Efficient Similarity Search on Multimedia Databases

Mariela Lopresti, Natalia Miranda, Fabiana Piccoli, Nora Reyes

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

Abstract. Manipulating and retrieving multimedia data has received increasing attention with the advent of cloud storage facilities. The ability of querying by similarity over large data collections is mandatory to improve storage and user interfaces. But, all of them are expensive operations to solve only in CPU; thus, it is convenient to take into account High Performance Computing (HPC) techniques in their solutions. The Graphics Processing Unit (GPU) as an alternative HPC device has been increasingly used to speedup certain computing processes. This work introduces a pure GPU architecture to build the *Permutation Index* and to solve approximate similarity queries on multimedia databases. The empirical results of each implementation have achieved different level of speedup which are related with characteristics of GPU and the particular database used.

Keywords: Multimedia database, Metric Space, Approximate Similarity Query, High Performance Computing, Graphics Processing Unit.

1   Introduction

Due to an increasing interest in manipulating and retrieving multimedia data, nowadays the problem of similarity searching receives much attention. The metric space model is a paradigm that allows to modelize all the similarity search problems. Metric databases permit storing objects from a metric space and performing similarity queries over them efficiently; that is, in general, by reducing the number of distance evaluations needed. 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 which make it a metric. In metric database there are two main queries of interest[1–3]: Range Searching and the k Nearest Neighbors. Given a dataset of $\mid U \mid = n$ objects, queries can be trivially answered by performing n distance evaluations, but sequential scan does not scale for large problems. Therefore, the goal is to preprocess the dataset such that queries can be answered with as few distance computations as possible. 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.

There are different indices that store different information about distances [3]. An index helps to retrieve the objects from U that are relevant to the query by making much less than n distance evaluation during searches. One of these indices is the *Permutation Index* [4]. In order to employ high performance computing 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. The use of GPUs in general-purpose computing is becoming a very accepted alternative.

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 manycores coprocessor ables to support fine grain parallelism (a lot of threads run in parallel, all of them collaborate in the solution of the same problem) [5,6]. GPU is different than other parallel architectures because it shows flexibility in the local resources allocation to the threads. In general, a GPU multiprocessor consists of several streams multiprocessors, each of them has 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.

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: the parallel work through concurrent threads and the memory hierarchy. A CUDA program consists of multiple phases executed on either CPU or GPU.

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

2 Previous Concepts and Related Works

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.

There are two main queries of interest[1–3]: Range Searching and the k Nearest Neighbors (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 $| k - NN(q) |= k$ and $\forall x \in k - NN(q), v \in U \wedge v \notin k - NN(q), d(q, x) \leq d(q, v)$.

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 evaluation 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 [7].

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 [3]: *pivot based algorithms* and *compact partition based algorithms*. In the first case, the index consists in 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 $\{d(p_1, x), d(p_2, x), \ldots, d(p_m, x)\}, x \in U$. The queries are solved considering all pivots. In the second case, the space is divided into small and compact zones. A set of objects, called *centers*, $\{c_1, \ldots, c_s\} \subseteq U$ are chosen and the rest of the elements are distributed into the 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.

Other classification for proximity searching are the *approximate* and *probabilistic* algorithms. 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 [8]. Probabilistic algorithms on the other hand state that the answer is correct with high probability. Some examples are [9, 10]. In the next section we detail a probabilistic method: *Permutation Index*[4].

GPGPU GPU is a dedicated graphic card for personal computers, workstations or video game consoles. It is an interesting architecture to high performance computing. GPU was developed with a highly parallel structure, high memory bandwidth and more chip surface dedicated to data processing than to data caching and flow control. It offers speedup to any standard graphics application[5].

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[5, 6]. 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 [5, 12]. 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.

GPU computation considers a hierarchy of abstraction layers: *grid*, *blocks* and *threads*. The *threads*, basic execution unit that executes *kernel* funtion, in the CUDA model are

grouped into *blocks*. All threads in a block execute on one SM and communicate among them through the *shared memory*. Threads in different blocks can communicate through *global memory*. Besides shared and global memory, the 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.

Respect of 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.

