Evaluating tradeoff between recall and performance of GPU Permutation Index

Mariela Lopresti, Natalia Miranda, Mercedes Barrionuevo,
Fabiana Piccoli, Nora Reyes

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

Abstract. Query-by-content, by means of similarity search, is a fundamental operation for applications that deal with multimedia data. For this kind of query it is meaningless to look for elements exactly equal to a given one as query. Instead, we need to measure the dissimilarity between the query object and each database object. This search problem can be formalized with the concept of metric space. In this scenario, the search efficiency is understood as minimizing the number of distance calculations required to answer them. Building an index can be a solution, but with very large metric databases is not enough, it is also necessary to speed up the queries by using high performance computing, as GPU, and in some cases is reasonable to accept a fast answer although it was inexact. In this work we evaluate the tradeoff between the answer quality and time performance of our implementation of *Permutation Index*, on a pure GPU architecture, used to solve in parallel multiple approximate similarity searches on metric databases.

1 Introduction

Similarity search is a fundamental operation for applications that deal with multimedia data. For a query in a multimedia database it is meaningless to look for elements exactly equal to a given one as query. Instead, we need to measure the similarity (or dissimilarity) between the query object and each object of the database. The similarity search problem can be formally defined through the concept of metric space. The metric space model is a paradigm that allows to model 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 which make it a metric. Given a dataset of n objects, a query can be trivially answered by performing n distance evaluations, but sequential scan does not scale for large problems. The reduction of number of distance evaluations is important to achieve better results. Therefore, in many cases preprocessing the dataset is a good option to solve queries with as few distance computations as is 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 a good representative of 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]. Inexact similarity searching is reasonable in many applications because the metric-space modelizations already involve an approximation to reality; hence, a second approximation at search time is usually acceptable.

Moreover, for very large metric database is not enough to preprocess the dataset by building an index, it is also necessary to speed up the queries by using high performance computing (HPC). In order to employ HPC to speedup the preprocess of the dataset to obtain an index, and to answer posed queries, the Graphics Processing Unit (GPU) represents a good alternative. The GPU is attractive in many application areas for its characteristics, especially because of its parallel execution capabilities and fast memory access. 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 [5–9] differents proposal are made, all of them are improvements to brute force algorithm (sequential scan) to find the k-NN of a query object.

The goal of this work is to analyze the tradeoff betwen the quality of similarity queries answer and time performance, using a parallel permutation index implemented on GPU. In this analysis particularly we consider the known measures from information retrieval area for answer quality and we consider the achieved performance of our parallel implementation of *Pemutation Index*.

The paper is organized as follows: the 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 $d : X \times X \rightarrow R^+$ 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 $(q, r)_d$ is to retrieve all the objects $x \in U$ within the radius r of the query q (i.e. $(q, r)_d = \{x \in U / d(q, x) \leq r\}$). 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 $\forall x \in k\text{-NN}(q), v \in U \wedge v \notin k\text{-NN}(q), d(q, x) \leq d(q, v)$.

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 element pairs from the database. This is usually avoided because $O(n^2)$ space is unacceptable for realistic applications [10].

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 these 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 [11]. Probabilistic algorithms on the other hand state that the answer is correct with high probability. 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. This relaxation gives faster algorithms as the threshold ϵ increases [11, 12]. Probabilistic algorithms on the other hand state that the answer is correct with high probability [13, 14].

2.1 Quality Measures of Approximate Search

As it is aforementioned, an approximate similarity searching can obtain an inexact answer. 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 $k\text{-NN}(q) = \{x_1, x_2, \ldots, x_k\}$, it determines the radius $r_k = \max_{1 \leq i \leq k}\{d(x_i, q)\}$ needed to enclose these k closest elements to q. An approximate answer of k-NN(q) could obtain some elements z whose $d(q, z) > r_k$. For the other hand, an approximate range query of $(q, r)_d$ 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 $(q, r)_d$.

In most of information retrieval (*IR*) systems it is necessary to evaluate retrieval effectiveness [15]. The judgements of document relevance used to evaluate effectiveness have some problems of subjectivity and unrealiableness. That is, different judges will assign different relevance values to a document retrieved in response to a given query. The seriousness of the problem is the subject of debate, with many IR researchers arguing that the relevance judgment reliability problem is not sufficient to invalidate the experiments that use relevance judgments. Many measures of retrieval effectiveness have been proposed. The most commonly used are *recall* and *precision*.

