An Evaluation of Large Set Intersection Techniques on GPUs

Christos Bellas, Anastasios Gounaris
Aristotle University of Thessaloniki
Thessaloniki, Greece
(chribell,gounaria)@csd.auth.gr

ABSTRACT

In this paper, we focus 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 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, while we investigate new hybrids for completeness. Finally, we present a comprehensive evaluation highlighting 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.

1 INTRODUCTION

Set intersection is an essential operation in numerous application domains, such as information retrieval for text search engines using an inverted index [2, 5], graph analytics for triangle counting and community detection [11, 17, 19], and database systems for merging RID-lists [15] and performing fast (potentially bitwise) operations on data in columnar format [9].

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 [4, 7, 8, 10, 13, 18]. 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 [3].

In this work, we compare and evaluate state-of-the-art GPU techniques for set intersection, when the sets are large, i.e., they contain millions of elements. More specifically, we gather together techniques belonging to five different rationales, namely intersect path, optimized binary search, hash, bitmap and set similarity join-based solutions, while we include novel promising hybrid solutions that combine existing techniques to better fit into our setting.

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 two intersection path flavors for the single-instance problem and of either bitmap-based or similarity join-based solutions for the multi-instance problem depending on the size of the sets.[1]

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

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

2 PRELIMINARIES

In this section, we first give a formal notation for the set intersection problem. Next, we give an overview of the related work about set intersection after explaining the exact scope of this paper.

2.1 Set Intersection Problem Description

Single-instance set intersection (SISI) problem: given i) a finite universe of elements E, and ii) a set $A = \{e_1^A, \ldots, e_n^A\}$ of size n and a set $B = \{e_1^B, \ldots, e_m^B\}$ 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 $C = \{S_1, \ldots, S_k\}$, and we aim to find the set intersection among all $\binom{k}{2}$ of pairs (S_i, S_j), where $0 < i < j \leq k$.

Obviously, the solutions of SISI are of $\Omega(n + m)$ complexity, whereas MISI solutions are of quadratic complexity in k without allowing for any pair pruning, as, for example, in problems such as set similarity joins [3]. 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 when n and m are at the order of millions and the size of E is orders of magnitude larger.

2.2 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 are tailored to specific algorithmic optimizations. In the context of this work, we extract, adapt and evaluate these techniques under a more demanding large set intersection scenario performing also a comparison against all methodologies proposed that can address the SISI and MIMI problems for large sets.

2.3 Overview of existing solutions

We present the relevant techniques mostly in chronological order. Ding et al. [6] 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 [20], Wu et al. highlight the inefficiency of the approach followed in [6] for sets of smaller size. Subsequently, they propose a GPU technique for 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 with 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 [21], 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 [1] further extends the original work in [21] by introducing a linear regression approach to reduce 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 [7], the authors extend the work of [8] by proposing an adaptive load balancing technique and dynamically assign work to GPU threads based on work estimations. The authors of [14] and [18] 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 and similar to a sequential CPU approach.

In [4], 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 is per thread, per warp or per block.

In [10], Hu et al. demonstrate that set intersection is faster employing efficient binary search-based techniques than IP-based techniques, arguing that the latter suffer from nontrivial 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 additional speedup gains.

In [13], Pandey et al. propose a hash-based technique for set intersection. In brief, first the 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.

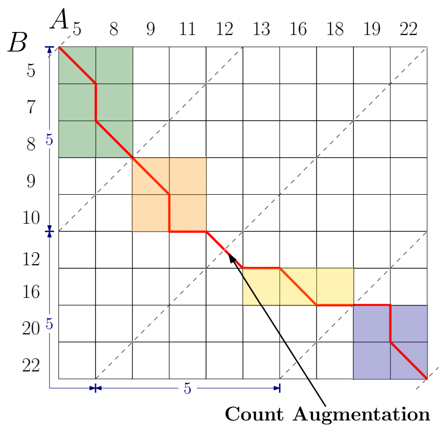

Figure 1: Intersect Path example using 4 thread blocks.

Last, in the context of set similarity join, the authors of [16] 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 and MISI problems.

As already mentioned, some of these works, such as [4, 8, 10, 14, 18] 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.

3 TECHNIQUES

In the previous section, we have identified five different methodologies, based on (1) IP, (2) optimized binary search, (3) bitmap operations, (4) hashing and (5) set similarity joins, respectively. Here, we present five different state-of-the-art GPU techniques for set intersection, one for each methodology, 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 [14, 18] that mostly rely on the preprocessing of input data and conduct intersection with a simple merge fashion algorithm. In contrast, we evaluate techniques that are more adaptable in a general set intersection scenario. We give a concise presentation for each evaluated technique below.