Metric Spaces and GPU In metric spaces, the 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 to GPU. Querying by k-NN has concentrated the greatest attention of researchers in the area, so there are many solutions that consider GPU. In [13–17] differents proposal are made, all of them improvement to brute force algorithm (sequential scan) to find the k-NN of a query object. They differ in which process part is parallelized or which methodology is applied.

Besides, different parallel implementations of scan sequential, there are others proposals, [13, 18, 19], that implement solutions for metric indices: List of Clusters, SSS-Index and Spaguettis index.

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

3 Permutation Index

Let be $\mathcal{P}$ 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 [4].

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 [11]. All of them are metrics, because they satisfy the afore-

```
RangeQuery(element q, radius r, fraction f)
1.    Let A[1,n] be an array of tuples and U = {x_1,...,x_n}
2.    Compute Π_q^{-1}
3.    For i ← 1 to n do
4.        A[i] ← ⟨x_i, S_ρ(Π_{x_i}, Π_q)⟩
5.    SortIncreasing(A) /* by second component of tuples */
6.    For i ← 1 to fn do
7.        ⟨x,s⟩ ← A[i]
8.        If d(q,x) ≤ r Then Report x
```

Algorithm 1: Range query of q with radius r in a permutation index, f database fraction.

mentioned properties. We use the Spearman Rho metric because it is not expensive to compute and according to the authors in [4] it has a good performance to predict proximity between elements. The square of the Spearman Rho S_ρ metric is defined as $S_\rho(x,q) = S_\rho(\Pi_x, \Pi_q) = \sum_{i=1}^{m} |\Pi_x^{-1}(p_i) - \Pi_q^{-1}(p_i)|^2$.

This distance $S_\rho(\Pi_q, \Pi_x)$ can be computed in $O(m)$ time [11]. Therefore, during pre-processing 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 $O(nm \log m)$ 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, and next we sort the elements $x \in U$ into increasing order according to $S_\rho(\Pi_x, \Pi_q)$ (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.

Algorithm 1 shows the process for a range query. The efficiency and the quality of the answer obviously depend on f: as f grows the efficiency degrades, but the answer quality improves. A way to obtain good values for f is discussed in [4].

4   GPU-CUDA 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. The next sections detail the characteristics of each process, their steps and peculiarities.

4.1   Building the Permutation Index

Building a permutation index in GPU involves at least two steps. The first step calculates the distance among every object in database and the permutants. The second one sets up the signatures of all objects in database, i.e. all object permutations. The process input is a database where some of its elements are 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 permutation object according to a global permutants set.

In the first task ($Distances(O, P)$), 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 step end, each threads block save
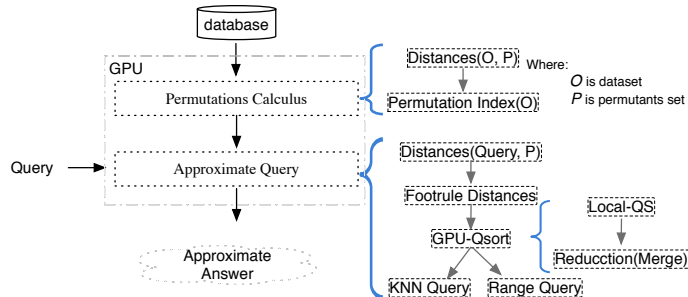
Fig. 1. Indexing and Querying in GPU-CUDA permutation index.

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 of fixed size defined in the shared memory of block (It stores the permutants, if the permutants 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 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 $N \times M$.

4.2 Solving Approximate Queries

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. As sorting methodology, we implement the Quick-sort in the GPU, its characteristics are explained bellow. 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 that each thread computes the *Euclidean distance*, all distances are sorted (using GPU-Qsort) and the results are the first k elements of sorted list.

Considering the sorted algorithm, we describe an 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 1.2 or higher.

