \end{array}$$

Fig. 8. Using matrix multiplication for set intersection.

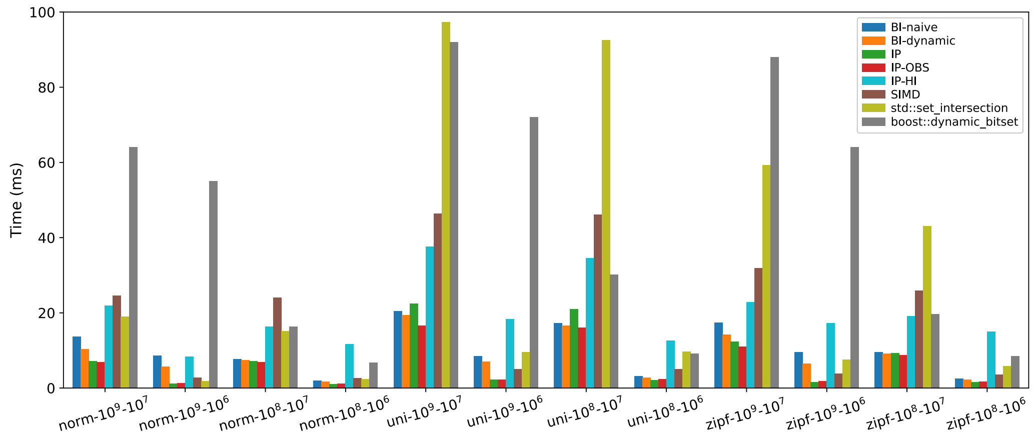

Fig. 9. Single-instance intersection experiments on artificial datasets.

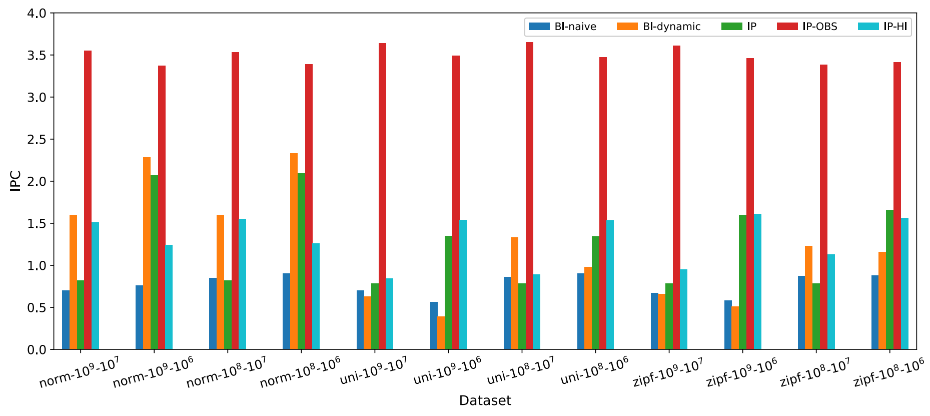

Fig. 10. SISI IPC profiling (higher is better).

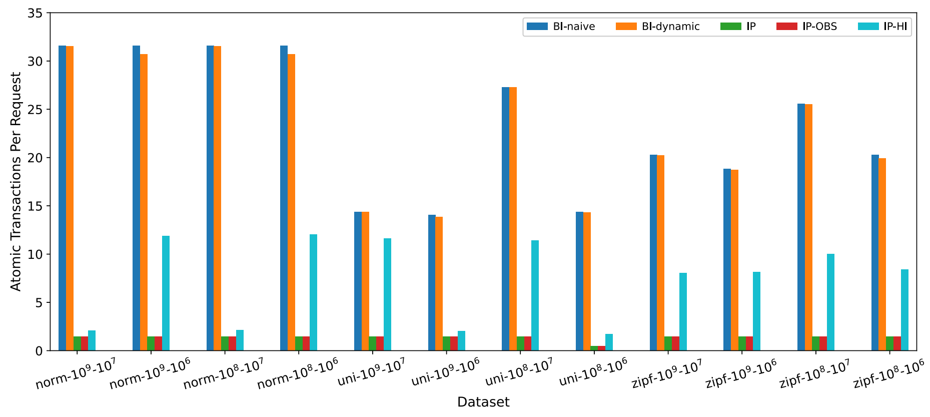

Fig. 11. SISI atomic transactions per request (lower is better).