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 kernel grid level and the other on block level. On 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 this intersection is not assigned to any thread block. An example of this scenario is shown in Figure 1. On 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.

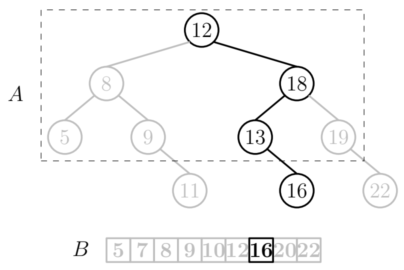

Figure 2: Optimized Binary Search example.

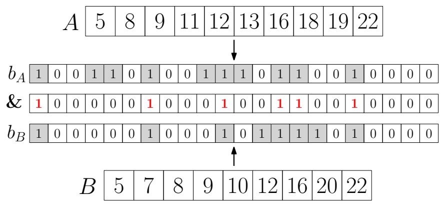

Figure 3: Naive Bitmap-based Intersection example.

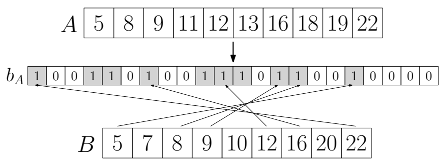

Figure 4: Dynamic Bitmap-based Intersection example.

Optimized Binary Search (OBS) [10]. 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 Figure 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 every subsequent one to perform the intersection for every pair. This requires sets to be sorted by their size.

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. The initial hashing of the smaller set is preferred in order to reduce the number of collisions. Buckets are statically allocated once in linear 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 overwrite the previous one.

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

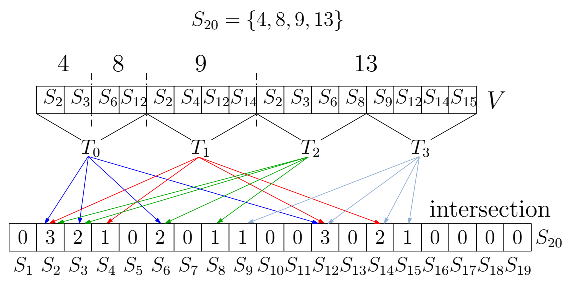
$S_{20} = \{4, 8, 9, 13\}$
4
8
9
13
V
T_0
T_1
T_2
T_3
intersection
0 3 2 1 0 2 0 1 1 0 0 3 0 2 1 0 0 0 0
S_{20}
S_1 S_2 S_3 S_4 S_5 S_6 S_7 S_8 S_9 S_{10} S_{11} S_{12} S_{13} S_{14} S_{15} S_{16} S_{17} S_{18} S_{19}

Figure 5: Static-index Intersection example using 4 GPU threads (Adapted from [3]).

(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 [4]. 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 Figure 3. 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 Figure 4).

Static-index Intersection (sf-gssjoin) [16]. 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 with the next ones. To ensure correctness, threads increment the intersection counts by using atomic operations. An example of *sf-gssjoin* is shown in Figure 5.

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. 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.

4 EVALUATION

In this section, we evaluate the techniques in the previous section, 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 instructions [12], (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*. 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

Dataset	Cardinality	Avg set size	# Element Universe
ENRON	$1.0 \cdot 10^4$	794	$1.1 \cdot 10^6$
ORKUT	$1.0 \cdot 10^4$	1001	$8.7 \cdot 10^6$
TWITTER	$1.0 \cdot 10^4$	150	$3.7 \cdot 10^4$

Table 1: Real world dataset characteristics.

memory allocations, by using the CUDA event API. We do not measure any preprocess time. The experiments were conducted on a machine with an Intel i7 5820k clocked at 3.3 GHz, 32 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 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 three real world datasets previously employed in [3]. More specifically, for each sorted dataset we extract the last $k = 10000$ sets, i.e. the largest sets. Table 1 gives an overview of the real-world datasets' characteristics.

4.1 SISI experiments

We examine the single instance scenario using twelve artificial datasets (combinations of 3 distributions, 2 element universe sizes and 2 set sizes). We have also experimented with several block sizes, but due to lack of space, we present the results of the best configuration for each technique.

As shown in Figure 6, *IP* and *IP-OBS* are the clear winners for the SISI epxeriments, 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. 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.

4.2 MISI experiments

For the multi-instance evaluation, we conduct two sets of experiments. In the first one, we use 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.