GPU-Qsort carries out the task into two stages: $Local-Qsort$ and $Merge-Reduction$. The first stage, **Local-Qsort**, has a data sequence as input and its output are N sorted subsequences. Each subsequence is ordered by a threads block according to iterative quicksort. Therefore, there are N threads blocks, where the number of threads by block

is fix and is determined in relation to the required resources by block. Each block chooses a local pivot (it has to belong to input data list of block) and divides the data sequence in two subsequences: one has the elements smaller than pivot and another has the elements greater or equal than pivot. The pivot is the median among three elements of data subsequence: the first, middle and last element [20]. Each block works independently of other blocks eliminating the need of synchronization among threads of different blocks. In base to the selected pivot, all elements lower than the pivot are moved to a position to the pivot's left, and the greater or equal are shifted to the pivot's right. The task is made by using shared memory and each thread can determine itself the position for its element in shared structure (using CUDA atomic functions).

The process is applied iteratively over two subsequences. It is possible if it uses an stack. The stack saves all subsequences that still remain to be sorted. When there are two ready subsequences to work, one is selected and the another is pushed in the stack. If one subsequence is sorted, the subsequence in the top of stack is selected to work. The iterative process ends when the stack is empty and the list is sorted. When the number of elements in the sequence is lower than eight, it is sorted in sequential manner, because the process overhead is too large compared to sequence size.

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

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

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

5   Analysis of Experimental Results

In this section we show and analyse the results for the solution of index building and the different kinds of queries. Each reported value is the average of many executions of corresponding algorithm that is detailed above. Our experiments considered different database sizes: 4KB, 29KB, 68KB, and 84KB, on a metric database consisting of English words and using the edit distance (that is, the minimum number of character insertions, deletions, and substitutions needed to make two strings equal). For lack of space we show only the results of sizes 29KB and 84KB, the other sizes yielded similar results. The analysis was made for three generations of GeForce GPU whose characteristics (Global Memory, SM, SP, Clock rate, Compute Capability) are GTX330: (512MB,6,48,1.04GHz,1.2), GTX470: (1280MB, 14, 448, 1.2GHz, 2.0) and GTX550Ti: (1024MB, 4, 192, 1.96GHz,2.1). The CPU is and intel core i3, 2.13 GHz and 3 GB of memory. The results are expressed in Speed up ($Sp = \frac{Time_{Sec}}{Time_{Par}}$).

In Figure 2 we show the accelerations obtained by our GPU parallel implementation of *Permutation Index*. Although in all devices we achieve to improve the performance, the better results are obtained in those devices with more resources and modern architecture. In oldest device, the behavior is similar for all size database. In Fermi devices: *GTX 470* and *GTX 550 Ti*, we can observe good profits when the database size is bigger. Moreover, when the permutants number is greater, better performance is reached.
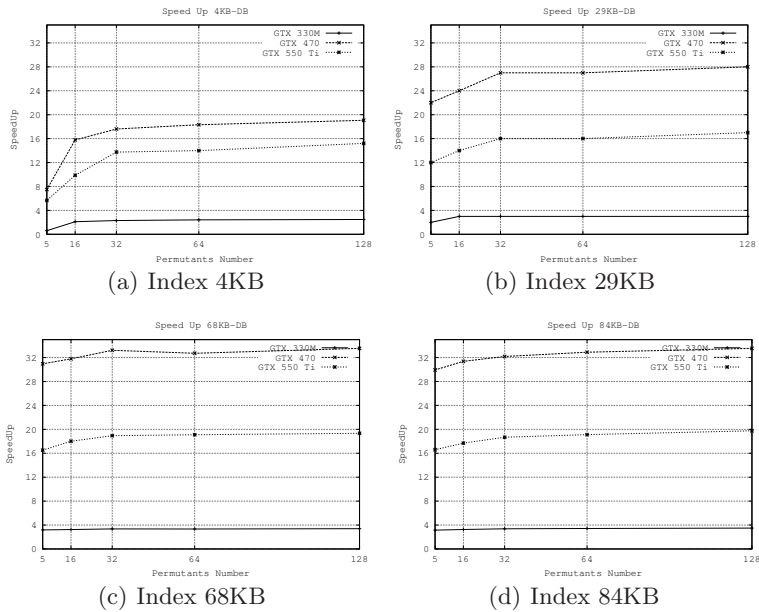
(a) Index 4KB

(b) Index 29KB

(c) Index 68KB

(d) Index 84KB

Fig. 2. Speedup of Permutation Index on three different GPUs.

