ISBN 978 -987-633-574-4

Goodness of the GPU Permutation Index: Performance and Quality Results

Mariela Lopresti, Fabiana Piccoli, Nora Reyes

LIDIC. Universidad Nacional de San Luis,
Ejército de los Andes 950 - 5700 - San Luis - Argentina
{omlopres,mpiccoli,nreyes}@ unsl.edu.ar

Abstract. Similarity searching is a useful operation for many real applications that work on non-structured or multimedia databases. In these scenarios, it is significant to search similar objects to another object given as a query. There exist several indexes to avoid exhaustively review all database objects to answer a query. In many cases, even with the help of an index, it could not be enough to have reasonable response times, and it is necessary to consider approximate similarity searches. In this kind of similarity search, accuracy or determinism is traded for faster searches. A good representative for approximate similarity searches is the *Permutation Index*.
In this paper, we give an implementation of the *Permutation Index* on GPU to speed approximate similarity search on massive databases. Our implementation takes advantage of the GPU parallelism. Besides, we consider speeding up the answer time of several queries at the same time. We also evaluate our parallel index considering answer quality and time performance on the different GPUs. The search performance is promising, independently of their architecture, because of careful planning and the correct resources use.

1 Introduction

For a query in a multimedia database, it is meaningless to look for elements exactly equal to a given one as a query. Instead, we need to measure the similarity (or dissimilarity) between the query object and each database object. The similarity search problem can be formally defined through the metric space model. It is a paradigm that allows modeling all the similarity search problems. A metric space (X, d) is composed of a universe of valid objects X and a distance function defined among them, that determines the similarity (or dissimilarity) between two given objects and satisfies properties that make it a metric. Given a dataset of n objects, a query can be trivially answered by performing n distance evaluations, but a sequential scan does not scale for large problems. The reduction of the number of distance evaluations is meaningful to achieve better results. Therefore, in many cases, preprocessing the dataset is an important option to solve queries with as few distance computations as possible. An index helps to retrieve the objects from the database that are relevant to the query by making much less than n distance evaluations during searches [1]. One of these indices is the *Permutation Index* [2].

The *Permutation Index* is an approximate similarity search algorithms to solve *inexact similarity searching* [3]. In this kind of similarity search, accuracy or determinism is traded for faster searches [1, 4]. There are many applications where their metric-space modelizations already involve an approximation to reality. Hence, a second approximation at search time is usually acceptable.

For very large metric databases, it is not enough to preprocess the dataset to build an index. It is also necessary higher speed, in consequence, techniques of

ISBN 978 -987-633-574-4

high-performance computing (HPC)[5, 6] are considered. The Graphics Processing Units (GPU)[7] are a meaningful alternative to employ HPC in the dataset preprocess to obtain an index and to answer posed queries. The GPU is attractive in many application areas for its characteristics because of its parallel execution capabilities. They promise more than an order of magnitude speedup over conventional processors for some non-graphics computations.

In metric spaces, the indexing and query resolution are the most common operations. They have several aspects that accept optimizations through the application of HPC techniques. There are many parallel solutions for some metric space operations implemented to GPU. Querying by k-Nearest Neighbors (k-NN) has concentrated the greatest attention of researchers in this area, so there are many solutions that consider GPU. In [8–11] different proposal are made, all of them are improvements to brute force algorithm (sequential scan) to find the k-NN of a query object. In [9], Kruslis et al. propose an GPU solution to the Permutation Index. They focus in high dimensional DB and use Bitonic Sort. Their performance results are good.

The goal of this work is to analyze the trade-off between the quality of similarity queries answer and time performance, using a parallel permutation index implemented on GPU. In this analysis, we consider: different databases, two well known measures of answer quality in the information retrieval area: *recall* and *precision*, and some performance parameters to evaluate parallel implementations.

The paper is organized as follows: the two next sections describe all the previous concepts. Sections 4 and 5 sketch the characteristics of our proposal and its empirical performance. Finally, the conclusions and future works are exposed.

2 Metric Space Model

A metric space (X, d) is composed of a universe of valid objects X and a distance function $d : X \times X \to R^+$ defined among them. The distance function determines the similarity (or dissimilarity) between two given objects and satisfies several properties such as strict positiveness (except $d(x, x) = 0$, which must always hold), symmetry ($d(x, y) = d(y, x)$), and the triangle inequality ($d(x, z) \leq d(x, y) + d(y, z)$). The finite subset $U \subseteq X$ with size $n = |U|$, is called the *database* and represents the set of objects of the search space. The distance is assumed to be expensive to compute, hence it is customary to define the search complexity as the number of distance evaluations performed, disregarding other components. There are two main queries of interest [1, 4]: Range Searching and the k-NN. The goal of a range search (q, r) is to retrieve all the objects $x \in U$ within the radius r of the query q (i.e. $(q, r) = \{x \in U/d(q, x) \leq r\}$). In k-NN queries, the objective is to retrieve the set k-NN(q)$\subseteq U$ such that $\mid k\text{-NN}(q) \mid = k$ and $\forall x \in k\text{-NN}(q), v \in U \wedge v \notin k\text{-NN}(q), d(q, x) \leq d(q, v)$. These two queries are considered "exact" because both retrieve all the elements that satisfy the query criterium.

