Solving Multiple Queries through a Permutation Index in GPU

Mariela Lopresti, Natalia Miranda, Fabiana Piccoli, and Nora Reyes

LIDIC. Universidad Nacional de San Luis,
Ejército de los Andes 950 - 5700, San Luis,
Argentina

{omlopres, ncmiran, 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 the one given as query. Instead, we need to measure dissimilarity between the query object and each database object. The metric space model is a paradigm that allows modeling all similarity search problems. Metric databases permit to store objects from a metric space and efficiently perform similarity queries over them, in general, by reducing the number of distance evaluations needed. Therefore, the goal is to preprocess a particular dataset in such a way that queries can be answered with as few distance computations as possible. Moreover, for a very large metric database it is not enough to preprocess the dataset by building an index, it is also necessary to speed up the queries via high performance computing using GPU. In this work we show an implementation of a pure GPU architecture to build a *Permutation Index* used for approximate similarity search on databases of different data nature and to solve many queries at the same time. Besides, we evaluate the tradeoff between the answer quality and time performance of our implementation.

Keywords. Metric space, approximate similarity search, permutation index, high performance computing, GPU.

Resolución de múltiples consultas usando índice de permutación en GPU

Resumen. Realizar consultas por contenido, a través de búsquedas de similitud, es una operación fundamental para aplicaciones relacionadas con datos multimedia. En este tipo de consultas no tiene sentido buscar elementos exactamente iguales a uno dado como consulta. En su lugar, es necesario medirla disimilitud entre el objeto de consulta y cada objeto de la base de datos. El modelo de espacio métrico es un paradigma que permite modelar todos los problemas de búsqueda por similitud. Las bases de datos métricas permiten el almacenamiento de objetos de un espacio métrico y responder consultas por similitud de manera eficiente, generalmente, mediante la reducción del número de evaluaciones de distancia. En consecuencia, el objetivo es pre-procesar el conjunto de datos de manera que las consultas pueden ser respondidas con el menor número posible de cálculos de distancia. Más aún, para grandes bases de datos métricas no basta con procesar previamente el conjunto de datos mediante la creación de un índice, también es necesario acelerar las consultas mediante el uso de computación de alto desempeño, una alternativa es utilizar GPU. En este trabajo se muestra una implementación de una arquitectura de GPU pura para construir el *Pemutation Index*, el cual nos permite resolver en paralelo múltiples consultas por similitud aproximadas en bases de datos de diferente naturaleza. Además se evalúa el compromiso entre la calidad de respuesta y el desempeño de nuestra aplicación. Finalmente se presentan resultados experimentales.

Palabras clave. Espacios métricos, búsquedas aproximadas por similitud, índice de permutación, computación de alto desempeño, GPU.

1 Introduction

Due to an increasing interest in manipulating and retrieving multimedia data, nowadays the problem of similarity search receives much attention. The metric space model is a paradigm that allows modeling all similarity search problems. 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 on them. The distance function determines the similarity (or dissimilarity) between two given objects and satisfies several properties which make it a metric. Given a dataset of $\mid U \mid = n$ objects, a query can be trivially answered by performing n distance evaluations, but sequential scan does not scale for large problems. To reduce the number of distance evaluations is important for achieving better results. Therefore, in many cases preprocessing the dataset is a good option to solve queries with as few distance computations as possible. An index helps to retrieve the objects from U that are relevant to the query by making much less than n distance evaluations during searches [6]. One of these indices is the *Permutation Index* [5].

Moreover, for a very large metric database it 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 speed up 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 due to 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.

A GPU computing system consists of two basic components: the traditional CPU and one or more GPUs (Streaming Processor Array). The GPU can be considered as a manycore coprocessor able to support fine grain parallelism (a lot of threads run in parallel, all collaborating in the solution of the same problem) [14, 19]. GPU is different from other parallel architectures because it shows flexibility in local resource allocation to the threads. In general, a GPU multiprocessor consists of several stream multiprocessors, each of them having multiple processing units, records and on-chip memory. Each stream multiprocessor can run a variable number of threads. There are many tools to program the GPU, CUDA is one of them.

CUDA is a standard C/C++ extended by several keywords and constructs. Its programming model is SPMD (Single Process-Multiple Data) with two main characteristics: parallel work through concurrent threads and memory hierarchy. A CUDA program consists of multiple phases executed on either CPU or GPU.

In metric spaces, indexing and query resolution are the most common operations. They have several aspects that accept optimizations through the application of high performance computing techniques. There are many parallel solutions for some metric space operations implemented for GPU. Querying by k-NN has attracted the greatest attention of researchers in the area, so there are many solutions which consider GPU. In [1, 4, 11, 13, 15] different proposals were made, all of them are improvements of the brute force algorithm (sequential scan) to find the k-NN of a query object. The improvements differ in which process part is parallelized or which methodology is applied.

Beside different parallel implementations of scan sequential, there are other proposals [1, 2, 24] which implement solutions for metric indices: List of Clusters, SSS-Index and Spaguettis index.

In all previous research, the authors report benefits which are strictly linked to the characteristics and architecture of the GPU.

The rest of the paper is organized as follows: Section 2 describes all previous concepts necessary to understand our work and state of art in the use of GPU to accelerate metric indices; Section 3 introduces the sequential version of *Permutation Index*; Sections 4, 5, and 6 describe the characteristics of our proposal and its empirical performance. Finally, the conclusions and future work are presented.

2 Previous Concepts

In this section, we explain the main concepts necessary to develop this work.

2.1 Metric Space, Queries and Index

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 searching techniques of interest [6, 25, 21]: Range Searching and the k Nearest Neighbors (k-NN). The goal of the range search $(q, r)_d$ is to retrieve all objects $x \in U$ within the radius r from 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 $|$

$k - \mathsf{NN}(q) \mid = k$ and $\forall x \in k - \mathsf{NN}(q), v \in U \wedge v \notin k - \mathsf{NN}(q), d(q,x) \leq d(q,v)$.

When the 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 information saved in the index may vary, some indices store a subset of distances between objects, and others maintain only 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. More information an index stores (it uses more memory), lower query cost it obtains. However, there are some indices that use memory better than others. Therefore in a database of n objects, the largest quantity of 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 is fulfilled in two stages: preprocessing and query time. During the preprocessing stage an index is built. It is used during the query time to avoid some distance computations. Basically the state of the art in this area can be divided into two families [6]: *pivot-based algorithms* and *compact partition-based algorithms*. In the first case, the index consists of a set of pivots $\{p_1, \ldots, p_m\} \subseteq U$, which computes and keeps (in a data structure, usually like a tree) some (or all) distances $\{d(p_1,x), d(p_2,x), \ldots, d(p_m,x)\}, x \in U$. Queries are solved considering all pivots. In the second case, the space is divided into small compact zones. A set of objects, called *centers*, $\{c_1, \ldots, c_s\} \subseteq U$ is chosen, and the rest of the elements is distributed into s zones defined in different ways by the centers c_i. The index is composed by the centers, the elements of each zone, and often some additional distances.