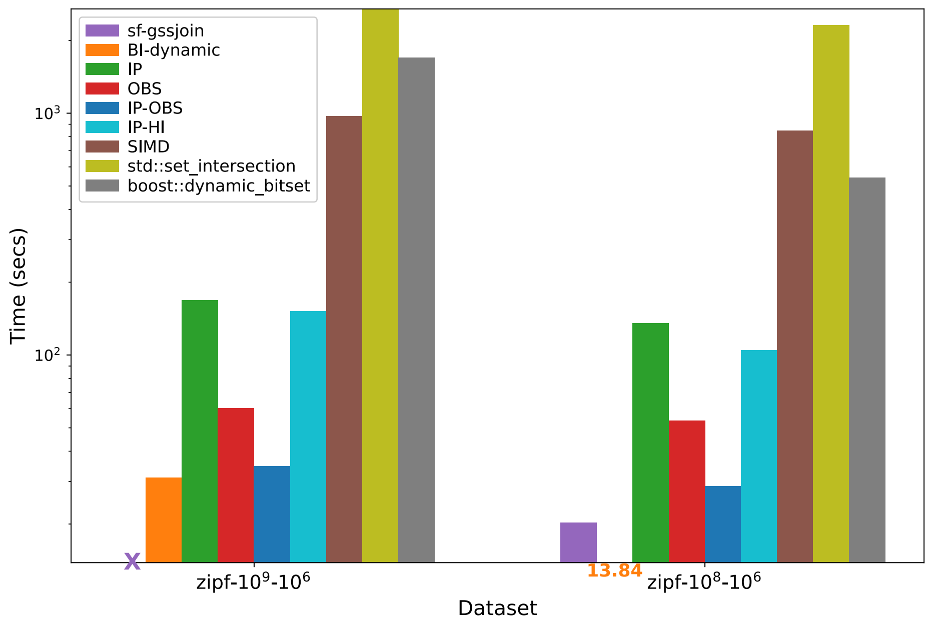

Fig. 12. Multi-Instance intersection experiments on artificial datasets with $k = 1000$. The X symbol indicates when the experiment could not run due to memory shortage. When the bars are too short, we also depict the running times.

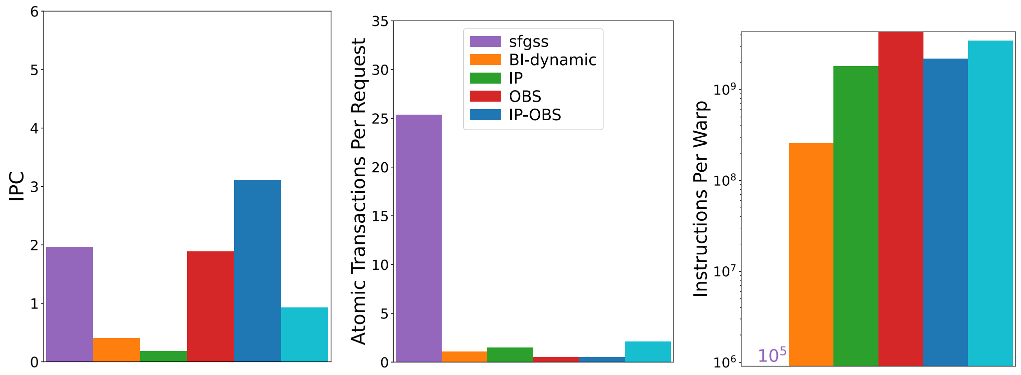

Fig. 13. MISI profiling for zipf-10^8–10^6 dataset: IPC (left), Atomic Transactions per request (middle), Instructions per warp (right).

dominant behavior of *BI-dynamic* in this dataset is attributed to the *combination* of two factors, namely low number of atomic transactions per request and not very high number of instructions per warp, which outweighs the relatively low IPC number. Out of these factors, the atomic transactions is the most important one and this explains the fact that *sf-gssjoin* is inferior to *BI-dynamic* despite its much lower instructions per warp. In the SISI case, *BI* solutions allocated a single set intersection computation to multiple blocks; otherwise the utilization would be prohibitively low. In the MISI case, a single set intersection is allocated to a single block, which leads to less atomic transactions per request. The impact of the atomic transactions is mitigated when the set sizes are smaller, as in the real-world datasets discussed below. The other low-level remarks when discussing the SISI case still hold.

We continue with the real world datasets. As shown in Fig. 14, for these datasets, there are two clear winners, namely *MM* and *sf-gssjoin*. *MM* achieves on average a 22X speedup over the best performing CPU technique in the cases where it is the best performing technique. Correspondingly, *sf-gssjoin* achieves a 25X speedup when it is the most efficient technique. In other words, GPU solutions outperform parallel CPU solutions in our setting by more 23.5X on average. In the appendix, we present exact numbers and more cases in terms of dataset sizes.

The reason behind *MM*'s efficiency is twofold and relates to the dataset characteristics. Firstly, for the BMS dataset, the small size of the element universe alone, which is at the order of 10^3, suffices to render matrix multiplication the most advantageous approach. Secondly, for the WORDS dataset, the reason behind *MM*'s superiority is that both the element universe size is rather small (order of 10^4) and the average set size only an order of magnitude smaller, hence the increased density of this dataset favors *MM* in contrast to techniques that operate directly on set elements. As already discusses, such density also benefits *BI-dynamic*. As datasets become more sparse, the efficiency of *MM* deteriorates, as it can been seen for the TWITTER dataset, where *MM* performs worse than the rest of GPU techniques despite the fact that the element universe is not that large (same order of magnitude as WORDS). In addition, for larger element universes, i.e. ENRON and ORKUT, *MM* is unable to launch, since the required memory to store the binary matrices exceeds the available GPU memory and is clearly inferior to resort to sub-matrix multiplications.

On the other hand, *sf-gssjoin* remains more robust throughout different dataset characteristics. Moreover, in its winning cases,