When an index is defined, it helps to retrieve the objects from U that are relevant to the query by making much less than n distance evaluations during searches. The saved information in the index can vary, some indices store a subset of distances between objects, others maintain just a range of distance values. In general, there is a tradeoff between the quantity of information maintained in the index and the query cost it achieves. As more information an index stores (more memory it uses), lower query cost it obtains. However, there are some indices that use memory better than others. Therefore in a database of n objects, the most information an index could store is the $n(n-1)/2$ distances among all

ISBN 978 -987-633-574-4

element pairs from the database. This is usually avoided because $O(n^2)$ space is unacceptable for realistic applications [12].

Proximity searching in metric spaces usually are solved in two stages: preprocessing and query time. During the preprocessing stage an index is built and it is used during query time to avoid some distance computations. Basically the state of the art in this area can be divided in two families [1]: *pivot-based algorithms* and *compact-partition-based algorithms*.

There is an alternative to "exact" similarity searching called *approximate similarity searching* [3], where accuracy or determinism is traded for faster searches [1, 4], and encompasses *approximate* and *probabilistic algorithms*. The goal of approximate similarity search is to reduce *significantly* search times by allowing some errors in the query output. In approximate algorithms one usually has a threshold ϵ as parameter, so that the retrieved elements are guaranteed to have a distance to the query q at most $(1+\epsilon)$ times of what was asked for [13]. This relaxation gives faster algorithms as the threshold ϵ increases [13, 14]. On the other hand, probabilistic algorithms state that the answer is correct with high probability [15, 16]. That is, if a k-NN query of an element $q \in X$ is posed to the index, it answers with the k elements viewed as the k closest elements from U between only the elements that are actually compared with q. However, as we want to save as many distance calculations as we can, q will not be compared against many potentially relevant elements. If the exact answer of k-NN$(q) = \{x_1, x_2, \ldots, x_k\}$, it determines the radius $r_k = \max_{1 \le i \le k}\{d(x_i, q)\}$ needed to enclose these k closest elements to q.

2.1 Quality Measures of Approximate Search

An approximate answer of k-NN(q) could obtain some elements z whose $d(q, z) > r_k$. Besides, an approximate range query of (q, r) can answer a subset of the exact answer, because it is possible that the algorithm did not have reviewed all the relevant elements. However, all the answered elements will be at distance less or equal to r, so they belong to the exact answer to (q, r).

In most of information retrieval (*IR*) systems it is necessary to evaluate retrieval effectiveness [17]. Many measures of retrieval effectiveness have been proposed. The most commonly used are *recall* and *precision*, where *recall* is the ratio of relevant documents retrieved for a given query over the number of relevant documents for this query in the database; and *precision* is the ratio of the number of relevant retrieved documents over the total number of documents retrieved. Both recall and precision take on values between 0 and 1.

In general IR systems, only in small test collections, the denominator of both ratios is generally unknown and must be estimated by sampling or some other method. However, in our case we can obtain the exact answer for each query q, as the set of relevant elements for this query in U. By this way it is possible to evaluate both measures for an approximate similarity search index. For each query element q, the exact k-NN$(q) = Rel(q)$ is determined with some exact metric access method. The approximate-k-NN$(q) = Retr(q)$ is answered with an approximate similarity search index, let be the set $Retr(q) = \{y_1, y_2, \ldots, y_k\}$. It can be noticed that the approximate search will also return k elements, so $|Retr(q)| = |Rel(q)| = k$. Thus, we can determine the number of k elements obtained which are relevant to q by verifying if $d(q, y_i) \le r_k$; that is $|Rel(q) \cap Retr(q)|$. In this case both measures are coincident:

$$recall = \frac{|Rel(q) \cap Retr(q)|}{|Rel(q)|} = \frac{|Rel(q) \cap Retr(q)|}{k}$$

and

ISBN 978 -987-633-574-4

$$precision = \frac{|Rel(q) \cap Retr(q)|}{|Retr(q)|} = \frac{|Rel(q) \cap Retr(q)|}{k},$$

and will allow us to evaluate the effectiveness of our proposal. In range queries the precision measure is always equal to 1. Thus, we decide to use recall in order to analyze the retrieval effectiveness of our proposal, both in k-NN and range queries.

2.2 GPGPU

Mapping general-purpose computation onto GPU implies to use the graphics hardware to solve any applications, not necessarily of graphic nature. This is called GPGPU (General-Purpose GPU), GPU computational power is used to solve general-purpose problems [18, 19, 7]. The parallel programming over GPUs has many differences from parallel programming in typical parallel computer, the most relevant are: the *number of processing units*, the *CPU-GPU memory structure*, and the *number of parallel threads*.