There is an alternative to "exact" similarity searching called approximate similarity searching [7], where accuracy or determinism is traded for faster searches [6][25][21][20], it encompasses *approximate* and *probabilistic algorithms*. The goal of approximate similarity search is to *significantly* reduce search times by allowing some errors in the query outcome.

In approximate algorithms one usually has a threshold ϵ as a 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 [3]. On the other hand, probabilistic algorithms state that the answer is correct with high probability. Some examples are [22, 17]. In the next section we detail a probabilistic method called *Permutation Index* [5].

2.2 GPGPU

Mapping general-purpose computation onto GPU implies a use of 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 [14, 19]. Parallel programming over GPUs has many differences from parallel programming in a typical parallel computer, the most relevant are *the number of processing units*, *CPU-GPU memory structure* and *the number of parallel threads*.

Every GPGPU program has many basic steps. First, the input data is transferred 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 must be copied back to the host main memory.

Not all kind of problems 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 by the NVIDIA Geforce 8 Series, is able to use GPU as a highly parallel computer for non-graphics applications [14, 18]. 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 (8, 32 or 48, depending on the 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 which are executed on either CPU or GPU. All phases that exhibit little or no data parallelism are implemented in CPU. Otherwise, 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 the parallel phase.

GPU computation considers a hierarchy of abstraction layers: $grid$, $blocks$ and $threads$. The $threads$, basic execution unit that executes the *kernel* function, in the CUDA model are grouped into $blocks$. All threads in a block function on one SM and communicate among each other through the *shared memory*. Threads in different blocks can communicate through the *global memory*. Beside shared and global memory, threads have their local variables. All $thread-blocks$ form a $grid$. The number of grids, blocks per grid and threads per block are parameters fixed by the programmer and adjustable to improve performance.

With respect to memory hierarchy, CUDA threads may access data from multiple memory spaces during their execution. Each thread has private local memory and each block has shared memory visible to all its threads. These memories have the same lifetime that the *kernel*. All threads have access to the same global memory and two additional read-only memory spaces: the constant and texture memory spaces which are optimized for different memory usages. The global, constant and texture memory spaces are persistent across launched *kernel* by the same application. Each kind of memory has its own access cost, and the global memory accesses are the most expensive.

3 Sequential Permutation Index

Let $\mathcal{P}$ be a subset of the database U, $\mathcal{P} = \{p_1, p_2, \ldots, p_m\} \subseteq U$, and it is called the permutant 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 chosen in an 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 [5].

Basically, the permutation-based algorithm is an example of a probabilistic algorithm; it is used to predict proximity between elements by using their permutations. The algorithm is very simple. At the offline preprocessing stage, it computes the permutation for each element in the database. All these permutations are stored and form the index. When a query q arrives, its permutation Π_q is computed. Then, the elements in the database are sorted in the increasing order of the similarity measure between permutations, then they are compared against the query q following the specified 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 [8]. 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 [5] 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 distance family, which estimates the distance between two permutations. Formally, Spearman Footrule metric F is defined as $F(\Pi_x, \Pi_q) = \sum_{i=1}^{m} |\Pi_x^{-1}(p_i) - \Pi_q^{-1}(p_i)$.

This distance $F(\Pi_x, \Pi_q)$ can be computed in $O(m)$ time [8]. Therefore, during the preprocessing phase we first compute the mn distances $d(x, p_i)$, and then compute and sort all the permutations for each element x in the database. This stage costs $O(nm \log m)$ of additional time, and requires $O(nm \log m)$ bits to store the whole index.

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. Next, we sort the elements $x \in U$ in the increasing order according to $F(\Pi_x, \Pi_q)$ (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 calculated so far and the corresponding elements are maintained.

Algorithm 1 shows the process for a range query. The efficiency and the quality of the answer obviously depend on database fraction f: as f grows, the efficiency degrades, but the answer quality improves. A way to obtain good values for f is discussed in [5].

```
RangeQuery(element $q$, radius $r$, database fraction $f$)
1.  Let $A[1,n]$ be an array of tuples and $U = \{x_1, \ldots, x_n\}$
2.  Compute $\Pi_q^{-1}$
3.  For $i \leftarrow 1$ to $n$ do
4.      $A[i] \leftarrow \langle x_i, F(\Pi_{x_i}, \Pi_q)\rangle$
5.  SortIncreasing($A$) /* by second component of tuples */
6.  For $i \leftarrow 1$ to $fn$ do
7.      $\langle x, s\rangle \leftarrow A[i]$
8.      If $d(q,x) \leq r$ Then Report $x$
```

Algorithm 1. Range query of q with radius r in a permutation index

4 GPU-CUDA Permutation Index

Fig. 1 shows how the GPU-CUDA system works with a permutation index, presenting the processes of indexing and querying. The indexing process has two stages, and the querying process includes four steps. In this last process, we pay special attention to a particular step, which is sorting. The next sections detail the characteristics of each process, their steps and peculiarities.

4.1 Building a Permutation Index

Building a permutation index in GPU involves at least two steps. The first step calculates the distance among all objects in the database and the permutants. The second step sets up the signatures of all objects in the database, i.e., all object permutations. The process input is the database and the permutants. When the process ends, the index is ready to be queried. The idea is to divide the work in thread blocks; each thread calculates the permutation object according to the global permutant set.

In the first task ($Distances(O,P)$), the number of blocks will be defined according to the size of the database and the number of threads per block which depends on the quantity of resources required by each block. At the step end, each thread block saves its calculated distances in the device memory. This stage requires a structure of size $M \times N$ (M: permutant number and N: database size) and an auxiliar structure of fixed size defined in the shared memory of a block (it stores the permutants, if the permutant size is greater than auxiliar structure size, the process is repeated until all distances to permutants are calculated).

The second step ($Permutation\ Index(O)$) takes all distances calculated in the previous step and determines the permutations of each object in the database: its signature. To establish the object permutation, each thread considers an object in the database and sorts the permutants according to their distance. The output of the second step is the *Permutation Index*, which is saved in the device memory. Its size is $N \times M$.