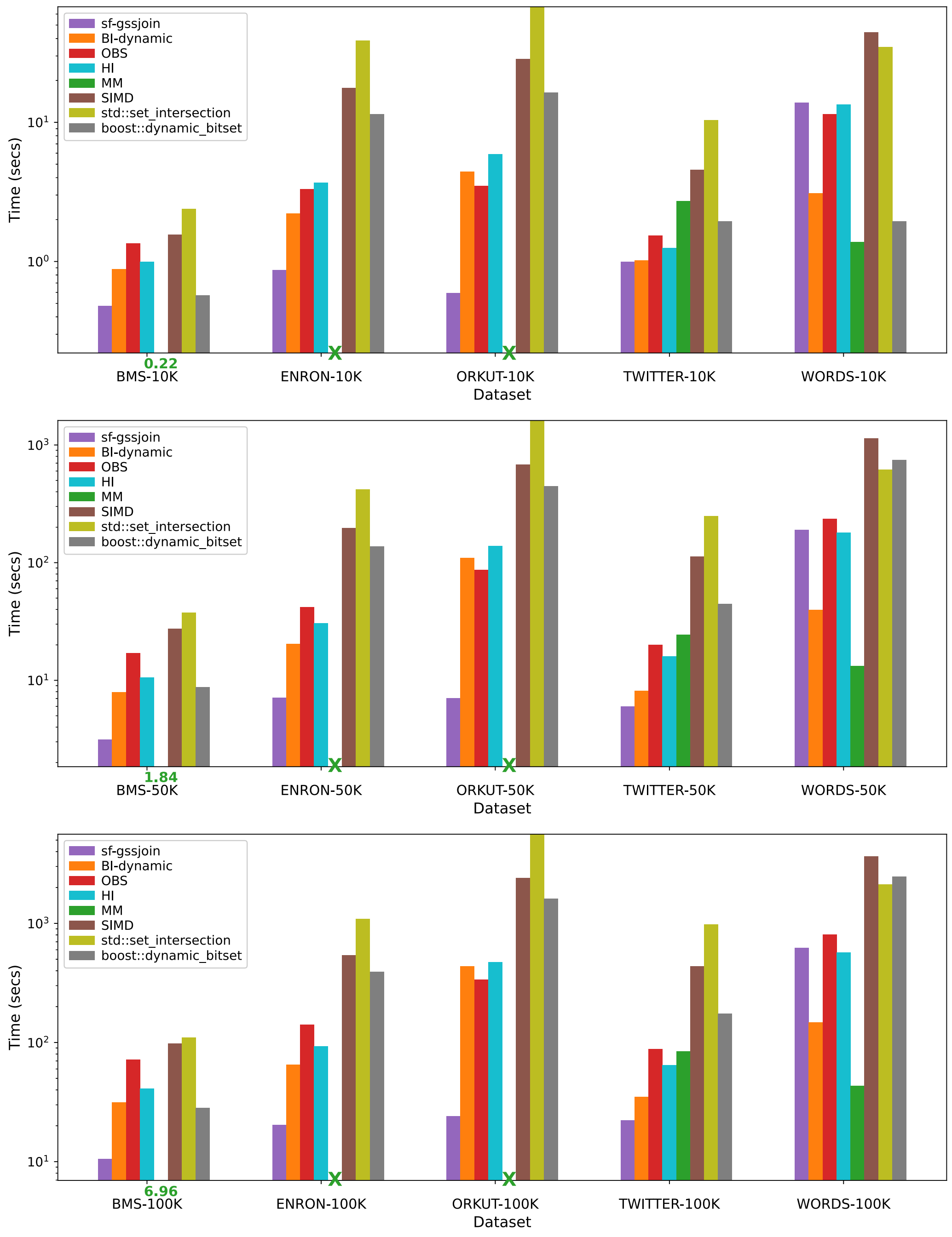

Fig. 14. Multi-Instance intersection experiments on real world datasets.

the small average set size leads in small static indices, which yields an optimal workload distribution among GPU threads and overall, results in better GPU utilization. In contrast, *BI-dynamic*, *OBS* and *HI* conduct set intersection at block level. Thus, there is an inherent workload imbalance among GPU thread blocks. We note that *IP*, and consequently its two workload allocation rationales, *IP-OBS* and *IP-HI* cannot always execute due to memory constraints. More specifically, the required memory to store the diagonals for the grid level partitioning of *IP* is $2 \times \binom{k}{2} \times (blocks+1)$. As a result, there is an upperbound to the number of blocks to comply with the global memory restriction. However, due to the small set sizes, we consider *IP* partitioning as an excessive approach to conduct set intersection on these datasets.

Based on our experimental evaluation, we conclude that for very large MISI problems in terms of set size, *BI-dynamic* and *IP-OBS* are preferable. On the other hand, when dealing with multiple instance smaller (but still large) set intersections, the grid level partitioning of *IP* adds overhead and results in severe GPU underutilization. In such cases, standalone *OBS* and *HI* perform better but are surpassed by either *MM* or *sf-gssjoin*. More specifically, *MM* is the dominant solution for small element universe sizes (order of 10^3) or for larger element universe sizes and dense datasets provided that the binary matrix fits into the main memory of the GPU; otherwise *sf-gssjoin* is the best performing technique. Finally, as in the SISI case, the development of efficient

$$
\begin{array}{l}
s_1 = \{2, 11\} \\
s_2 = \{10, 11\} \\
\hdashline
s_3 = \{3, 4, 5, 11\} \\
s_4 = \{3, 5, 10, 11\} \\
s_5 = \{5, 6, 7, 11\}
\end{array}
\qquad
\begin{array}{l}
I(1) = I(8) = I(9) = [] \\
I(2) = [s_1] \\
I(4) = [s_3] \\
I(6) = [s_5] \\
I(7) = [s_5] \\
I(3) = [s_3, s_4] \\
I(10) = [s_2, s_4] \\
\hdashline
I(5) = [s_3, s_4, s_5] \\
I(11) = [s_1, s_2, s_3, s_4, s_5]
\end{array}
\qquad
\begin{array}{c|cc}
 & 5 & 11 \\
\hline
s_3 & 1 & 1 \\
s_4 & 1 & 1 \\
s_5 & 1 & 1
\end{array}
\times
\begin{array}{c|ccc}
 & s_3 & s_4 & s_5 \\
\hline
5 & 1 & 1 & 1 \\
11 & 1 & 1 & 1
\end{array}
=
\begin{array}{c|ccc}
 & s_3 & s_4 & s_5 \\
\hline
s_3 & 2 & 2 & 2 \\
s_4 & 2 & 2 & 2 \\
s_5 & 2 & 2 & 2
\end{array}
$$

Fig. 15. An example of the MMJoin algorithm with $\Delta_1 = \Delta_2 = 3$.

