(Very) Fast (All) k-Nearest Neighbors in Metric and Non Metric Spaces without Indexing

Natalia Miranda[1], Edgar Chávez[2], María Fabiana Piccoli[1], and Nora Reyes[1]

[1] Universidad Nacional de San Luis, Argentina
[2] Universidad Nacional Autónoma de México

Abstract. Proximity queries consists in retrieving objects near a given query. To avoid a brute force scan over a large database, an index can be used. However, for some problems, indexes are mostly useless (their running times are worst than sequential scan).

On the other hand, researchers have tried massively parallel hardware (as GPGPU) in the quest of faster query times. The results have been modest because current algorithms are cumbersome, while GPGPU architectures favor simple kernels, have a clear memory hierarchy and need close to zero cross-talk between processing units. We have engineered very fast algorithms for proximity queries taking into account this principles, all of them are presented in this paper.

In our approach no index is built, the cross-talk between threads is eliminated, and the higher (faster) levels of memory hierarchy are consistently used. The absence of data structures allows to use all the available memory for the database, and furthermore makes possible to do stream processing on very large data collections.

1 Introduction

Due to an increasing interest in manipulating and retrieving complex data, the problem of proximity searching has received a lot of attention, while simultaneously give hard times to practitioners. Obtaining the k-Nearest Neighbors (k-NN) of a query object is central to many complex data operations such as query by content, copy detection, object tracking, and a large set of other applications. The problem is pervasive and it is found in many areas of research, from statistics, pattern recognition, computer vision to multimedia databases, and much more applications. For this and other proximity searching problems a sequential scan over the database can solve the problem, but for large instances it does not scale. This is a very active topic of research well documented in the literature[1–3]. The goal of indexing is to avoid a sequential scan to answer proximity queries. Indexing is a two stage problem, firstly the database is preprocessed and then it is ready to be queried. Indexing can be very time consuming, some indexes use quadratic time and/or quadratic space. Moreover, there are situations where the need to query is immediate. In those cases building a index is not even an option. An example is *object tracking*. As soon as an object is marked for tracking it needs to be reported in real time in every frame; there is no time to index.

N. Brisaboa, O. Pedreira, and P. Zezula (Eds.): SISAP 2013, LNCS 8199, pp. 300–311, 2013.

Another example is duplicate detection in video and/or audio streams. Indexing is very slow and furthermore, in some applications the database will be queried only a few times making the amortized cost higher than a few sequential scans.

The above scenario is very restrictive and the usual (sequential and massively parallel) techniques documented in the literature are of no use for the problem. In this paper we present algorithms to query on the fly databases for proximity, using massive parallel hardware, more specifically, we are interested in the General Purpose Graphics Processing Unit (GPGPU). Current solutions for the GPU architecture are translated from the sequential counterparts, the net result is a very modest speedup due to auxiliary data structures, and/or the need to precompute an index. In this paper, by using a very strict adherence to the design principles and recommendations for the GPU architecture we report a solution orders of magnitude faster than sequential indexes and faster than other massively parallel approaches reported in the literature. Since we use no index, the data is readily available for querying on load, leveraging streaming applications where massive data is seeing only once and there is no time to preprocess or index.

1.1 Metric Space and Proximity Searching

A database for proximity searching is usually modeled as a finite subset of a metric space. A metric load is (X, S, d), with X a set (possibly infinite), a finite subset $S \subseteq X$ the database, and a distance $d : X \times X \to R^+$. Distances are preferred to plain (dis)similarity functions because they can be indexed, in principle, exploiting the triangle inequality. Non metric spaces have been also indexed making a reduction to metric spaces. We will not make use of triangle inequality, hence our model is more general. Queries of interest include Range Searching and the k Nearest Neighbors (k-NN). A range search is defined as $(q, r)_d = \{x \in U / d(q, x) \leq r\})$. In a k-NN query the goal is to retrieve the k closest elements in S to a query q, namely $|k\text{-NN}(q)| = k$ and $\{\forall x \in k\text{-NN}(q), v \in S \wedge v \notin k\text{-NN}(q), d(q, x) \leq d(q, v)\}$. Other problem of interest is the *All-k*-NN, solving the above problem for all x in S.

The goal of indexing is to avoid a sequential scan, but there is a well documented phenomenon known as *the curse of dimensionality* [1–3]. The measurable effect is that using an index can be slower than a sequential scan over the data. To avoid this odd behavior, approximate algorithms have been proposed, they usually have 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, or they have probabilistic guarantees[4].

1.2 GPGPU

The GPU is a dedicated graphic card for personal computers, workstations or video game consoles. It is an interesting architecture for high performance computing. The 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, in principle, a speedup to any standard graphics application[5]. Mapping general-purpose computation onto GPU implies the use of 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*, the *CPU-GPU memory structure* and the *Number of parallel threads*.