Every GPGPU program has many basic steps, first the input data transfers to the graphics card. Once the data are in place on the card, many threads can be started (with little overhead). Each thread works over its data and, at the end of the computation, the results should be copied back to the host main memory. Not all kind of problem can be solved in the GPU architecture, the most suitable problems are those that can be implemented with stream processing and using limited memory, i.e. applications with abundant parallelism. Each GPU-algorithm must be carefully analyzed and its data structures must be designed considering hierarchy of GPU memory, its architectures and limitations. A good GPU-algorithm has the next characteristics:

- As the data transfers between CPU and GPU could take significant amount of time, therefore, these have to be overlapped or reduced.
- The algorithm must adopt to the MIMD and SIMD paradings, and accept the SIMT execution model.
- A lot of workload needs to be spawned in order to utilize efficiently all available GPU cores.

The Compute Unified Device Architecture (CUDA) enables to use GPU as a highly parallel computer for non-graphics applications [20, 21]. CUDA provides an essential high-level development environment with standard C/C++ language. It defines the GPU architecture as a programmable graphic unit which acts as a coprocessor for CPU. The CUDA programming model has two main characteristics: the parallel work through concurrent threads and the memory hierarchy. The user supplies a single source program encompassing both host (CPU) and *kernel* (GPU) code. Each CUDA program consists of multiple phases that are executed on either CPU or GPU. All phases that exhibit little or no parallelism are implemented in CPU. Contrary, if the phases present much parallelism, they are coded as *kernel* functions in GPU. A *kernel* function defines the code to be executed by each thread launched in a parallel phase over GPU.

3 Sequential Permutation Index

Let $\mathcal{P}$ be a subset of the database U, $\mathcal{P} = \{p_1, p_2, \ldots, p_m\} \subseteq U$, that is called the permutants set. Every element x of the database sorts all the permutants according to the distances to them, thus forming a permutation of $\mathcal{P}$: $\Pi_x = \langle p_{i_1}, p_{i_2}, \ldots p_{i_m} \rangle$. More formally, for an element $x \in U$, its permutation Π_x of $\mathcal{P}$ satisfies $d(x, \Pi_x(i)) \leq d(x, \Pi_x(i+1))$, where the elements at the same distance

ISBN 978 -987-633-574-4

are taken in arbitrary, but consistent, order. We use $\Pi_x^{-1}(p_{i_j})$ for the *rank* of an element p_{i_j} in the permutation Π_x. If two elements are similar, they will have a similar permutation [2].

Basically, the permutation based algorithm is an example of probabilistic algorithm, it is used to predict proximity between elements, by using their permutations. The algorithm is very simple: In the offline preprocessing stage it is computed the permutation for each element in the database. All these permutations are stored and they form the index. When a query q arrives, its permutation Π_q is computed. Then, the elements in the database are sorted in increasing order of a similarity measurement between permutations, and next they are compared against the query q following this order, until some stopping criterion is achieved. The similarity between two permutations can be measured, for example, by *Kendall Tau*, *Spearman Rho*, or *Spearman Footrule* metrics [22]. All of them are metrics, because they satisfy the aforementioned properties. We use the Spearman Footrule metric because it is not expensive to compute and according to the authors in [2], and it has a good performance to predict proximity between elements. The Spearman Footrule distance is the *Manhattan distance* L_1, that belongs to the Minkowsky's distances family, between two permutations. Formally, Spearman Footrule metric F is defined as: $F(\Pi_x, \Pi_q) = \sum_{i=1}^{m} |\Pi_x^{-1}(p_i) - \Pi_q^{-1}(p_i)|$.

At query time we first compute the real distances $d(q, p_i)$ for every $p_i \in \mathcal{P}$, then we obtain the permutation Π_q, and next we sort the elements $x \in U$ into increasing order according to $F(\Pi_x, \Pi_q)$ (the sorting can be done incrementally, because only some of the first elements are actually needed). Then U is traversed in this sorted order, evaluating the distance $d(q, x)$ for each $x \in U$. For range queries, with radius r, each x that satisfies $d(q, x) \leq r$ is reported, and for k-NN queries the set of the k smallest distances so far, and the corresponding elements, are maintained. The database traversal is stopped at some point f, and the rest of the database elements are just ignored. This makes the algorithm probabilistic, as even if $F(\Pi_q, \Pi_x) < F(\Pi_q, \Pi_v)$ it does not guarantee that $d(q, x) < d(q, v)$, and the stopping criterion may halt the search prematurely. On the other hand, if the order induced by $F(\Pi_q, \Pi_x)$ is close to the order induced by the real distances $d(q, u)$, the algorithm performs very well. The efficiency and the quality of the answer obviously depend on f. In [2], the authors discuss a way to obtain good values for f for sequential processing.