4.2 Solving Approximate Queries

The permutation index allows answering all kinds of queries in an approximated manner. Queries can be *"by range"* or *"k-NN"*. This process implies four steps. In the first step, the permutation of the query object is computed. This task is carried out by as many threads as there are permutants. The task of the next step is to compare all permutations in the index with the query permutation. Such comparison is done through the $Footrule$ distance, one thread by object in the database. In the third step, the calculated $Footrule$ distances are sorted. As sorting methodology, we implement Quick-sort in the GPU; its characteristics are explained bellow. Finally, depending on the query type, the selected objects have to be evaluated. During this evaluation, the *Euclidean distance* between the query object and each candidate element is calculated again. Only the specified percentage of the database is considered at this step, for example the 10% (it can be a parameter). If the query is by range, the elements in the answer will be such that their distances are less than the reference range. If it is k-NN query, once each thread computes the *Euclidean distance*, all distances are sorted (using GPU-Qsort) and the results are the first k elements of the sorted list.

Considering the sorting algorithm, we describe a parallel Quicksort algorithm for GPU, called GPU-Qsort. The designed algorithm takes into account the highly parallel nature of graphics processors (GPUs) and the CUDA capabilities of 1.2 or higher.

GPU-Qsort carries out the task in two stages: *Local-Qsort* and *Merge-Reduction*. The first stage, **Local-Qsort**, has a data sequence as input and its output is N sorted subsequences. Each subsequence is ordered by a thread block according to the iterative quick sort. Therefore, there are N thread blocks, where the number of threads by block is fixed and determined in relation to the resources required by the block. Each block chooses a local pivot (it has to belong to the input data list of the block) and divides the data sequence in two subsequences: one

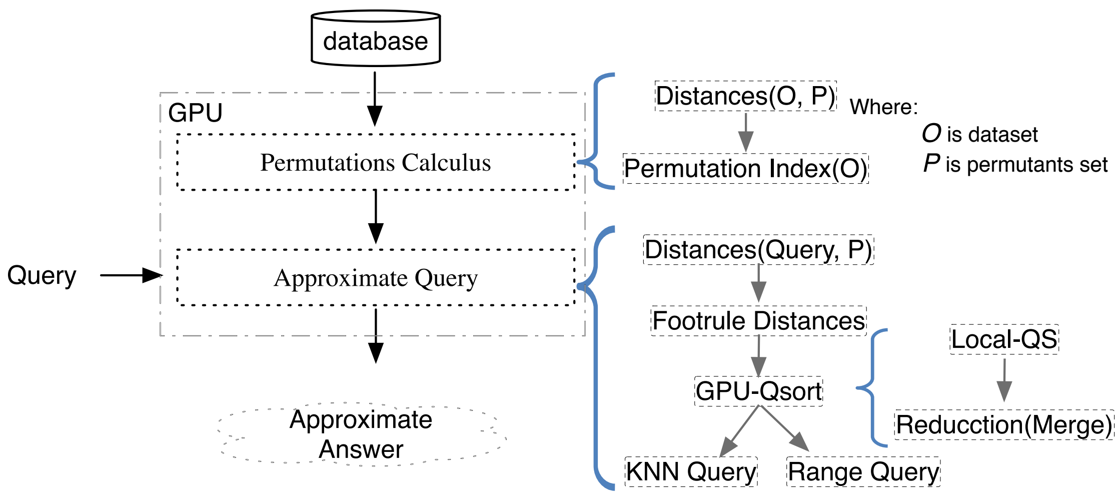


Fig. 1. Indexing and querying in GPU-CUDA permutation index

has the elements smaller than the pivot and another has the elements greater or equal than the pivot. The pivot is the median among three elements of data subsequence: the first, the middle and the last element [23]. Each block works independently of other blocks thus eliminating the need of synchronization among threads of different blocks. Based on the selected pivot, all elements lower than the pivot are moved to the left of the pivot, and the elements which are greater or equal to the pivot are shifted to the pivot's right. The task is fulfilled by using shared memory and each thread can determine the position for its element in the shared structure (using CUDA atomic functions).

The process is applied iteratively to two subsequences using a stack. The stack saves all subsequences that still remain to be sorted. When there are two ready subsequences to be processed, one is selected and the other is pushed in the stack. When one subsequence is sorted, the subsequence on the top of the stack is selected for processing. The iterative process ends when the stack is empty and the list is sorted. When the number of elements in the sequence is less than eight, it is sorted in a sequential manner, because the process overhead is too large compared to the sequence size.

At the end of this stage, each one of N blocks copies its sorted subsequence to the device memory. The output is the N sorted subsequence.

For the second stage of GPU-Qsort, **Merge-Reduction**, its input is the N sorted list and the output is the whole sorted sequence. This phase makes a reduction; the reduction operation is a merge of sorted lists. A block merges two lists at a time. Therefore, $log_2 N$ iterations are necessary to find the final result. This stage requires $\lceil \frac{N}{2} \rceil$ blocks with 32 threads per each block and an auxiliar structure in the device memory.

In both stages, different techniques are used to optimize the performance: the use of shared memory, anticipatory copies and coalesced access to global memory.

5 Parallel Solution of Many Queries

In large-scale systems such as Web Search Engines indexing multimedia content, it is critical to efficiently deal 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 answer several queries in parallel. 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 software and hardware characteristics which allow us to think of solving many approximated queries in parallel. To this end, the system presented in Fig. 1 is modified and shown in Fig. 2. In this Figure it can be observed that the permutation index is built once and then used to answer many queries.

Therefore, the amount of resources required for this processing is equal to the amount of resources required to compute a query multiplied by the number of parallel queries resolved.

In order to answer many approximate queries in parallel, GPU receives the query set and has to solve all queries in the set. In parallel processing, each query applies the process explained in 4.2, therefore the amount of resources required for this is equal to the amount of resources to compute a query multiplied by the number of queries solved in parallel. The number of queries to be solved in parallel is determined according to the GPU resources, mainly its memory. If q are parallel queries, m is the required memory quantity per query and i is the required memory for the permutation index, then $q*m+i$ 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 careful management of blocks and their threads. At the same time, blocks of different queries are accessed in parallel. Hence it is important to have good administration of threads. i.e. to trace which query is solved and which database element is responsible. This task can be fulfilled by establishing a relationship among *Thread Id*, *Block Id*, *Query Id*, and *Database Element*.