Recall is the ratio of relevant documents retrieved for a given query over the number of relevant documents for that query in the database. *Precision* is the ratio of the number of relevant documents retrieved over the total number of documents retrieved. Both recall and precision take on values between 0 and 1.

Since one often wishes to compare IR performance in terms of both recall and precision, methods for evaluating them simultaneously have been developed

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 the k elements obtained which are relevant to q by verifying if $d(q, y_i) \leq r_k$; 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 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 [16]. The parallel programming over GPUs has many differences from parallel programming in typical parallel computer, the most relevant are: *The number of processing units*, *CPU-GPU memory structure* and *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.

The Compute Unified Device Architecture (CUDA), supported from the NVIDIA Geforce 8 Series, enables to use GPU as a highly parallel computer for non-graphics applications [16, 17]. 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. It has multiple streaming multiprocessors (SMs), each of them contains several (eight, thirty-two or forty-eight, depending GPU architecture) scalar processors (SPs). 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 data parallelism are implemented in CPU. Contrary, if the phases present much data 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.

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 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 [18]. 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] 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 that 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 Figure 1 shows the GPU-CUDA system to work with a permutation index: the processes of indexing and querying. The Indexing process has two stages and the Querying process four steps. In this last process, we pay special attention to one step: the sorting.

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

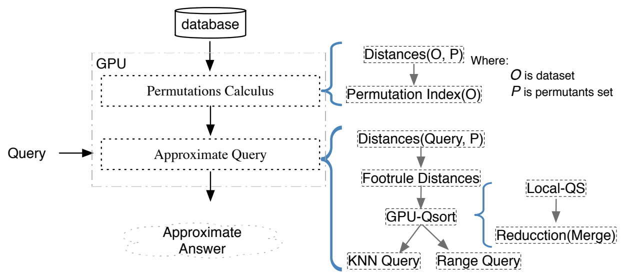


Fig. 1. Indexing and Querying in GPU-CUDA permutation index.

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 process end, the index is ready to be queried. The idea is to divide the work in threads blocks, each thread calculates the object permutation according to a global permutants set.

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 save in device memory its calculated distances. This stage requires a structure of 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 object in database: its signature. To stablish the object permutation, each thread considers an object in database and sorts the permutants according to their distance. The output of second step is the *Permutation Index*, which is saved in 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 object in database. 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 graphics processors (GPUs) and the CUDA capabilities 1.2 or higher. Its main characteristics are: iterative algorithm and heavy use of shared memory of each block, you can find an detailed description in[19].

In large-scale systems such as Web Search Engines indexing multimedia content, it is critical to deal efficiently with streams of queries rather than with single queries. Therefore, it is not enough to speed up the time to answer only one query, but it is necessary to leverage the capabilities of the GPU to parallely answer several queries. So we have to show how to achieve efficient and scalable performance in this context. We need to devise algorithms and optimizations specially tailored to support high-performance parallel query processing in GPU. GPU has characteristics of software and hardware which allow us to think in to solve many approximated queries in parallel. The represented system in Figure 1 is modified and it is shown in Figure 2. In this, it can be observed that the permutation index is built once and then is used to answer many queries.

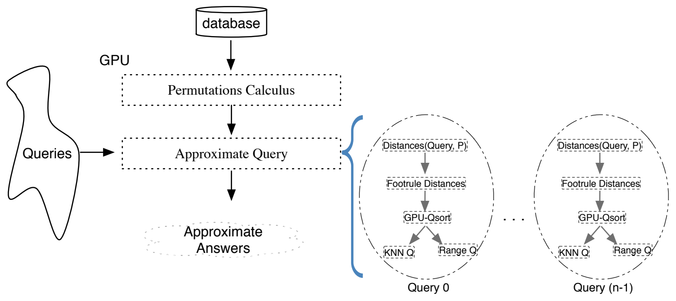


Fig. 2. Solving many queries in GPU-CUDA permutation index.

In order to answer parallely 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 1, therefore the number of needed resources for this is equal to the resources amount to compute one query multiplied the number of queries solved in parallel. The number of queries to solve in parallel is determined according to the GPU resources mainly its memory. If Q are parallel queries, m the needed memory quantity per query and i the needed memory by permutation index, $Q * m + i$ is the total required memory to solve Q queries in parallel. Once the Q parallel queries are solved, the results are sent from the GPU to the CPU through a single transfer via PCI-Express.