4 GPU-Permutation Index

The GPU-CUDA system has two different steps: indexing and query resolution, they correspond whit two processes that have to be executed in sequence, first indexed process and next, query process. The Indexed process has two stages and the query process, four steps. The Figure 1 shows whole system.

Building a permutation index in GPU involves at least two steps. The first step (*Distance(O,P)*) calculates the distance among every object in database and the permutants. The second one (*Permutation Index(O)*) sets up the signatures of all objects in database, i.e. all object permutations. The process input is the database and the permutants. At the process end, the index is ready to be queried. The idea is to divide the work into threads blocks; each thread calculates the object permutation according to a global set of permutants.

In $Distances(O, P)$, the number of blocks will be defined according of the size of the database and the number of threads per block which depends of the quantity of resources required by each block. At the end, each threads block saves in the device memory its calculated distances. This stage requires a structure of

ISBN 978 -987-633-574-4

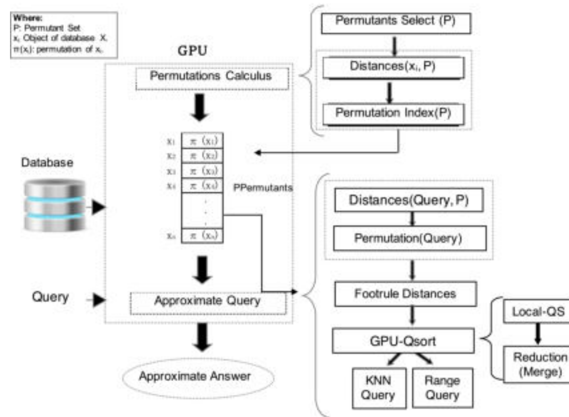


Fig. 1. Indexing and Querying in GPU-CUDA Permutation Index.

size $m \times n$ (m: permutants number, and n: database size), and an auxiliar structure in the shared memory of block (it stores the permutants, if the permutants size is greater than auxiliar structure size, the process is repeated). The second step ($Permutation\ Index(O)$) takes all calculated distances in the previous step and determines the permutations of each database object: its signature. To stablish the object permutation, each thread considers one database object and sorts the permutants according to their distance. The output of second step is the *Permutation Index*, which is saved in the device memory. Its size is $n \times m$.

The pemutation index allows to answer to all kinds of queries in approximated manner. Queries can be *"by range"* or *"k*-NN". This process implies four steps. In the first, the permutation of query object is computed. This task is carried out by so many threads as permutants exist. The next step is to contrast all permutations in the index with query permutation. Comparison is done through the $Footrule$ distance, one thread by each database object. In the third step, it sorts the calculated $Footrule$ distances. Finally, depending of query kind, the selected objects have to be evaluated. In this evaluation, the *Euclidean distance* between query object and each candidate element is calculated again. Only a database percentage is considered for this step, for example the 10% (it can be a parameter). If the query is by range, the elements in the answer will be those that their distances are less than reference range. If it is k-NN query, once each thread computes the *Euclidean distance*, all distances are sorted and the results are the first k elements of sorted list.

As sorting methodology, we implement the Quick-sort in the GPU, GPU-Qsort. The designed algorithm takes into account the highly parallel nature of GPUs. Its main characteristics are: iterative algorithm and heavy use of shared memory of each block, more details in[23].

By software and hardware characteristics, GPU allows us to think in to solve many approximated queries in parallel. The Figure 2 shows how the system is modified to solve many queries at the same time. In this Figure, you can observe that the Permutation Index is built once and then is used to answer all queries. In order to answer in parallel many approximate queries, GPU receives the queries set and it has to solve all of them. Each query, in parallel, applies the process explained in Figure 1. Therefore, the number of needed resources for this is equal to the amount of resources to compute one query multiplied by the

ISBN 978 -987-633-574-4

number of queries solved in parallel. This multiple-parallel computation involves a care management the blocks and their threads: blocks of different queries are accessed in parallel. Hence, it is important a good administration of threads. Each thread has to know which query it is solving and which database element is its responsibility. This is possible by establishing a relationship among *Thread Id*, *Block Id*, *Query Id*, and *Database Element*.

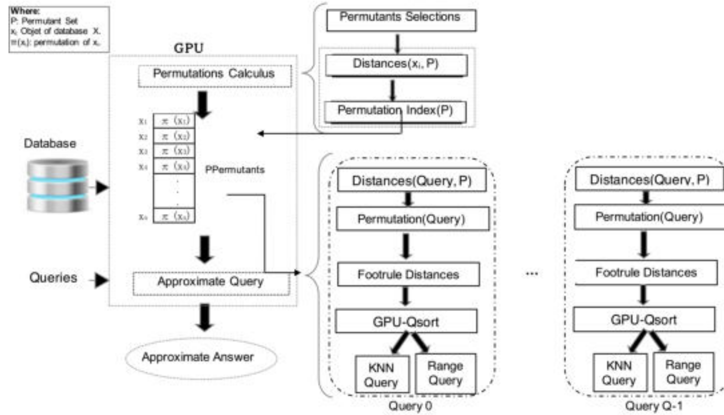


Fig. 2. Solving many queries in GPU-CUDA Permutation Index.