6 Experimental Results

Our experiments considered different database sizes: 4KB, 29KB, 68KB, and 84KB, on a metric database consisting of English words using the *Levenshtein* distance, also called *edit* distance (the minimum number of character insertions, deletions, and substitutions needed to make two strings equal). The analysis was made for three GeForce GPU whose characteristics (Global Memory, SM, SP, Clock rate, Compute Capability) are GTX330: (512MB, 6, 48, 1.04GHz, 1.2), GTX520MX: (1024MB, 1, 48, 1.8GHz, 2.1) and GTX550Ti: (1024MB, 4, 192, 1.96GHz, 2.1). The CPU is an Intel core i3, 2.13 GHz and 3 GB of memory. The results are expressed in speedup ($Sp = \frac{Time_{Sec}}{Time_{Par}}$).

In this paper, we do not display the speedup of construction of *Permutation Index*. These results are illustrated in [16].

Fig. 3 and 4 show the obtained acceleration in range queries (3) and k-NN (4) queries for three database sizes and different number of permutants. In these results, 80 queries are solved in parallel. As it can be noticed *range queries* show improvements with respect to k-NN queries, but in both cases the achieved speedup is very good. In all cases, the influence of database size is clear, but good performance is achieved. The best results are for the largest database and the maximum number of permutants.

Tables 1 and 2 show the obtained throughput (number of queries by second) by our implementation. The results clearly show the benefits for all used architectures of GPU. In every case and query kind, the number of queries by second is high.

Table 1. Range search throughput

# Permut.	GTX520MX	GTX550Ti	GTX330
128	27639,72	29310,63	16973,21
100	28188,86	28907,35	16279,76
80	28921,82	29621,19	16828,75
64	29539,57	29362,77	16379,24
32	29144,71	29582,42	16774,19
16	29255,39	29248,81	16464,29
5	28197,27	29604,32	16474,46

Table 2. k-NN search throughput

# Permut.	GTX520MX	GTX550Ti	GTX330
128	19824,25	19377,68	10850,85
100	18816,38	19696,23	10956,71
80	19771,86	19203,07	11051,72
64	19797,83	18857,32	11137,65
32	19774,12	19289,62	10263,80
16	18785,79	19645,99	11237,29
5	19906,59	19121,16	11262,48

We can also observe that the number of permutants is not so important; for each GPU architecture and for all number of permutants, the throughput is similar, quasi constant.

Fig. 5 and 6 show the behavior of two operations: Range Query (Fig. 5) and k-NN Query (Fig.

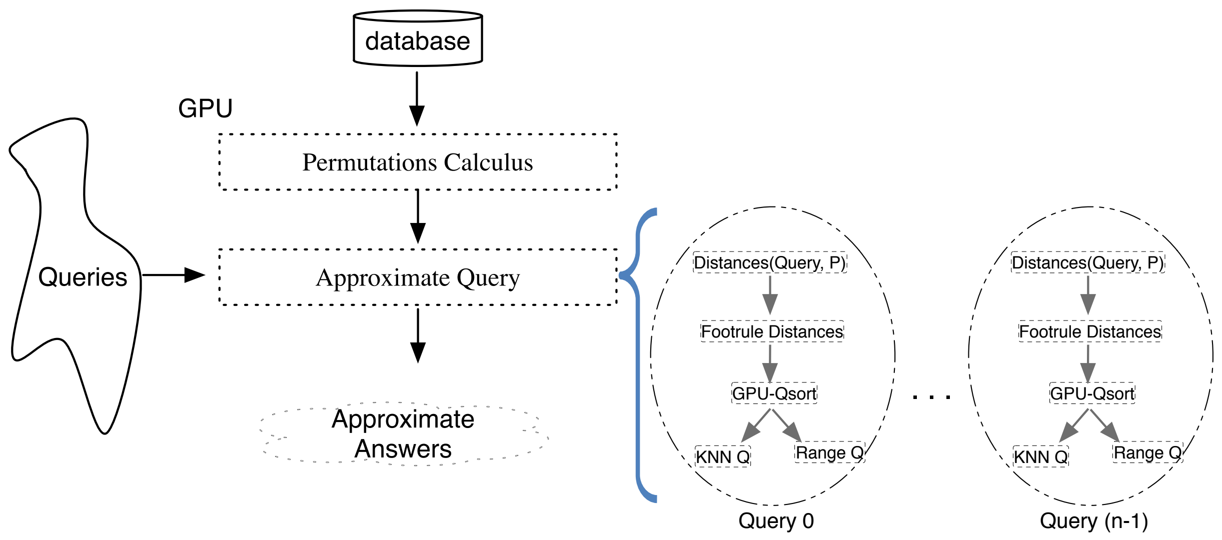


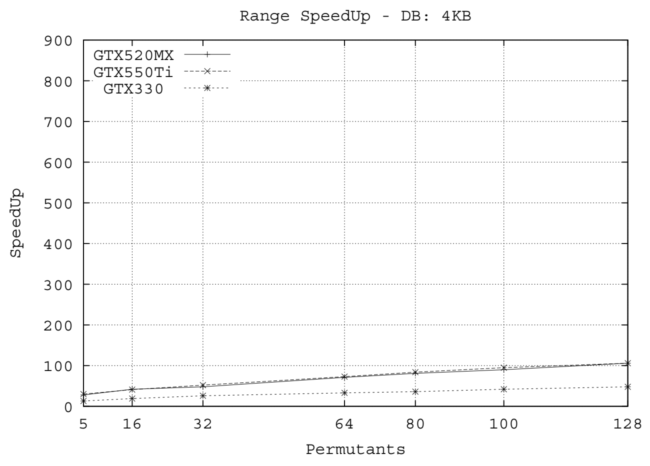
Fig. 2. Solving q queries in GPU-CUDA permutation index

6) for the biggest database and three numbers of permutants (5, 64, 128) when we vary the number of parallel queries in three GPUs. It can be seen that the best speedup was obtained when the number of queries is equal to 80 and the number of permutants is maximum. Also, the influence of GPU architecture is clear: when it has more resources, better speedup is achieved. Fig. 5(b) and 6(b) present these results.