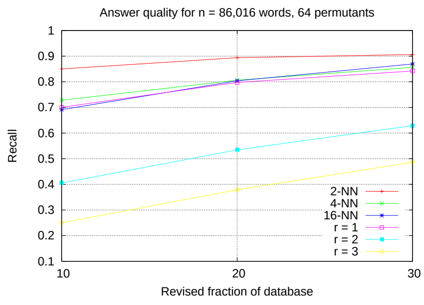
Solving many queries in parallel involves carefully manage the blocks and their threads. At the same time, blocks of different queries are accessed in parallel. Hence it is important a good administration of threads: which query it is solved and which database element it is responsible. The task is possible by establishing a relationship among *Thread Id*, *Block Id*, *Query Id*, and *Database Element*.

5 Experimental Results

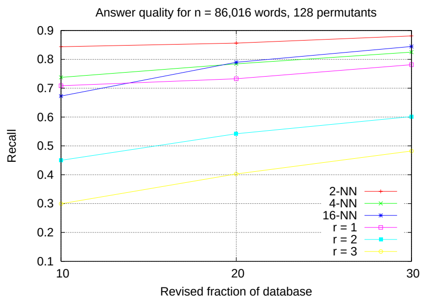
Our experiments considered a metric database of 86,016 English words and using the *Levenshtein* distance, also called *edit* distance [20]. The analysis was made for a GeForce GPU GTX520MX whose characteristics (Global Memory: 1024MB, SM:1, SP:48, Clock rate:1.8GHz, Compute Capability: 2.1). The CPU is an Intel core i3, 2.13 GHz and 3 GB of memory.

The experiments consider for k-NN searches the values of k: 2, 4, and 16; and for range the radii: 1, 2, and 3. For the parameter f of the Permutation Index, that indicates the fraction of database revised during searches, we consider 10%, 20%, and 30% of the database size. The number of permutants used for the index are 64 and 128. In each case the results shown are the average over 1000 different queries and 80 solved queries in parallel. In this paper, we do not display the speed up of construction of *Permutation Index*. These results are illustrated in [21].

Our focus is to evaluate the tradeoff 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 $Retr()$. Figure 3 illustrates the average quality answer obtained for both kinds of queries, considering the Permutation Index respectively with 64 (Figure 3(a)) and 128 (Figure 3(b)) permutants. As it can be noticed, the Permutation Index retrieves a high percentage of exact answer only reviewing a little fraction of the database. For example, the 10% retrieves 85% for 2-NN queries both with 64 and 128 permutants. It needs to review the 20% to retrieve almost 80% of exact answer for $k = 4$ and $k = 16$ with 64 and 128 permutants. The effectiveness in range queries decreases as the radius grows. For $r = 1$ the index retrieves almost 80% of the relevant objects.



(a) 64 permutants

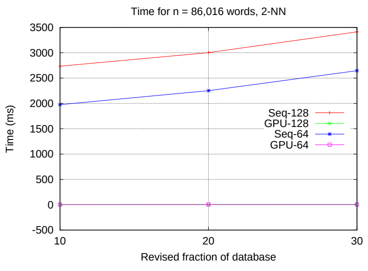


(b) 128 permutants

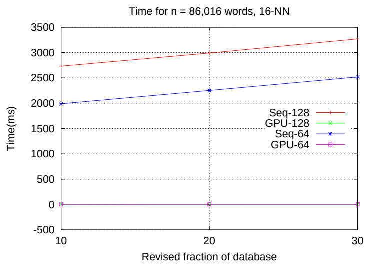
Fig. 3. Recall of approximate-k-NN and range queries obtained with Permutation Index.

Figures 4 and 5 show the obtained times by k-NN and range queries for three f values and all the number of permutants considered. In these results, 80 queries are solved in parallel on GPU. As it can be noticed, the parallel times are so smaller than the corresponding sequential times. In both types of queries the achieved speed up is very good, it can be observed the same behavior for all options of our parallel solution, they are independent of the number of permutants and fraction f of database to be revised.

For lack of space, despite of we have tested another database sizes, we show only the results for 86,016 elements, but the other sizes have yielded similar results.

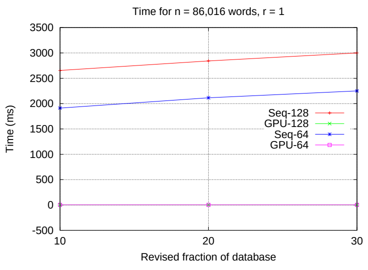


(a) 2-NN

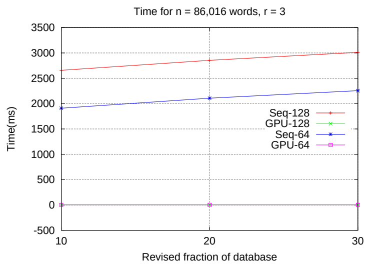


(b) 16-NN

Fig. 4. Time of k-NN queries obtained with Sequential and Parallel Permutation Index.