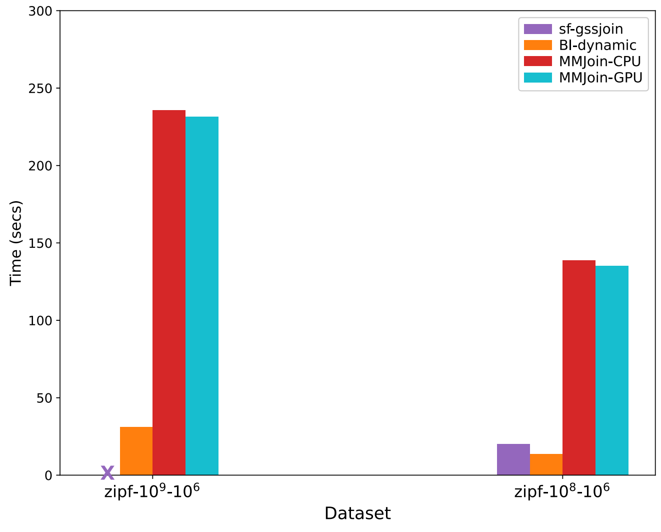

Fig. 16. Multi-Instance set containment experiments on artificial datasets.

GPU techniques should place emphasis on the atomic transactions along with aspects, such as IPC.

4. Testing set containment joins using also co-processing

In this section, we investigate the set containment join problem in the GPGPU setting. As mentioned in Section 2, all techniques presented so far can solve the SISC/MISC problems with negligible differences in runtimes due to the extremely light weight extra computation needed to compute the degree of containment. Also, we have already discussed that no GPU-based solution for the set containment join has been proposed so far. Therefore, we focus on CPU techniques that can be fully or partially transferred to the GPU. We select *MMJoin* [28], since it is the most prominent technique that can be accelerated with the GPU. Next, we give an overview of *MMJoin* and explain our modifications to yield a co-processing CPU–GPU implementation. Afterwards, we evaluate the GPU set intersection techniques presented in the previous section and adapted for the set containment join, against the CPU-standalone *MMJoin* and its GPU-enabled version. We also discuss and present experiments regarding the comparison against the second state-of-the-art (binary) set containment solution, namely LCJoin [17].

4.1. Matrix Multiplication Set Containment Join (MMJoin) [28]

Given a collection of k sets S, *MMJoin* evaluates the set containment join problem as a join query with projection using an output-sensitive algorithm. More specifically, *MMJoin* uses a twofold data partitioning scheme, parameterized by two integer constants $\Delta_1, \Delta_2 \geq 1$, to mitigate the memory allocation-related computational cost in fast matrix multiplication. First, each set $s \in S$ is classified as light if $|s| < \Delta_1$, or heavy otherwise. Similarly, each set element $e \in E$, with $I(e)$ its corresponding inverted list, is classified as light if $|I(e)| < \Delta_2$, or heavy otherwise. To process the set containment join, *MMJoin* uses a static inverted index to compute set intersections between (i) light sets for both light and heavy set elements and (ii) heavy sets for light set elements. For the heavy elements of the heavy sets, intersections are computed via matrix multiplication. The reason for this is that computing the intersections by accessing the inverted lists of the heavy set elements incurs a high computational cost, while matrix multiplication can mitigate this cost. An example of the *MMJoin* algorithm is depicted in Fig. 15.

In the original implementation of *MMJoin*, the authors use the Eigen library [38] to perform matrix multiplication on the CPU. Since GPUs support fast matrix multiplication, we modify *MMJoin* by offloading the complete matrix multiplication operation to the GPU, i.e., *MM* is partially applied. We also modify the order, in which we process the set containment join. More specifically, first we build the complete inverted index on the CPU and process all light sets. Then, we invoke the GPU for fast matrix multiplication, in a similar fashion to the MISI problem, in order to process the heavy set elements of the heavy sets. Last, we process the light set elements of the heavy set on the CPU using the inverted index.

4.2. MISC experiments

We evaluate the best performing GPU set intersection techniques, as presented in Section 3.3, adapted for set containment, against two flavors of *MMJoin*,[6] (i) *MMJoin-CPU* which runs completely on the CPU, and (ii) *MMJoin-GPU* which invokes the GPU for the matrix multiplication. We also run both *MMJoin-CPU* and *MMJoin-GPU* in parallel by using OpenMP with 12 threads. We experiment with the same artificial and real world datasets as in Section 3.3. For the majority of datasets, we use the original cost optimizer to calculate the Δ_1, Δ_2, as presented in [28]. For the ENRON and ORKUT datasets, we manually set Δ_1, Δ_2 so that the input binary matrices fit in the GPU memory. Since the GPU set intersection techniques store the complete $\binom{k}{2}$ intersections in main memory, we can directly calculate the binary set containment or the set containment degree easily in a linear pass.[7]

As shown in Fig. 16, for both artificial datasets with $k = 1000$, *BI-dynamic* is the best performing technique and achieves average 8.6X speedup compared to the *MMJoin-GPU*. For real world datasets, as shown in Fig. 17, in the majority of the experiments, the adapted GPU set intersection techniques are more efficient and on average, they achieve a 2.3X speedup compared to *MMJoin*.