As shown in Figure 7, in both artificial datasets with $k = 1000$, *BI-dynamic* is the best performing technique and achieves average 35X 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*. Last, even though *IP* and *IP-HI* are the worst performing GPU techniques, they manage to achieve average 5.7X speedup compared to the best performing CPU technique.

As shown in Figure 8, for real world datasets with $k = 10000$, *sf-gssjoin* is the most efficient technique across every dataset and achieves average 147X speedup over the best performing CPU technique. Moreover, 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 large MISI problems, *BI-dynamic* and *IP-OBS* are preferable. On the other hand, when dealing with multiple instance small 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 *sf-gssjoin*, which achieves optimal GPU utilization.

5 CONCLUSION

In this work, we adapt and evaluate GPU set intersection techniques that were previously applied to graph analytics. 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. Finally, if the set sizes are relatively smaller, multi-instance set intersection stands to benefit from adapting a set similarity join technique.

Acknowledgments. The research work was supported by the Hellenic Foundation for Research and Innovation (HFRI) 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).

REFERENCES

[1] Naiyong Ao, Fan Zhang, Di Wu, Douglas S Stones, Gang Wang, Xiaoguang Liu, Jing Liu, and Sheng Lin. 2011. Efficient parallel lists intersection and index compression algorithms using graphics processing units. *Proceedings of the VLDB Endowment* 4, 8 (2011), 470–481.

[2] Jérémy Barbay, Alejandro López-Ortiz, and Tyler Lu. 2006. Faster adaptive set intersections for text searching. In *International Workshop on Experimental and Efficient Algorithms*. Springer, 146–157.

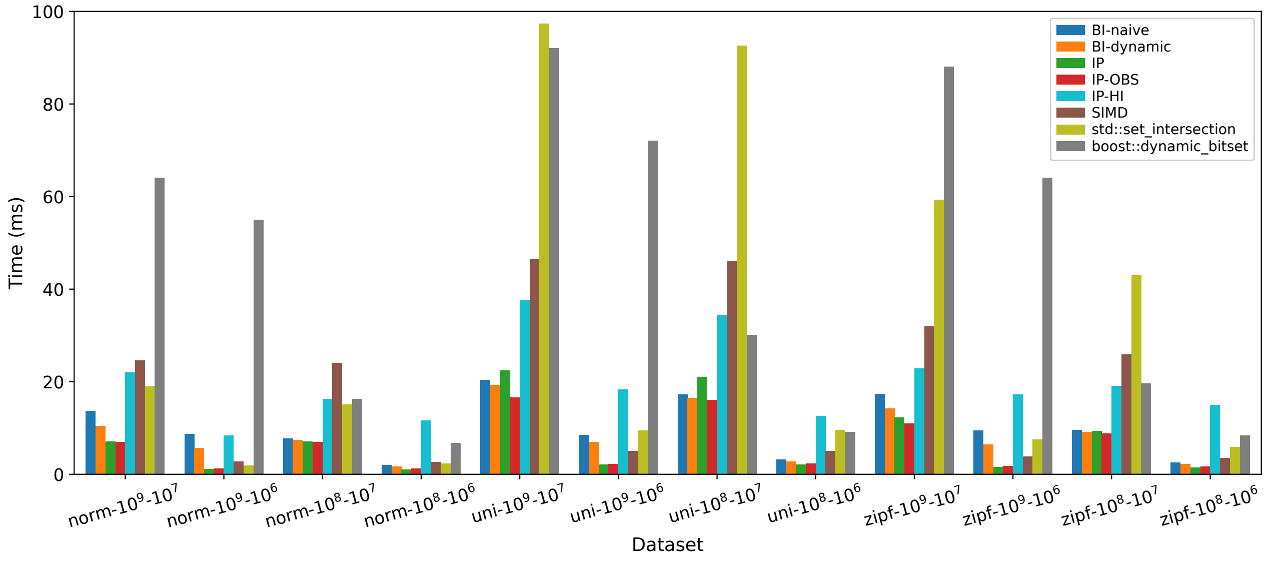

Figure 6: Single-instance experiments on artificial datasets.

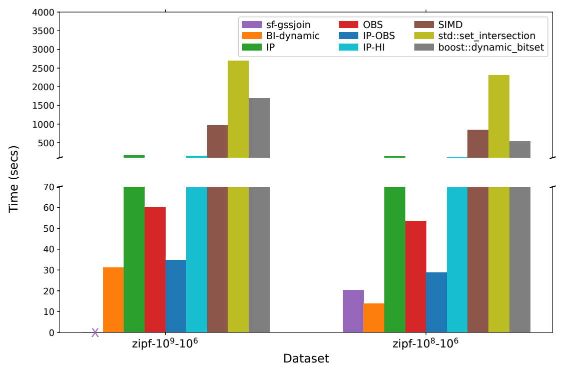

Figure 7: Multi-Instance experiments on artificial datasets with $k = 1000$.