The number of queries to solve in parallel is determined according to the GPU resources, mainly its memory. If Q is the number of parallel queries, m the needed memory quantity per query and i the needed memory by the Permutation Index, $Q*m+i$ is the total required memory to solve Q queries in parallel. After Q parallel queries are solved, the results can be sent to CPU or they can be joined with other Q results and transfer them all together once via PCI-Express.

5 Experimental Results

Our experiments consider two metric databases selected from SISAP METRIC SPACE LIBRARY (www.sisap.org). The characteristics of each database are the following:

- English words(DBs): a set of English words. It uses the *Levenshtein* distance or *edit* distance.
- Colors histogram(DBh): a set of 112-vectors. It considers Euclidean distance.

In both cases, different DB size are considered, they are expressed in name: DBs or $DBh +$ <DBsize> in kB.

The hardware scenario was:

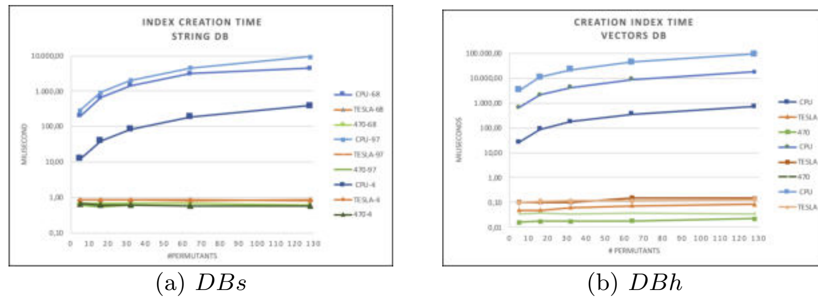
- CPU is an Intel(R) Xeon(R) CPU E5, 2603 v2 @1.80GHz x 8 and 15,6GB of memory.
- Two GPU are considered with the next characteristics (GPU Model, Memory, CUDA Cores, Clock Rate and Capability):
  - Tesla K20c, 4800 MB, 2496, 0.71 GHz, 3.5.

ISBN 978 -987-633-574-4

- GTX470, 1216 MB, 448, 1.22 GHz, 2.0.

The experiments consider for k-NN searches the values of k: 3 and 5; and for range the radii, for DBS: 1, 2, and 3, and DBh: 0,05, 0,08 and 0,13. For the parameter f of the Permutation Index, that indicates the fraction of DB revised during searches, we consider 10, 20, 30 and 50% of the DB size. The number of permutants used for the index are $5, 16, 32, 64$, and 128. In each case the results shown are the average over 1000 different queries.

In Figure 3, the Index Creation times for all the devices and for each BD are shown. We managed to increase the performance with respect to the CPU. Although only the times to build the index were taken into account in the time comparisons, the transfer time of the complete DB to the GPU was measured. In our case, both devices have the same PCI Express technology. For example the transfer time for BDs97 is 1.23 milliseconds. We can observe for the case of BDs, regarding the total creation time of the index, the load from the DB to the GPU implies 60% of the total process time in the Tesla K20c and 66% in the GTX 470.



(a) DBs (b) DBh

Fig. 3. Time of Index Creation for two DB.

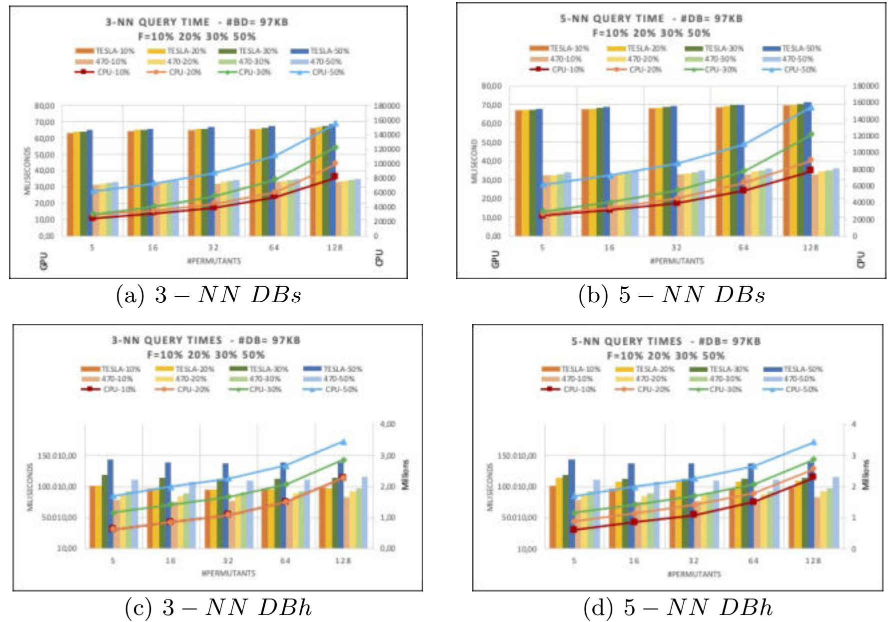
Figures 4 and 5 show the obtained time in k-NN and range queries respectively, for different parameters: permutants number, range, k, and DB percentage. In these results, 80 queries are solved in parallel. As it can be noticed *Range queries* show improvements respect to k-NN queries, but in both cases the achieved times are much less than CPU times. In all cases, it is clear the influence of DB size, but evenly we accomplish good performance. In all cases, the permutants number does not influence the time.