6 The source code of *MMJoin* was provided by the authors of [28].

7 SISC experiments do not differ from MISC and are thus omitted.

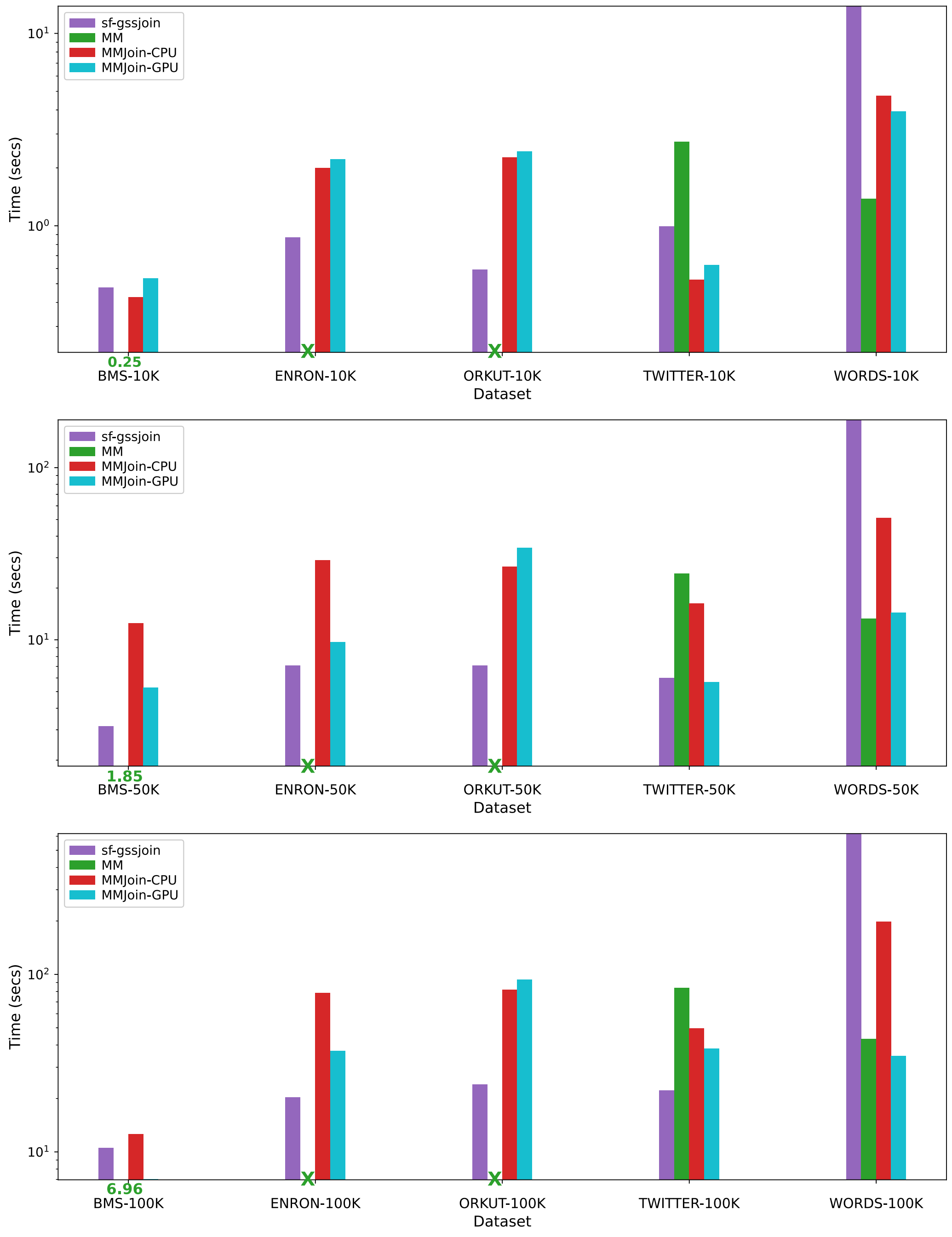

Fig. 17. Multi-Instance set containment experiments on real world datasets.

The main observation drawn from Fig. 17 is that all the conclusions drawn from the MISI experiments, as summarized at the end of Section 3.3 still hold with three exceptions, where *MMJoin* seems superior. We discuss each of these cases in turn. Firstly, parallel *MMJoin-CPU* yields the lowest running times for TWITTER-10K. However, if we look at the actual times, these are 0.52 s compared to 0.99 s achieved by *sf-gssjoin*; i.e., the differences in actual runtimes are very small if not negligible. The same applies for the TWITTER-50K.[8] The exact times are given in the appendix. The third case, in which *MMJoin* beats GPU-only solutions is for WORDS-100K. This case needs to be discussed in more detail. *MMJoin-GPU* runs faster than *MM* because the binary matrix is very large and not all parts of it are dense; therefore allocating its sparser parts to CPU, to be processed through the inverted index, is beneficial.

For completeness, we have also implemented *LCJOIN* following the algorithm presented in [17] and compared it against the rest of the evaluated solutions. However, as shown in Fig. 18, the GPU-based techniques are faster by two orders of magnitude.[9]

In summary, the main conclusion from these experiments is that GPU-based set intersection techniques can be employed to

[8] In TWITTER-50K, all 50K sets are classified as heavy; also, out of the 37 704 set elements, only 421 are classified as heavy and the rest are classified as light.

[9] Our implementation of *LCJOIN* is included in our github repository; we have been unable to obtain the implementation of the authors in [17].