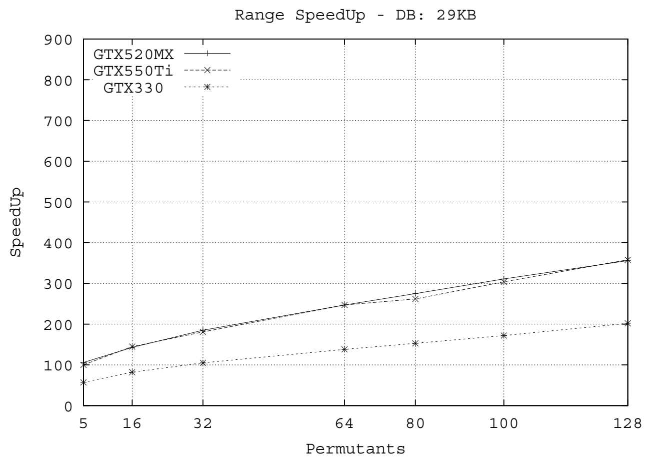
As it was mentioned previously, approximate similarity searching may obtain an inexact answer. That is, if a k-NN query of an element $q \in X$ is posed to the index, it answers with the k elements viewed as the k closest elements from U only between the elements that are actually compared with q. However, since 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 $r_k = \max_{1 \leq i \leq k}\{d(x_i, q)\}$ needed to enclose these k closest elements to q. An approximate answer of k-NN(q) could obtain some elements z whose $d(q, z) > r_k$. On 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 has not reviewed all the relevant elements. However, all the answered elements will be at a distance less or equal to r, so they belong to the exact answer to $(q, r)_d$. We evaluate the retrieval effectiveness through most commonly used measures: *recall* and *precision*. *Recall* is the ratio of relevant documents retrieved for a given query to the number of relevant documents for that query in the database. *Precision* is the ratio of the number of relevant documents retrieved to the total number of documents retrieved. Both recall and precision take on values between 0 and 1.

For k-NN searches, the experiments consider the values of k of 2, 4, and 16; and for the radii range the values are 1, 2, and 3. For the parameter f of the Permutation Index which 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 queries solved in parallel. In this paper, we do not display the speedup of construction of the *Permutation Index*. These results are illustrated in [16].

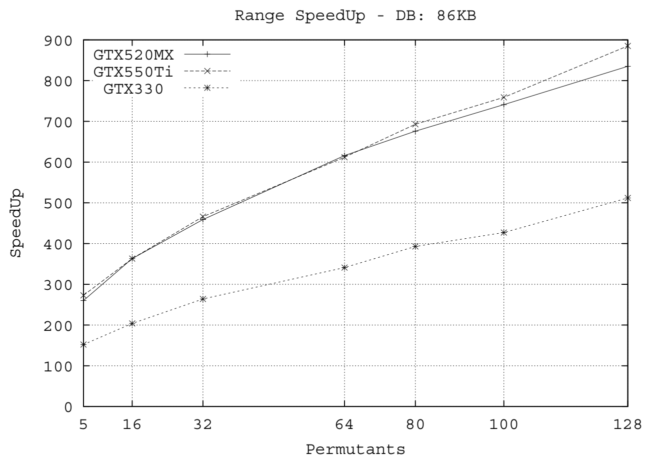
Our focus is to evaluate the tradeoff between the answer quality and time performance of our parallel index with respect to the sequential index. For each k-NN or range query we have previously obtained the exact answer, that is $Rel()$, and we obtain the approximate answer $Retr()$. Fig. 7 illustrates the average answer quality obtained for both kinds of queries, considering the Permutation Index with 64 (Fig. 7(a)) and 128 (Fig. 7(b)) permutants, respectively. It can be noticed that the Permutation



(a) Range Search 4KB.

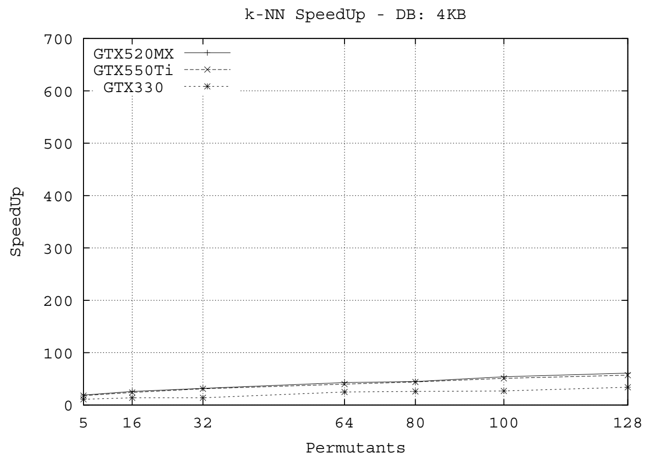


(b) Range Search 29KB.

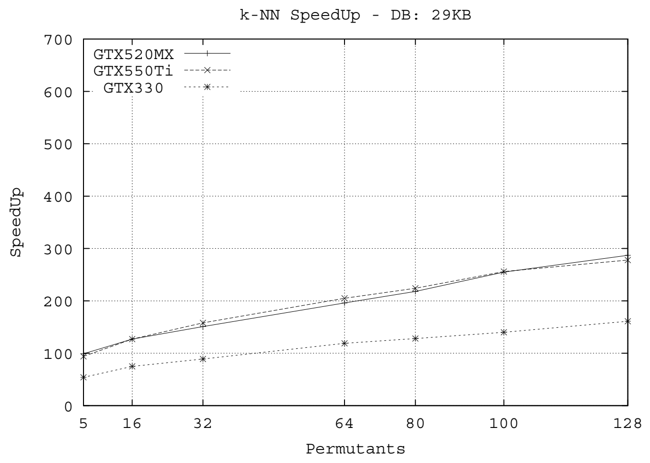


(c) Range Search 84KB.

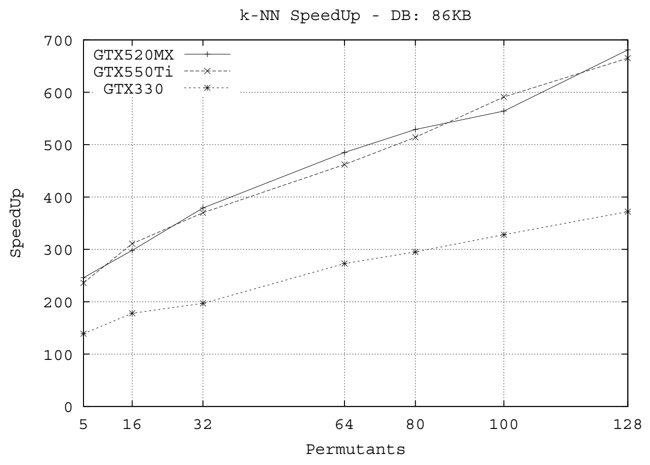
Fig. 3. Speedup of range search queries on three different GPUs



(a) k-NN Search 4KB.