In Figure 6, we can see how the queries number to be solved in parallel influences the performance. Shorter times are achieved when queries number is greater. For BDh, the time to solve 1 (one) query vs 30 (thirty) queries decreases in the best of cases in the order of 1.8x (Tp1 / Tp30). In the case of the BDs, the gains obtained in solving multiple queries in parallel are greater: an improvement of 2.5x.

For the case of k-NN, the times are similar. This behavior is similar in both DB, i.e. the GPU resources have more work to do and, consequently, less idle time.

The trade-off between the answer quality and time performance of our parallel index with respect to the sequential index. For each k-NN or range query we have previously obtained the exact answer, that is $Rel()$, and we obtain the approximate answer $Retr()$. Figures 7 and 8 illustrate the average quality answer obtained for both kinds of queries, considering the Permutation Index respectively with 5,64 and 128 permutants, and different DB percentages. As it can

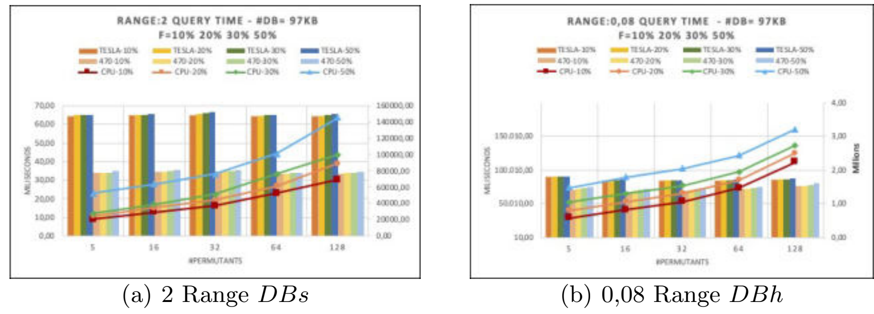
ISBN 978 -987-633-574-4



(a) $3-NN\ DBs$ (b) $5-NN\ DBs$

(c) $3-NN\ DBh$ (d) $5-NN\ DBh$

Fig. 4. k-NN Query Time for two DB.



(a) 2 Range DBs (b) 0,08 Range DBh

Fig. 5. Range Query Time for two DB.

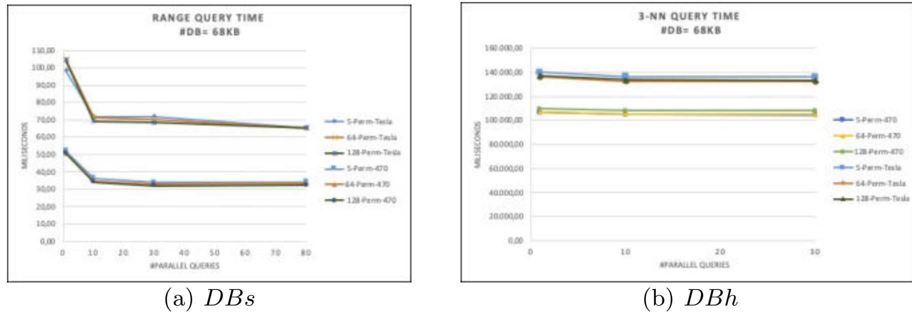
be noticed, the Permutation Index retrieves a good percentage of exact answer only reviewing a little fraction of the DB. For example, the 10% retrieves 40% and it needs to review the 30% to retrieve almost 80% of exact answer.

For lack of space, despite of we have tested another database sizes, we show only for the biggest database. In the other sizes have yielded similar results.