Every GPGPU program have some basic steps. Firstly the input data should be transferred to the graphics card from the CPU(host). Once the data is in place, a massive amount of 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 every class of problem can be solved with the GPU architecture, the most suitable problems are those implementable with stream processing and using limited memory size, i.e. applications with abundant data parallelism without cross-talking among processes. The programming model is Single Instruction Multiple Data (SIMD).

The CUDA, supported since the NVIDIA Geforce 8 Series, enables to use GPU as a highly parallel computer for non-graphics applications [5, 7]. 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 the CPU. It has multiple streaming multiprocessors (SMs), each of them contains several (eight, thirty-two or forty-eight) 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. In opposition, if the phases present much data parallelism, they are implemented as *kernel* functions in the GPU. A *kernel* function defines the code to be executed by all the threads launched in a parallel phase. The GPU resources are much more efficiently used if the kernel do not make branching (represented as `if` instructions), in other words, if all the threads follow the same execution path.

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 are executed on one SM and can 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 own local data space for variables. All $Thread - blocks$ form a *grid*. The number of grids, blocks per grid and threads per block are parameters fixed by the programmer, they can be adjusted to improve the performance.

With respect to the 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 than the kernel. All threads have access to the global memory and two additional read-only memory spaces: the constant and texture memory spaces. The constant and texture memory spaces are optimized for different memory usages. The global, constant and texture memory spaces are persistent all the application life time. Each kind of memory has its own access cost, in order of speed it will be local, shared and global memory which is the most expensive to access. Please notice that local and shared memory have higher throughput and smaller latency than standard RAM in the CPU. Please bear that in mind, because this contributes to the very large speedup of our algorithms.

1.3 Related Work on GPU Proximity Searching

There are many massively parallel algorithms for metric indexes implemented in a GPU. Querying for k-NN has obtained most of the attention of researchers in the area. In [8–14] improve explicity the brute force algorithm (or sequential scan) to find the k-NN. They differ in the parts parallelized or the methodology applied. Other works [8, 15, 16] implemented some well known sequential metric indices, such as the List of Clusters (LC) and the SSS-Index. For the case of vector data authors in [17, 17–19] use Kd-trees for finding the k-NN and [18] apply a variant of the Kd-tree for the all k-NN problem.

All algorithms in the literature [8–13, 15, 16, 19] for k-NN using GPU, solutions have high complexity in the data structures. Furthermore, they have a high granularity. Kernels are not uniform and have a lot of branching. This implies synchronization and serialization of the threads, which means all of them have to wait to be in the same path again to resume. In a nutshell, they use conditionals and do diverse tasks depending on comparisons. On the other hand the algorithms demand a lot of memory resources for the data structures and intermediate data, e.g. distances to pivots, and allocate only very small instances of the metric databases. For example in [16] they use only one thread block for the actual k-NN search, this implies overloading all the threads in the block and consequently suboptimal GPU resources usage, most of the threads are not used. In [10–12] they propose to solve several queries at a time, but they use just the same amount of threads than for a single query. This again implies thread overload, memory starvation and idle processing units in the GPU. In [13] is also suboptimal in resource usage, to the point of letting a single thread to finish the searching process, implying all other threads are idle.

We have learned from all the above examples and in our proposal, detailed below, we have closely tailored a solution which is uniform, with a single branch which maximize the GPU usage. We have carefully selected the number of threads, have coalescent memory access, using sharing memory and with data independence which implies zero cross talk between threads. Additionally we have zero overload in the data structures, which implies all the available memory can

be used for the database. Other researchers in High Performance Computing arrived to similar conclusions parallelizing matrix to matrix multiplication[20]. Shared memory is faster than GPU memory, and by avoiding branching they have obtained super linear speedup for their problem.

2 Our Approach

In the GPU we have enough juice to compute the distances from a query to all the database elements in one step. Even more, there will be much more threads available than the number of database elements allocated in the global GPU memory. After all the distances are computed, the objects with the smallest distances have to be identified, this is the real challenge. Those objects have to be selected avoiding crosstalk. In this context crosstalk is simply comparing the result of two threads. We will exemplify the problem with an unplugged setup. Please also notice that for a range query, all threads having distances smaller than the query radius can report themselves without the need to compare with anyone else, the problem is relevant only for k-NN queries.

The above problem is equivalent to the following: In a room with one thousand (say N) people everybody gets a number in a piece of paper. How can you know if you are among the k people with the smallest number in the piece of paper? A sorting algorithm is overkill, and looking at every other number is also very slow. Please notice that we have dropped the condition of a central process selecting the k smallest, the person have to claim the place without talking to others. We have devised a mechanism to obtain those k smallest numbers without exchanging information between everyone in the room. The kernel of a GPU/CPU CUDA program works better if the code is simple. Our contribution relies precisely in this step. We essentially need an algorithm to find the Top-k smallest elements in an array without a central supervisor.

Following the people in the room example, what we do is to arrange people in small groups and all of them show their paper among each other. If all of the numbers are bigger than mine I have the smaller one. I know I will be among the finalists and silently move to the finalist stage without talking to anyone else. The process can be repeated until only the smaller one remains. Extending the algorithm to k elements is straightforward.