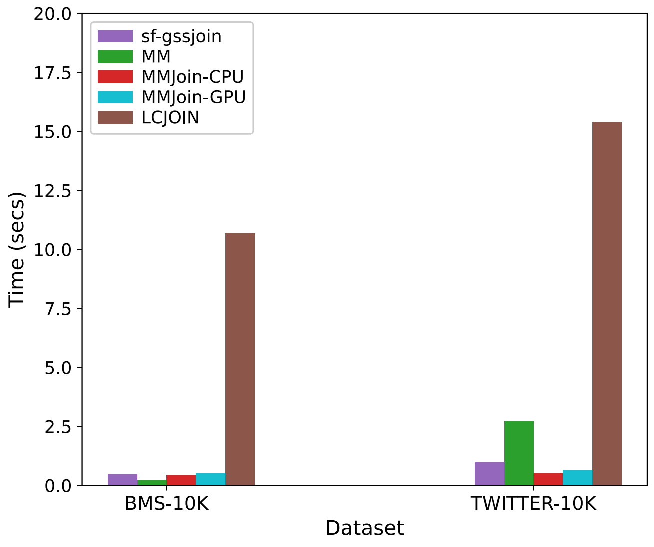

Fig. 18. Comparison of LCJOIN against GPU-based solutions.

solve set containment joins more efficiently that the current CPU-based state-of-the-art, and in specific large cases, co-processing solutions need to be considered in addition to techniques that run on a GPU exclusively.

5. Conclusion

In this work, we investigate a wide variety of approaches to large set intersection using a GPU. We adapt and evaluate GPU set intersection techniques that were previously applied to graph analytics, and although these techniques are better suited for intersecting sets of smaller size, we experimentally show that, through certain enhancements, they can be easily adapted for large set intersection. We explain which are the best approaches in the single- and multi-instance cases, and we introduce a novel hybrid, namely, combining *IP* with *OBS*, which proves to be a dominant solution for the former cases, and competitive in the latter ones. Also, employing bitmap-based solutions pays off in the multi-instance case when set sizes are very large, while, if the set sizes are relatively smaller, multi-instance set intersection stands to benefit from either adapting a set similarity join technique or employing matrix multiplication. In addition, we experimentally show that these techniques are superior to the existing state-of-the-art techniques for set containment join and can further benefit from CPU–GPU co-processing.

Overall, our work covers all the known competitive GPU-enabled approaches to large set intersection. The evaluation however is conducted using a single GPU. A direction for future work is to exploit multiple GPUs for even larger problem instances.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The research work was supported by the Hellenic Foundation for Research and Innovation (HFRI), Greece under the HFRI PhD Fellowship grant (Fellowship Number: 1154). In addition, the authors gratefully acknowledge the support of NVIDIA through the donation of the GPU (GPU Grant Program).

Table A.2
MISI runtimes in seconds with the respective speedups.

Technique	BMS				
	10K	25K	50K	75K	100K
sf-gssjoin	0.47	1.2	3.13	6.18	10.51
BI-dynamic	0.88	2.01	7.9	17.73	31.38
OBS	1.35	4.39	17.02	39.91	71.97
HI	0.99	2.72	10.56	23.85	41.00
MM	**0.22**	**0.75**	**1.84**	**4.35**	**6.96**
SIMD	1.56	8.11	27.39	72.52	97.85
std	2.39	11.11	37.45	57.44	109.49
boost	**0.57**	**2.67**	**8.79**	**17.43**	**28.17**
Speedup	2.6	3.56	4.75	4.00	4.04
Technique	ENRON				
	10K	25K	50K	75K	100K
sf-gssjoin	**0.86**	**2.35**	**7.09**	**12.50**	**20.36**
BI-dynamic	2.21	6.59	20.41	39.33	64.92
OBS	3.32	12.54	41.80	85.50	141.59
HI	3.68	10.75	30.67	58.43	92.59
MM	–	–	–	–	–
SIMD	17.64	73.09	196.85	363.75	541.25
std	38.70	148.80	419.78	736.87	1091.96
boost	**11.49**	**47.50**	137.90	254.76	391.88
Speedup	13.22	20.16	19.43	20.37	19.24
Technique	ORKUT				
	10K	25K	50K	75K	100K
sf-gssjoin	**0.59**	**2.21**	**7.07**	**14.85**	**24.06**
BI-dynamic	4.41	26.38	109.47	245.39	435.30
OBS	3.49	19.36	86.63	192.77	337.00
HI	5.89	36.01	138.74	294.20	473.64
MM	–	–	–	–	–
SIMD	28.58	188.18	679.13	1352.21	2408.17
std	67.66	418.08	1614.08	3270.73	5601.96
boost	**16.44**	**105.14**	**446.96**	**931.66**	**1611.23**
Speedup	27.69	47.51	63.14	62.71	66.95
Technique	TWITTER				
	10K	25K	50K	75K	100K
sf-gssjoin	**0.99**	**2.20**	**5.98**	**12.48**	**22.25**
BI-dynamic	1.01	2.20	8.16	18.88	34.95
OBS	1.54	4.88	20.10	47.13	88.14
HI	1.25	3.98	15.95	36.48	64.52
MM	2.73	8.53	24.31	51.01	84.08
SIMD	4.55	28.30	112.91	246.90	435.47
std	10.41	62.90	247.86	550.69	982.36
boost	**1.94**	**11.48**	**44.47**	**99.46**	**174.23**
Speedup	1.95	5.21	7.43	7.96	7.83
Technique	WORDS				
	10K	25K	50K	75K	100K
sf-gssjoin	13.85	54.26	189.54	371.13	620.93
BI-dynamic	3.08	10.46	39.78	79.45	146.92
OBS	11.48	65.40	235.98	488.92	804.00
HI	13.41	58.97	179.70	352.14	567.65
MM	**1.38**	**4.75**	**13.22**	**26.79**	**43.36**
SIMD	64.19	332.04	1135.02	2287.81	3638.85
std	**34.92**	**180.15**	**617.83**	**1263.55**	**2124.46**
boost	44.54	224.63	744.71	1397.57	2461.01
Speedup	25.23	37.91	46.71	47.115	48.99

Appendix. Additional experimental results

In the appendix, we present the exact numbers for the MISI and MISC experiments along with the consideration of additional dataset sizes (Tables A.2 and A.3).