6 Conclusions

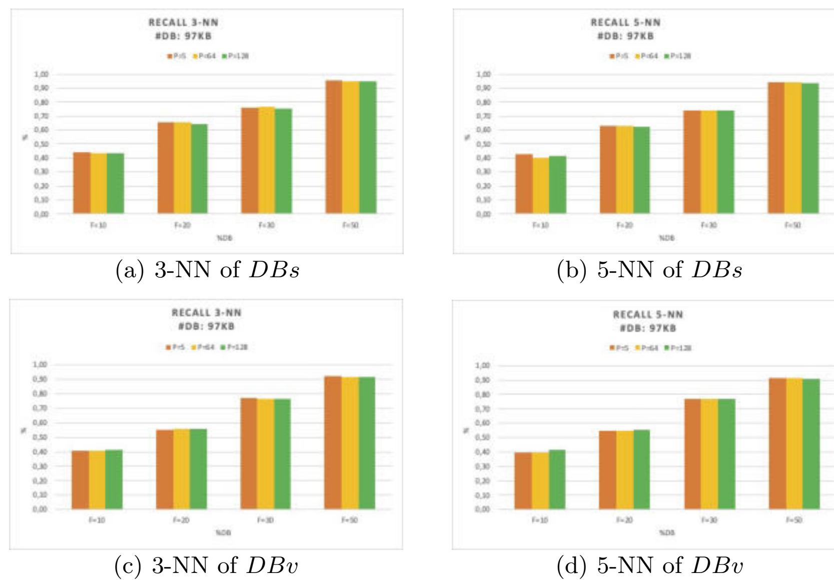
When we work with databases into large-scale systems such as Web Search Engines, it is not enough to speed up the answer time of only one query, but it is necessary to answer several queries at the same time. In this work, we present

ISBN 978 -987-633-574-4



(a) DBs (b) DBh

Fig. 6. Multi-Queries Time for two DB.



(a) 3-NN of DBs (b) 5-NN of DBs

(c) 3-NN of DBv (d) 5-NN of DBv

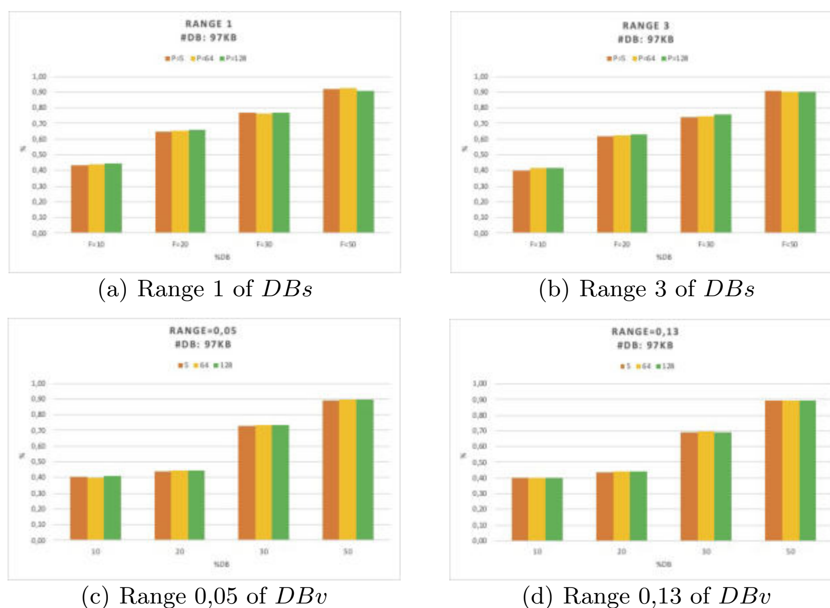
Fig. 7. Recall of approximate-k-NN queries for two DB.

a reliable solution to solve many queries in parallel, verifying the correctness of obtained results. This solution takes advantage of GPU and its high throughput: parallel processing for thousands of threads.

We check GPU-*Permutation Index* performance to the different GPUs. All accomplished performance results are very good, independently of the GPU architecture, because of careful planning and correct use of GPU resources. The index showed a good performance, allowing us to increase the fraction f of the database that will be examined to obtain better and accurate approximate results. An extensive validation process is carried out to guarantee the quality of the solution provided by the GPU.

In the future, we plan to make an exhaustive experimental evaluation, considering other types of databases and other solutions that apply GPUs to solve similarity searches in metric spaces; extend our proposal to other metric databases such as documents, DNA sequences, images, music, among others, and use other

ISBN 978 -987-633-574-4



(a) Range 1 of DBs

(b) Range 3 of DBs

(c) Range 0,05 of DBv

(d) Range 0,13 of DBv

Fig. 8. Recall of approximate-range queries for two DB.

distance functions. Another point to consider is to work with larger DBs, mainly when they are larger than the GPU memory. In this case, it is necessary to study strategies to partition the databases and/or use several GPUs. Besides, we plan to consider the Permutation's Signatures [24] to reduce the size of Permutation Index without removing any permutant.