Figure 3 shows the obtained acceleration in range (3(a) and 3(b)) and k-NN (3(c) and 3(d)) queries. As it can be noticed *Range queries* shows improvements in its performance for all databases, but as permutants number increases, minor speedup is obtained. In k-NN queries better results are achieved with 16 and 32 permutants, for large k, and for the smallest database.

The Figure 4 resumes the behavior of three operations: Index Creation(4(a)), Range Query(4(b)) and k-NN Query (4(c)), for 64 permutants. Even though the best performance is achieved by Index Creation Process, the profits of the query processes are encouraging, because when they solve only one query (they do not use all GPU resources), they speed up the solutions, in consequence, everything indicates that better performance will get when multiple queries will be solve in parallel, even in those cases without speed up.

6 Conclusions, Remarks and Future Work

The GPU is a massively parallel architecture, it has a high throughput because its capacity of parallel processing for thousands of threads. With each new generation of GPU, new parallel processing capabilities are incorporated. Therefore, it is adequate to accelerate metric space solutions, because they are expensive to solve only in CPU.

In this work we show an implementation of one index, the *Pemutation Index*, used for approximate similarity search on a database of words. However, it is possible to easily extend our proposal to other metric databases of different data nature, such as vectors, documents, DNA sequences, images, music, among others. The empirical evaluation has
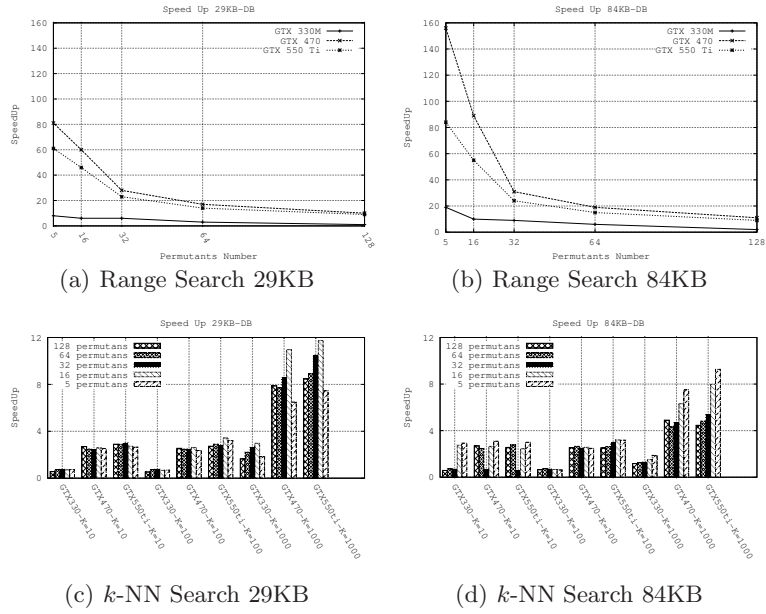
(a) Range Search 29KB (b) Range Search 84KB

(c) k-NN Search 29KB (d) k-NN Search 84KB

Fig. 3. Speedup of Queries on three different GPUs.

(a) Index 64 Permutants (b) Range 64 Permutants
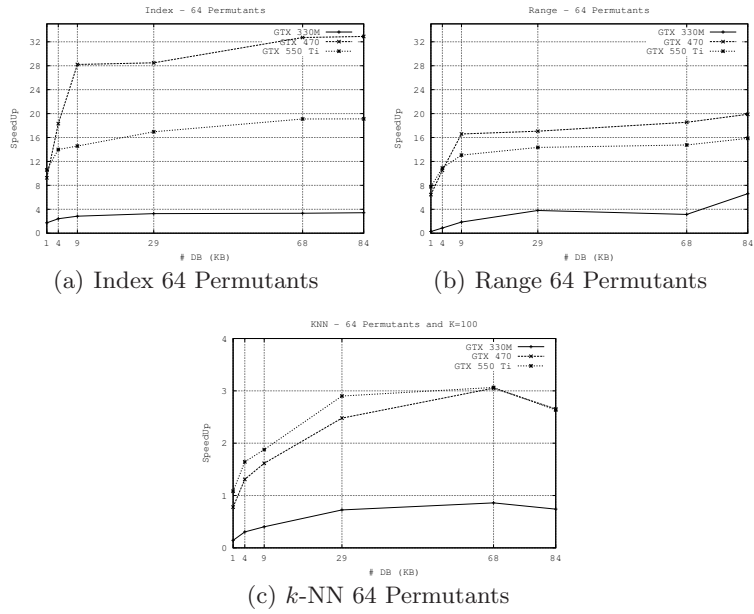
(c) k-NN 64 Permutants