Table A.3
MISC runtimes in seconds with the respective speedups.

Technique	BMS				
	10K	25K	50K	75K	100K
sf-gssjoin	0.48	1.21	3.14	6.19	10.51
MM	**0.25**	**0.83**	**1.85**	**4.48**	**6.96**
MMJoin-CPU	**0.42**	2.34	12.44	11.23	12.62
MMJoin-GPU	0.53	**1.59**	**5.25**	**6.57**	**7.03**
Speedup	1.7	1.91	2.83	1.47	1.01
Technique	ENRON				
	10K	25K	50K	75K	100K
sf-gssjoin	**0.87**	**2.36**	**7.16**	**12.53**	**20.38**
MM	–	–	–	–	–
MMJoin-CPU	**2.00**	12.86	28.92	63.43	78.9
MMJoin-GPU	2.22	**4.59**	**9.66**	**23.97**	**37.11**
Speedup	2.29	1.94	1.35	1.91	1.82
Technique	ORKUT				
	10K	25K	50K	75K	100K
sf-gssjoin	**0.59**	**2.21**	**7.08**	**14.86**	**24.07**
MM	–	–	–	–	–
MMJoin-CPU	**2.27**	**6.30**	**26.62**	**43.75**	**82.20**
MMJoin-GPU	2.43	6.49	34.15	44.71	93.32
Speedup	3.78	2.84	3.75	2.94	3.41
Technique	TWITTER				
	10K	25K	50K	75K	100K
sf-gssjoin	**0.99**	**2.20**	**5.98**	**12.48**	**22.25**
MM	2.77	8.59	24.40	51.14	84.02
MMJoin-CPU	**0.52**	3.02	16.22	45.09	49.57
MMJoin-GPU	0.62	**1.73**	**5.66**	**11.77**	**38.28**
Speedup	0.52	0.78	0.94	0.94	1.72
Technique	WORDS				
	10K	25K	50K	75K	100K
sf-gssjoin	13.86	54.26	189.57	371.14	621.20
MM	**1.38**	**4.82**	**13.31**	**26.89**	**43.48**
MMJoin-CPU	4.73	13.74	50.91	118.92	198.88
MMJoin-GPU	**3.92**	**9.28**	**14.37**	**29.76**	**34.74**
Speedup	2.82	1.92	1.07	1.10	0.79

References

[1] E.D. Demaine, A. López-Ortiz, J.I. Munro, Experiments on adaptive set intersections for text retrieval systems, in: Workshop on Algorithm Engineering and Experimentation, Springer, 2001, pp. 91–104.

[2] J. Barbay, A. López-Ortiz, T. Lu, Faster adaptive set intersections for text searching, in: International Workshop on Experimental and Efficient Algorithms, Springer, 2006, pp. 146–157.

[3] T. Schank, D. Wagner, Finding, counting and listing all triangles in large graphs, an experimental study, in: International Workshop on Experimental and Efficient Algorithms, Springer, 2005, pp. 606–609.

[4] N. Wang, J. Zhang, K.-L. Tan, A.K. Tung, On triangulation-based dense neighborhood graph discovery, Proc. VLDB Endow. 4 (2) (2010) 58–68.

[5] X. Huang, H. Cheng, L. Qin, W. Tian, J.X. Yu, Querying k-truss community in large and dynamic graphs, in: Proceedings of the 2014 ACM SIGMOD Int. Conf. on Management of Data, 2014, pp. 1311–1322.

[6] V. Raman, L. Qiao, W. Han, I. Narang, Y.-L. Chen, K.-H. Yang, F.-L. Ling, Lazy, adaptive rid-list intersection, and its application to index anding, in: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, 2007, pp. 773–784.

[7] J. Han, M. Kamber, J. Pei, Data Mining: Concepts and Techniques, third ed., Morgan Kaufmann, 2011.

[8] O. Green, P. Yalamanchili, L.-M. Munguía, Fast triangle counting on the gpu, in: Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms, 2014, pp. 1–8.

[9] L. Wang, Y. Wang, C. Yang, J.D. Owens, A comparative study on exact triangle counting algorithms on the gpu, in: Proceedings of the ACM Workshop on High Performance Graph Processing, 2016, pp. 1–8.

[10] M. Bisson, M. Fatica, High performance exact triangle counting on gpus, IEEE Trans. Parallel Distrib. Syst. 28 (12) (2017) 3501–3510.

[11] J. Fox, O. Green, K. Gabert, X. An, D.A. Bader, Fast and adaptive list intersections on the gpu, in: 2018 IEEE High Performance Extreme Computing Conference, HPEC, IEEE, 2018, pp. 1–7.

[12] Y. Hu, H. Liu, H.H. Huang, Tricore: Parallel triangle counting on gpus, in: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2018, pp. 171–182.