References

1. E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín, "Searching in metric spaces," *ACM Comput. Surv.*, vol. 33, no. 3, pp. 273–321, 2001.
2. E. Chávez, K. Figueroa, and G. Navarro, "Proximity searching in high dimensional spaces with a proximity preserving order," in *Proc. 4th Mexican International Conference on Artificial Intelligence (MICAI)*, ser. LNAI 3789, 2005, pp. 405–414.
3. P. Ciaccia and M. Patella, "Approximate and probabilistic methods," *SIGSPATIAL Special*, vol. 2, no. 2, pp. 16–19, Jul. 2010. [Online]. Available: http://doi.acm.org/10.1145/1862413.1862418
4. P. Zezula, G. Amato, V. Dohnal, and M. Batko, *Similarity Search: The Metric Space Approach*, ser. Advances in Database Systems, vol.32. Springer, 2006.
5. P. Pacheco and M. Malensek, *An Introduction to Parallel Programming*. Elsevier Science, 2019. [Online]. Available: https://books.google.com.ar/books?id=uAfXnQAACAAJ
6. R. Robey and Y. Zamora, *Parallel and High Performance Computing*. Manning Publications, 2021. [Online]. Available: https://books.google.com.ar/books?id=jNstEAAAQBAJ
7. D. Kirk and W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. Elsevier Science, 2016. [Online]. Available: https://books.google.com.ar/books?id=wcS_DAAAQBAJ
8. R. Barrientos, F. Millaguir, J. L. Sánchez, and E. Arias, "Gpu-based exhaustive algorithms processing knn queries," *The Journal of Supercomputing*, vol. 73, pp. 4611–4634, 2017.

9. M. Kruliš, H. Osipyan, and S. Marchand-Maillet, "Employing gpu architectures for permutation-based indexing," *Multimedia Tools and Applications*, vol. 76, 05 2017.
10. S. Li and N. Amenta, "Brute-force k-nearest neighbors search on the gpu," in *Similarity Search and Applications*, G. Amato, R. Connor, F. Falchi, and C. Gennaro, Eds. Cham: Springer International Publishing, 2015, pp. 259–270.
11. P. Velentzas., M. Vassilakopoulos., and A. Corral., "In-memory k nearest neighbor gpu-based query processing," in *Proceedings of the 6th International Conference on Geographical Information Systems Theory, Applications and Management - GISTAM,*, INSTICC. SciTePress, 2020, pp. 310–317.
12. K. Figueroa, E. Chávez, G. Navarro, and R. Paredes, "Speeding up spatial approximation search in metric spaces," *ACM Journal of Experimental Algorithmics*, vol. 14, p. article 3.6, 2009.
13. B. Bustos and G. Navarro, "Probabilistic proximity searching algorithms based on compact partitions," *Discrete Algorithms*, vol. 2, no. 1, pp. 115–134, Mar. 2004. [Online]. Available: http://dx.doi.org/10.1016/S1570-8667(03)00067-4
14. K. Tokoro, K. Yamaguchi, and S. Masuda, "Improvements of tlaesa nearest neighbour search algorithm and extension to approximation search," in *Proceedings of the 29th Australasian Computer Science Conference - Volume 48*, ser. ACSC '06. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2006, pp. 77–83. [Online]. Available: http://dl.acm.org/citation.cfm?id=1151699.1151709
15. A. Singh, H. Ferhatosmanoglu, and A. Tosun, "High dimensional reverse nearest neighbor queries," in *The twelfth international conference on Information and knowledge management*, ser. CIKM '03. New York, NY, USA: ACM, 2003, pp. 91–98. [Online]. Available: http://doi.acm.org/10.1145/956863.956882
16. F. Moreno-Seco, L. Micó, and J. Oncina, "A modification of the laesa algorithm for approximated k-nn classification," *Pattern Recognition Letters*, vol. 24, no. 1–3, pp. 47 – 53, 2003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167865502001873
17. R. A. Baeza-Yates and B. A. Ribeiro-Neto, *Modern Information Retrieval - the concepts and technology behind search, Second edition.* Pearson Education Ltd., Harlow, England, 2011.
18. J. Cheng and M. Grossman, *Professional Cuda C Programming.* CreateSpace Independent Publishing Platform, 2017. [Online]. Available: https://books.google.com.ar/books?id=BcjItAEACAAJ
19. J. Han and B. Sharma, *Learn CUDA Programming: A beginner's guide to GPU programming and parallel computing with CUDA 10.x and C/C++.* Packt Publishing, 2019. [Online]. Available: https://books.google.com.ar/books?id=dhWzDwAAQBAJ
20. D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors, A Hands on Approach.* Elsevier, Morgan Kaufmann, 2010.
21. NVIDIA, "Nvidia cuda compute unified device architecture, programming guide," in *NVIDIA*, 2020.
22. R. Fagin, R. Kumar, and D. Sivakumar, "Comparing top k lists," in *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, ser. SODA '03. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003, pp. 28–36. [Online]. Available: http://dl.acm.org/citation.cfm?id=644108.644113
23. M. Lopresti, N. Miranda, F. Piccoli, and N. Reyes, "Permutation Index and GPU to Solve efficiently Many Queries," in *VI Latin American Symposium on High Performance Computing, HPCLatAm 2013*, 2013, pp. 101–112.
24. K. Figueroa and N. Reyes, "Permutation's signatures for proximity searching in metric spaces," in *Similarity Search and Applications*, G. Amato, C. Gennaro, V. Oria, and M. Radovanović, Eds. Cham: Springer International Publishing, 2019, pp. 151–159.