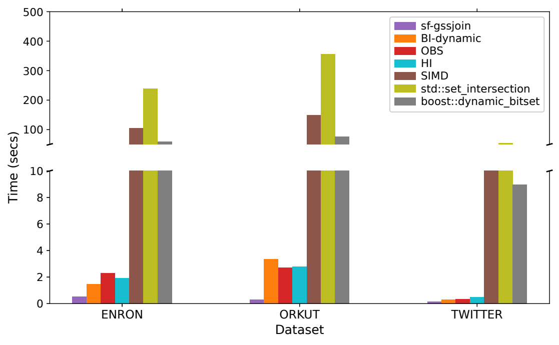

Figure 8: Multi-Instance experiments on real world datasets with $k = 10000$.

[3] Christos Bellas and Anastasios Gounaris. 2020. An empirical evaluation of exact set similarity join techniques using GPUs. *Inf. Syst.* 89 (2020), 101485.

[4] Mauro Bisson and Massimiliano Fatica. 2017. High performance exact triangle counting on gpus. *IEEE Transactions on Parallel and Distributed Systems* 28, 12 (2017), 3501–3510.

[5] Erik D Demaine, Alejandro López-Ortiz, and J Ian Munro. 2001. Experiments on adaptive set intersections for text retrieval systems. In *Workshop on Algorithm Engineering and Experimentation*. Springer, 91–104.

[6] Shuai Ding, Jinru He, Hao Yan, and Torsten Suel. 2009. Using graphics processors for high performance IR query processing. In *Proceedings of the 18th international conference on World wide web*. 421–430.

[7] James Fox, Oded Green, Kasimir Gabert, Xiaojing An, and David A Bader. 2018. Fast and adaptive list intersections on the gpu. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–7.

[8] 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*. 1–8.

[9] Jiawei Han, Micheline Kamber, and Jian Pei. 2011. *Data Mining: Concepts and Techniques, 3rd edition*. Morgan Kaufmann.

[10] Yang Hu, Hang Liu, and H Howie Huang. 2018. Tricore: Parallel triangle counting on gpus. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 171–182.

[11] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. In *Proceedings of the 2014 ACM SIGMOD Int. Conf. on Management of data*. 1311–1322.

[12] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. 2016. SIMD compression and the intersection of sorted integers. *Software: Practice and Experience* 46, 6 (2016), 723–749.

[13] Santosh Pandey, Xiaoye Sherry Li, Aydin Buluc, Jiejun Xu, and Hang Liu. 2019. H-INDEX: Hash-Indexing for Parallel Triangle Counting on GPUs. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[14] Adam Polak. 2016. Counting triangles in large graphs on GPU. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 740–746.

[15] Vijayshankar Raman, Lin Qiao, Wei Han, Inderpal Narang, Ying-Lin Chen, Kou-Horng Yang, and Fen-Ling Ling. 2007. Lazy, adaptive rid-list intersection, and its application to index anding. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 773–784.

[16] 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), 255–255.

[17] Thomas Schank and Dorothea Wagner. 2005. Finding, counting and listing all triangles in large graphs, an experimental study. In *International workshop on experimental and efficient algorithms*. Springer, 606–609.

[18] Leyuan Wang, Yangzihao Wang, Carl Yang, and John D Owens. 2016. A comparative study on exact triangle counting algorithms on the gpu. In *Proceedings of the ACM Workshop on High Performance Graph Processing*. 1–8.

[19] Nan Wang, Jingbo Zhang, Kian-Lee Tan, and Anthony KH Tung. 2010. On triangulation-based dense neighborhood graph discovery. *Proceedings of the VLDB Endowment* 4, 2 (2010), 58–68.

[20] Di Wu, Fan Zhang, Naiyong Ao, Fang Wang, Xiaoguang Liu, and Gang Wang. 2009. A batched gpu algorithm for set intersection. In *2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks*. IEEE, 752–756.

[21] Di Wu, Fan Zhang, Naiyong Ao, Gang Wang, Xiaoguang Liu, and Jing Liu. 2010. Efficient lists intersection by cpu-gpu cooperative computing. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE, 1–8.