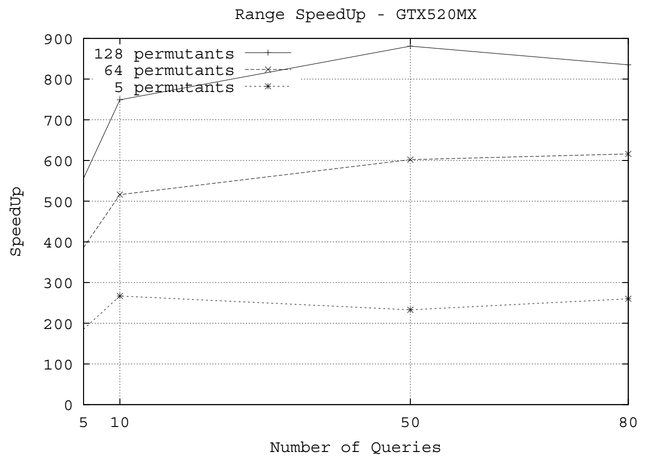


(b) k-NN Search 29KB.

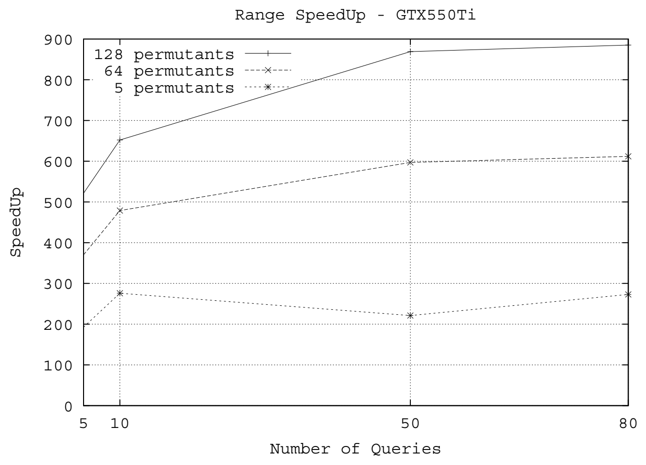


(c) k-NN Search 84KB.

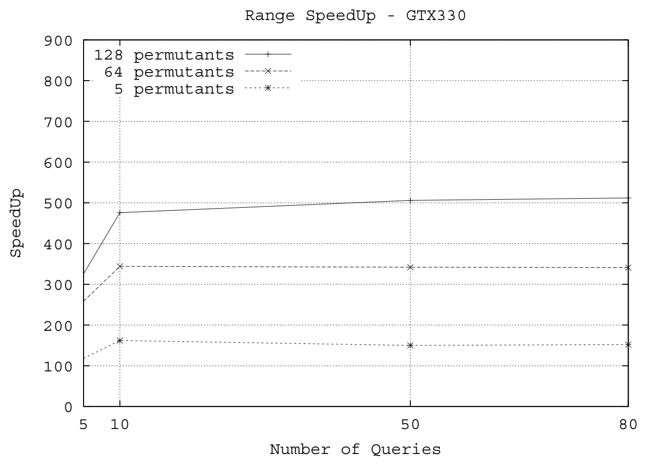
Fig. 4. Speedup of k-NN search queries on three different GPUs



(a) GTX520MX

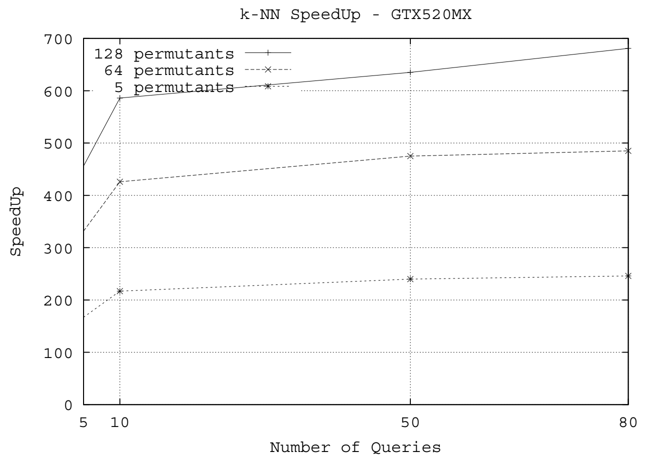


(b) GTX550Ti

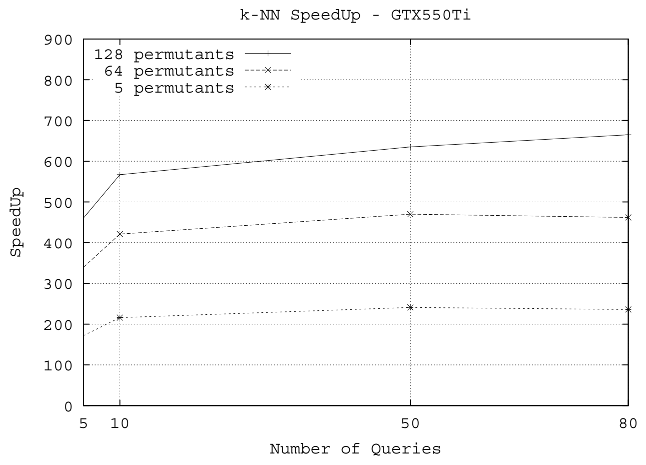


(c) GTX330

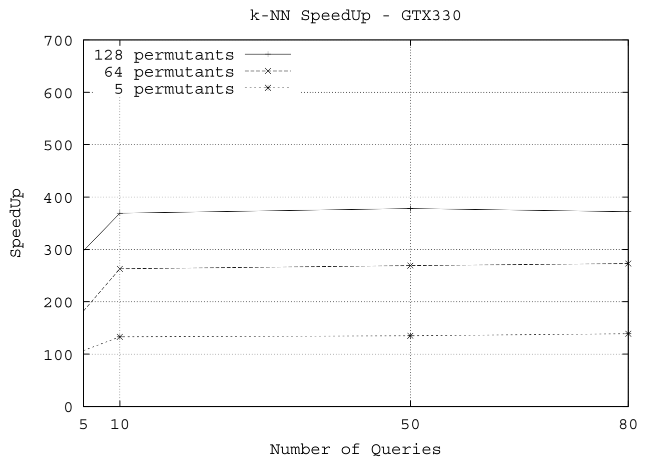
Fig. 5. Speedup of range search queries for different number of parallel queries



(a) GTX520MX



(b) GTX550Ti



(c) GTX330

Fig. 6. Speedup of k-NN search queries for different number of parallel queries

Index retrieves a high percentage of exact answer only by reviewing a little fraction of the database. For example, 10% retrieves 85% for 2-NN queries both with 64 and 128 permutants. It needs to review 20% to retrieve almost 80% of the 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 relevant objects.