Fig. 4. Speedup of three operations for 64 permutants on three different GPUs.

demonstrated improvements in the different architectures considered. Still we can optimize some issues, for example solving many queries at the same time by taking advantage of characteristics of GPU and its programming model.

In the future, we plan to make an exhaustive experimental evaluation considering others kinds of database, comparing with other solutions that apply GPU in the scenario of metric space approximate searches. We need also to evaluate retrieval effectiveness of the answer of the *Permutation Index*, as the number of objects directly compared with the query grows, by using *Recall* and *Precission* measures.

References

1. P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search: The Metric Space Approach.* Advances in Database Systems, vol.32. Springer, 2006.
2. H. Samet. *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, 2005.
3. E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín. Searching in metric spaces. *ACM Comput. Surv.*, 33(3):273–321, 2001.
4. 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)*, LNAI 3789, pages 405–414, 2005.
5. D. B. Kirk and W. W. Hwu. *Programming Massively Parallel Processors, A Hands on Approach.* Elsevier, Morgan Kaufmann, 2010.
6. J.D. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU Computing. In *IEEE*, volume 96, pages 879 – 899, 2008.
7. K. Figueroa, E. Chávez, G. Navarro, and R. Paredes. Speeding up spatial approximation search in metric spaces. *ACM Journal of Experimental Algorithmics*, 14:article 3.6, 2009.
8. B. Benjamin and G. Navarro. Probabilistic proximity searching algorithms based on compact partitions. *Discrete Algorithms*, 2(1):115–134, March 2004.
9. A. Singh, H. Ferhatosmanoglu, and A. Tosun. High dimensional reverse nearest neighbor queries. In *The twelfth international conference on Information and knowledge management*, CIKM '03, pages 91–98, New York, NY, USA, 2003. ACM.
10. F. Moreno-Seco, L. Mic, and J. Oncina. A modification of the laesa algorithm for approximated k-nn classification. *Pattern Recognition Letters*, 24(13):47 – 53, 2003.
11. R. Fagin, R. Kumar, and D. Sivakumar. Comparing top k lists. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '03, pages 28–36, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.
12. NVIDIA. Nvidia cuda compute unified device architecture, programming guide version 4.2. In *NVIDIA*, 2012.
13. R.J. Barrientos, J.I. Gomez, C. Tenllado, and M. Prieto. Heap based k-nearest neighbor search on gpus. pages 559–566, 09/2010 2010.
14. B. Bustos, O. Deussen, S. Hiller, and D. Keim. A graphics hardware accelerated algorithm for nearest neighbor search. In *Proc. International Conference on Computational Science (ICCS'06) Part IV*, volume 3994 of *LNCS*, pages 196–199. Springer, 2006.
15. 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 International Conference on Image Processing*, Hong Kong, Sept. 2010.
16. K. Kato and T. Hosino. 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*, pages 769–773, 2010.
17. 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)*, pages 53 – 60, 2010.
18. R. J. Barrientos, J.I. Gomez, C. Tenllado, M. Prieto, and M. Marin. kNN Query Processing in Metric Spaces using GPUs. volume 6852, pages 380–392, 2011.
19. R. Uribe-Paredes, P. Valero-Lara, E. Arias, J. L. Sánchez, and D. Cazorla. A GPU-Based Implementation for Range Queries on Spaghettis Data Structure. In *ICCSA (1)*, volume 6782 of *Lecture Notes in Computer Science*, pages 615–629. Springer, 2011.
20. R.C. Singleton. Algorithm 347: an efficient algorithm for sorting with minimal storage [m1]. *Commun. ACM*, 12(3):185–186, March 1969.