(a) Range query r = 1



(b) Range query, r = 3

Fig. 5. Time of Range queries obtained with Sequential and Parallel Permutation Index.

6 Conclusions

When we work with databases, there are different realities where it is not enough to speed up the time to answer only one query, but it is necessary to solve several queries at the same time. In this work we present a solution to solve many queries in parallel taking advantage of GPU architecture: it is a massively parallel architecture, it has a high throughput because its capacity of parallel processing for thousands of threads, and verify the correctness of obtained results.

The implemented GPU-*Pemutation Index* showed a good performance, allowing us to increase the fraction f of database that will be examined to obtain better and accurate approximate results. This affirmation is made in function of an extensive validation process carried out to guarantee the quality of the solution provided by the GPU.

As future task, we need to validate every performance parameters: recall, speed up and throughput, with other kinds of database, comparing with other solutions that apply GPU in the scenario of metric space approximate searches.

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. Chavez, K. Figueroa, and G. Navarro, “Effective proximity retrieval by ordering permutations,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 30, pp. 1647–1658, 2008.
3. P. Ciaccia and M. Patella, “Approximate and probabilistic methods,” *SIGSPATIAL Special*, vol. 2, no. 2, pp. 16–19, Jul. 2010.
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. R. J. Barrientos, J. Gomez, C. Tenllado, M. Prieto, and M. Marin, “knn query processing in metric spaces using gpus,” in *17th International European Conference on Parallel and Distributed Computing*, L. N. i. C. S. Springer, Ed., vol. 6852, 2011, pp. 380–392.
6. V. Garcia, E. Debreuve, F. Nielsen, and M. Barlaud, “k-nearest neighbor search: fast GPU-based implementations and application to high-dimensional feature matching,” in *IEEE Intern. Conf. on Image Processing*, Hong Kong, Sept. 2010.
7. K. Kato and T. Hosino, “Solving k-nearest neighbor problem on multiple graphics processors,” in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGRID*, ACM, Ed., 2010, pp. 769–773.
8. S. Liang, Y. Liu, C. Wang, and L. Jian, “Design and evaluation of a parallel k-nearest neighbor algorithm on CUDA-enabled GPU,” in *IEEE 2nd Symposium on Web Society (SWS)*, 2010, pp. 53 – 60.
9. R. Uribe, P. Valero, E. Arias, J. L. Sánchez, and D. Cazorla, “A GPU-Based Implementation for Range Queries on Spaghettis Data Structure,” in *ICCSA (1)*, ser. Lecture Notes in Computer Science, vol. 6782. Springer, 2011, pp. 615–629.
10. 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.
11. B. Benjamin and G. Navarro, “Probabilistic proximity searching algorithms based on compact partitions,” *Discrete Algorithms*, vol. 2, no. 1, pp. 115–134, Mar. 2004.
12. 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.
13. A. Singh, H. Ferhatosmanoglu, and A. Tosun, “High dimensional reverse nearest neighbor queries,” in *The 12th intern. conf. on Information and knowledge management*, ser. CIKM ’03. New York, NY, USA: ACM, 2003, pp. 91–98.
14. F. Moreno, L. Mic, and J. Oncina, “A modification of the laesa algorithm for approximated k-nn classification,” *Pattern Recognition Letters*, vol. 24, no. 13, pp. 47 – 53, 2003.
15. 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.
16. D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors, A Hands on Approach*. Elsevier, Morgan Kaufmann, 2010.
17. NVIDIA, “Nvidia cuda compute unified device architecture, programming guide version 4.2.” in *NVIDIA*, 2012.
18. R. Fagin, R. Kumar, and D. Sivakumar, “Comparing top k lists,” in *Proc. of the 40th annual ACM-SIAM symposium on Discrete algorithms, SODA ’03*. Philadelphia, USA: Society for Industrial and Applied Mathematics, 2003, pp. 28–36.
19. 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.
20. V. I. Levenshtein, “Binary codes capable of correcting spurious insertions and deletions of ones,” *Problems of Information Transmission*, vol. 1, pp. 8–17, 1965.
21. M. Lopresti, N. Miranda, F. Piccoli, and N. Reyes, “Efficient similarity search on multimedia databases,” in *XVIII Congreso Argentino de Ciencias de la Computacin, CACIC 2012*, 2012, pp. 1079–1088.