6.1 GPU-Qsort with Other Solution

There are many quick sort libraries, one of them is *Thrust*. It is a part of CUDA repositories.

Thrust library provides a collection of fundamental parallel algorithms such as *scan*, *sort* and *reduction*. It solves a complementary set of problems, namely, (1) those that are implemented efficiently without a detailed mapping of work onto the target architecture or (2) those that do not merit or simply will not receive significant optimization effort by the user. With this library, developers describe their computation using a collection of high-level algorithms and completely delegate the decision of how to implement the computation to the library. This abstract interface allows programmers to describe what to compute without placing any additional restrictions on how to carry out the computation [12]. A disadvantage of *Thrust* is that it may isolate the developer from the hardware and expose only a subset of the hardware capabilities. In some circumstances, the C++ interface may become too awkward or verbose [9].

We compared our implementation with a solution based on *Thrust* library. We used the *Thrust* as a black box. Fig. 8 shows the comparison considering four database sizes. The results are the average over one hundred executions.

In the figure, we can observe that our implementation obtains better speedup than the solution using Thrust library. Besides, it is important to notice the independence of GPU-Qsort from GPU characteristics, it works fine in all GPU.

7 Conclusions and Future Work

As it is mentioned before, in large-scale systems such as Web Search Engines indexing multimedia content, it is critical to efficiently deal 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 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 having a high throughput due to its capacity of parallel processing of thousands of threads.

In this work we show an implementation that uses a *Pemutation Index* to solve approximate similarity search on a database of words. However, it is possible to extend our proposal easily to other metric databases of different data nature such as vectors, documents, DNA sequences, images, music, among others.

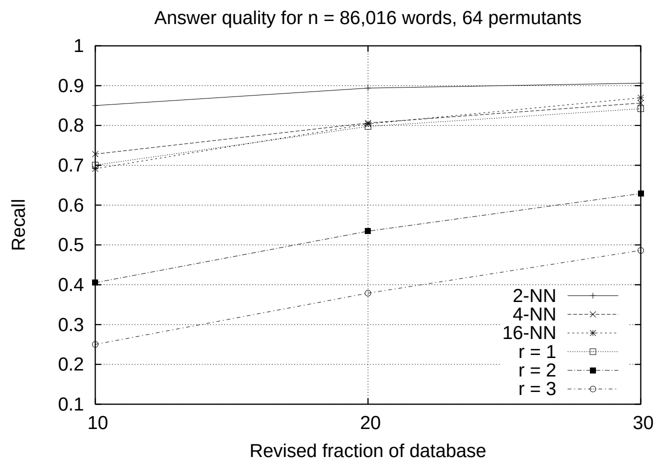
The empirical results have shown improvements in each architecture of GPU considered. Both speedup and throughput obtained are very good, showing better performance when the load work is hard.

Besides, the implemented GPU-*Pemutation Index* showed 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.

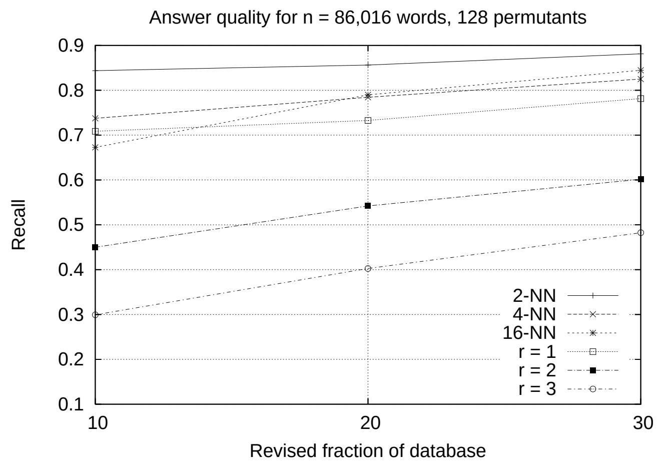
In future, we plan to make an exhaustive experimental evaluation considering other kinds of databases, and to compare our implementation with other solutions that apply GPU in the scenario of metric space approximate searches.

Acknowledgements

We wish to thank the UNSL for allowing us to access their computational resources. This research has been partially supported by Project UNSL-PROICO-30310 and Project UNSL-PROICO-330303.



(a) 64 permutants



(b) 128 permutants

Fig. 7. Recall of approximate-k-NN and range queries obtained with permutation index