[13] S. Pandey, X.S. Li, A. Buluc, J. Xu, H. Liu, H-index: Hash-indexing for parallel triangle counting on gpus, in: 2019 IEEE High Performance Extreme Computing Conference, HPEC, IEEE, 2019, pp. 1–7.

[14] C. Bellas, A. Gounaris, An empirical evaluation of exact set similarity join techniques using gpus, Inf. Syst. 89 (2020) 101485.

[15] L. Hu, L. Zou, Y. Liu, Accelerating triangle counting on gpu, in: Proceedings of the 2021 International Conference on Management of Data, 2021, pp. 736–748.

[16] C. Bellas, A. Gounaris, An evaluation of large set intersection techniques on gpus, in: K. Stefanidis, P. Marcel (Eds.), Proceedings of the 23rd International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data (DOLAP), 2840 of CEUR Workshop Proceedings, 2021, pp. 111–115.

[17] D. Deng, C. Yang, S. Shang, F. Zhu, L. Liu, L. Shao, Lcjoin: Set containment join via list crosscutting, in: 35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11 2019, IEEE, 2019, pp. 362–373.

[18] S. Ding, J. He, H. Yan, T. Suel, Using graphics processors for high performance IR query processing, in: Proceedings of the 18th International Conference on World Wide Web, 2009, pp. 421–430.

[19] D. Wu, F. Zhang, N. Ao, F. Wang, X. Liu, G. Wang, A batched gpu algorithm for set intersection, in: 2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks, IEEE, 2009, pp. 752–756.

[20] D. Wu, F. Zhang, N. Ao, G. Wang, X. Liu, J. Liu, Efficient lists intersection by cpu-gpu cooperative computing, in: 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, IPDPSW, IEEE, 2010, pp. 1–8.

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

[22] A. Polak, Counting triangles in large graphs on gpu, in: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW, IEEE, 2016, pp. 740–746.

[23] S. Ribeiro-Junior, R.D. Quirino, L.A. Ribeiro, W.S. Martins, Fast parallel set similarity joins on many-core architectures, J. Inf. Data Manag. 8 (3) (2017) 255.

[24] J. Yang, W. Zhang, S. Yang, Y. Zhang, X. Lin, L. Yuan, Efficient set containment join, VLDB J. 27 (4) (2018) 471–495.

[25] R. Jampani, V. Pudi, Using prefix-trees for efficiently computing set joins, in: L. Zhou, B.C. Ooi, X. Meng (Eds.), Database Systems for Advanced Applications, 10th International Conference, DASFAA 2005, Beijing, China, April 17-20 2005, Proceedings, in: Lecture Notes in Computer Science, vol. 3453, Springer, 2005, pp. 761–772.

[26] P. Bouros, N. Mamoulis, S. Ge, M. Terrovitis, Set containment join revisited, Knowl. Inf. Syst. 49 (1) (2016) 375–402.

[27] A. Kunkel, A. Rheinländer, C. Schiefer, S. Helmer, P. Bouros, U. Leser, Piejoin: Towards parallel set containment joins, in: P. Baumann, I. Manolescu-Goujot, L. Trani, Y.E. Ioannidis, G.G. Barnafóldi, L. Dobos, E. Bányai (Eds.), Proceedings of the 28th International Conference on Scientific and Statistical Database Management, SSDBM 2016, Budapest, Hungary, July 18-20 2016, ACM, 2016, pp. 11:1–11:12.

[28] S. Deep, X. Hu, P. Koutris, Fast join project query evaluation using matrix multiplication, in: D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, H.Q. Ngo (Eds.), Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, Online Conference [Portland, OR, USA], June 14-19 2020, ACM, 2020, pp. 1213–1223.

[29] J. Yang, W. Zhang, S. Yang, Y. Zhang, X. Lin, Tt-join: Efficient set containment join, in: 33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22 2017, IEEE Computer Society, 2017, pp. 509–520.

[30] J. Luo, W. Zhang, S. Shi, H. Gao, J. Li, W. Wu, S. Jiang, Freshjoin: An efficient and adaptive algorithm for set containment join, Data Sci. Eng. 4 (4) (2019) 293–308.

[31] Y. Luo, G.H.L. Fletcher, J. Hidders, P.D. Bra, Efficient and scalable trie-based algorithms for computing set containment relations, in: J. Gehrke, W. Lehner, K. Shim, S.K. Cha, G.M. Lohman (Eds.), 31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17 2015, IEEE Computer Society, 2015, pp. 303–314, http://dx.doi.org/10.1109/ICDE.2015.7113293.

[32] S. Helmer, G. Moerkotte, Evaluation of main memory join algorithms for joins with set comparison join predicates, in: M. Jarke, M.J. Carey, K.R. Dittrich, F.H. Lochovskya, P. Loucopoulos, M.A. Jeusfeld (Eds.), VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29 1997 , Athens, Greece, Morgan Kaufmann, 1997, pp. 386–395.

[33] S. Melnik, H. Garcia-Molina, Adaptive algorithms for set containment joins, ACM Trans. Database Syst. 28 (2003) 56–99.
[34] Z. Huang, N. Ma, S. Wang, Y. Peng, Gpu computing performance analysis on matrix multiplication, J. Eng. 2019 (23) (2019) 9043–9048.
[35] cublas, https://developer.nvidia.com/cublas.
[36] D. Lemire, L. Boytsov, N. Kurz, SIMD compression and the intersection of sorted integers, Softw. - Pract. Exp. 46 (6) (2016) 723–749.
[37] words, https://archive.ics.edu/ml/datasets/bag+of+words.
[38] eigen, https://eigen.tuxfamily.org/.