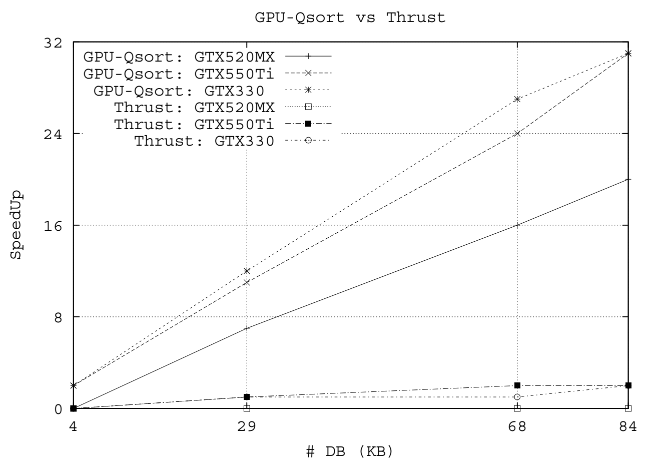


Fig. 8. Speedup of GPU-Qsort and thrust on three different GPUs

References

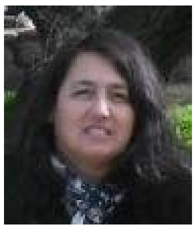
1. **Barrientos, R., Gomez, J., Tenllado, C., & Prieto, M.** (**2010**). Heap based k-nearest neighbor search on gpus. In *XXI Jornadas de Paralelismo*. 559–566.
2. **Barrientos, R. J., Gomez, J., Tenllado, C., Prieto, M., & Marin, M.** (**2011**). kNN Query Processing in Metric Spaces using GPUs. volume 6852. ISBN 978-3-642-23399-9, 380–392.
3. **Benjamin, B. & Navarro, G.** (**2004**). Probabilistic proximity searching algorithms based on compact partitions. *Discrete Algorithms*, 2(1), 115–134. ISSN 1570-8667. doi:10.1016/S1570-8667(03)00067-4.
4. **Bustos, B., Deussen, O., Hiller, S., & Keim, D.** (**2006**). A graphics hardware accelerated algorithm for nearest neighbor search. In *Proc. International Conference on Computational Science (ICCS'06) Part IV*, volume 3994 of *LNCS*. Springer, 196–199.
5. **Chavez, E., Figueroa, K., & Navarro, G.** (**2005**). Proximity searching in high dimensional spaces with a proximity preserving order. In *Proc. 4th Mexican International Conference on Artificial Intelligence (MICAI)*, LNAI 3789. 405–414.
6. **Chávez, E., Navarro, G., Baeza-Yates, R., & Marroquín, J.** (**2001**). Searching in metric spaces. *ACM Comput. Surv.*, 33(3), 273–321.
7. **Ciaccia, P. & Patella, M.** (**2010**). Approximate and probabilistic methods. *SIGSPATIAL Special*, 2(2), 16–19. ISSN 1946-7729. doi:10.1145/1862413.1862418.
8. **Fagin, R., Kumar, R., & Sivakumar, D.** (**2003**). Comparing top k lists. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '03. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA. ISBN 0-89871-538-5, 28–36.
9. **Farber, R.** (**2011**). *CUDA Application Design and Development*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition. ISBN 0123884268, 9780123884268.
10. **Figueroa, K., Chávez, E., Navarro, G., & Paredes, R.** (**2009**). Speeding up spatial approximation search in metric spaces. *ACM Journal of Experimental Algorithmics*, 14, article 3.6.
11. **Garcia, V., Debreuve, E., Nielsen, F., & Barlaud, M.** (**2010**). k-nearest neighbor search: fast GPU-based implementations and application to high-dimensional feature matching. In *IEEE International Conference on Image Processing*. Hong Kong, –.
12. **Hoberock, J. & Bell, N.** (**2010**). Thrust: A parallel template library. Version 1.3.0.
13. **Kato, K. & Hosino, T.** (**2010**). Solving k-nearest neighbor problem on multiple graphics processors. In **ACM**, editor, *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGRID*. 769–773.
14. **Kirk, D. B. & Hwu, W. W.** (**2010**). *Programming Massively Parallel Processors, A Hands on Approach*. Elsevier, Morgan Kaufmann. ISBN 978-0-12-381472-2.
15. **Liang, S., Liu, Y., Wang, C., & Jian, L.** (**2010**). Design and evaluation of a parallel k-nearest neighbor algorithm on CUDA-enabled GPU. In *IEEE 2nd Symposium on Web Society (SWS)*. ISBN 978-1-4244-6356-5, 53 – 60.
16. **Lopresti, M., Miranda, N., Piccoli, F., & Reyes, N.** (**2012**). Efficient similarity search on multimedia databases. In *XVIII Congreso Argentino de Ciencias de la Computación, CACIC 2012*. 1079–1088.
17. **Moreno-Seco, F., Micó, L., & Oncina, J.** (**2003**). A modification of the laesa algorithm for approximated k-nn classification. *Pattern Recognition Letters*, 24(1), 47 – 53. ISSN 0167-8655. doi:10.1016/S0167-8655(02)00187-3.
18. **NVIDIA** (**2012**). Nvidia cuda compute unified device architecture, programming guide version 4.2. In *NVIDIA*.
19. **Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., & Phillips, J.** (**2008**). GPU Computing. *IEEE*, 96(5), 879 – 899.
20. **Patella, M. & Ciaccia, P.** (**2009**). Approximate similarity search: A multi-faceted problem. *J. Discrete Algorithms*, 7(1), 36–48.
21. **Samet, H.** (**2005**). *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN 0123694469.
22. **Singh, A., Ferhatosmanoglu, H., & Tosun, A.** (**2003**). High dimensional reverse nearest neighbor queries. In *The twelfth international conference on Information and knowledge management*, CIKM '03. ACM, New York, NY, USA. ISBN 1-58113-723-0, 91–98. doi:10.1145/956863.956882.
23. **Singleton, R.** (**1969**). Algorithm 347: an efficient algorithm for sorting with minimal storage [m1]. *Commun. ACM*, 12(3), 185–186. ISSN 0001-0782.
24. **Uribe-Paredes, R., Valero-Lara, P., Arias, E., Sánchez, J. L., & Cazorla, D.** (**2011**). A GPU-Based Implementation for Range Queries on Spaghettis Data Structure. In *ICCSA (1)*, volume 6782 of *Lecture Notes in Computer Science*. Springer. ISBN 978-3-642-21927-6, 615–629.

25. **Zezula, P., Amato, G., Dohnal, V., & Batko, M.** (**2006**). *Similarity Search: The Metric Space Approach*. Advances in Database Systems, vol.32. Springer.

Mariela Lopresti received her B.Sc. degree from the Universidad Nacional de San Luis, Argentina, in 2005. She pursues her M.Sc. in Computer Science at the Universidad Nacional de San Luis, where she is also an assistant professor. Her research interests include similarity searching, non-conventional database, parallel algorithms, HPC, and GPGPU.

Natalia Miranda received her B.Sc. degree from the Universidad Nacional de San Luis, Argentina, in 2005. Currently, she is a Ph.D. student in Computer Science at the Universidad Nacional de San Luis, where she is also an assistant professor and has a CONICET scholarship. Her research interests include multimedia data, similarity searching, non-conventional database, parallel algorithms, HPC, and GPGPU.

Fabiana Piccoli received her B.Sc. degree from the Universidad Nacional de San Luis, Argentina, in 1995, the M.Sc. degree in Computer Science from the Universidad Nacional de Sur, Argentina, in 2001, and the Ph.D. degree in Computer Science from the Universidad Nacional de San Luis, Argentina, in 2005, where she is currently a professor. Her research interests include parallel systems, parallel models and paradigm, parallel algorithms, HPC, and GPGPU. She has more than 70 technical contributions in national and international conference proceedings, books and journals.



Nora Reyes received her B.Sc. degree (with honor) and M.Sc. degree in Computer Science from the Universidad Nacional de San Luis, Argentina, in 1999 and 2002, respectively, where she is currently a professor and has been the head of the Departamento de Informática for 6 years. Her research interests include algorithms, similarity searching, non-conventional database, information retrieval, and data structures in general. She has more than 40 technical contributions in national and international conference proceedings and journals.

Article received on 19/02/2013; accepted on 25/07/2013.