We have implemented three different solutions for obtaining the $Top-k$ smallest elements in the array. Please notice that it is not necessary to sort all the objects in the stream, since we need only the k smallest. Figure 1 illustrates this stage.

The process iterates while the size of k-NN$_P$ list is grater than k. The *Partition* and *Join-K* stages are implicitly computed. Each launched thread determines the sub-list of work (*Partition*). Each thread determines over which element it has to work based on the local information brought from *IdThread* in the block and *IdBlock* that thread belongs. The *Select_k* stage is a computation unit that is responsible to choose k-NN objects of list L_i ($\cup L_i = L \forall i = 0 \cdots x - 1$ and $x = \#B$) and store in partial list k-NN$_P$ ($Join - K$). The first time L is equal

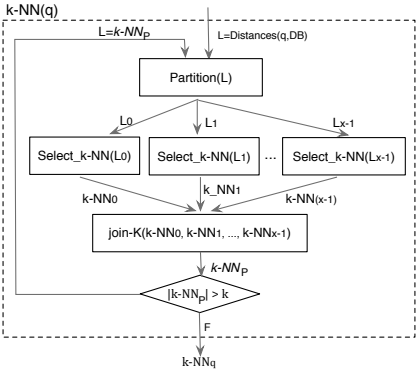


Fig. 1. Generic GPU Process to obtain the k nearest neighbor of a query q

to distances list. The next iteration, L has only all k-NN objects of each block. In the next sections, we explain different implementations of the *Select_k* stage.

2.1 *TopK_AA*

This is the implementation of the algorithm we designed to solve the problem, we named it *TopK_AA* (*AA=All-to-All*). The *Select_k* stage selects k-NN objects from the local list. Each thread in its block determines if its *distance* is among k-NN of its block. The comparison is *one-to-all*, each thread in the block contrasts its *distance* against all *distances* in the block. There are as many threads as distances in the local list. Once each thread establishes which is its order, it analyses if its distance is one of k-NN. If true, it reports its object to k-NN$_P$, otherwise it finishes. In the next iteration, the k-NN$_P$ size is equal to $k \times \#B$, where $\#B$ is the number of blocks.

TopK_AA has been designed taking into account the GPU characteristics such as shared memory, coalesced access to global memory and number of threads by block. In the code we avoid central decisions. For example each thread will compute where in the shared memory she can write and when she is among the top k results. Using atomic functions over the shared memory is faster than serialization over the global memory operations. Even more, the shared memory of the GPU is faster than the RAM memory in the CPU.

2.2 QuickSelect

The well known QuickSelect algorithm finds the x smallest elements of an unsorted array of z elements. In this case, *Select_k* stage applies the *QuickSelect* (*QSeP*) algorithm to choose the k-NN objects of a local list (in shared memory) by block. This process is applied iteratively while the pivot position is not equal to k. For each iteration, we select the pivot (it is a mean of three values in the local list), and partition the local list. If pivot position is equal to k, the partition with smaller elements than pivot is the k-NN of local list. Otherwise, two

cases are possible: the position pivot is greater than k, or it is lower than k. In the first case, we run the *QuickSelect* over the partition with smaller elements. Otherwise, we work only over the second partition of local list (it has elements larger than the pivot).

2.3 TopK_AA + QuickSelect

This implementation of Select k stage is a combination of the two routines above. We called as QSeH (H for Hybrid). In this case, the first time we select the k-NN elements of local list applying QuickSelect algorithm (The local list is equal to distances list calculated in Distances stage). In the next steps, we determine the local k-NN using TopK_AA algorithm. We tried this implementation to see if the combination of the two approaches can boost the performance.

2.4 Thrust Library

There are many Sort libraries for CUDA, one of them is *Thrust*. It is part of the CUDA repositories. In this solution, we used the *Thrust* library to obtain the k-NN. Using the *Thrust* library is just a global sort, the k-NN stage is replaced by a call to *sort* algorithm of *Thrust* library, its output is the k-NN of each q. *Sort* algorithm is a black box, its implementation details are hidden to the user.

3 Solving Many *k*-NN Queries in Parallel

In large-scale systems it is not enough to speedup one query at a time, but it is necessary to leverage the capabilities of the GPU to answer in parallel several queries. Thereby 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 the GPU.

In order to answer many queries in parallel, the GPU receives a query set and solves all of them at once. Each query, in parallel, applies a k-NN routing as explained in the previous sections, therefore the number of resources for this is equal to the resource 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 the available memory.

Solving many queries in parallel involves carefully managing 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 is solved and which database element correspond to the query outcome. This can be easily accomplished by establishing a relationship among *Thread Id*, *Block Id*, *Query Id*, and *Database Element*.

4 The *All-k*-NN Problem

We need to determine the k-NN for each object in the database. Since we have solved one instance, it is enough to iterate through all the elements in the database considering them as queries. One just need to be careful to codify the results appropriately before sending the results back to the CPU. Considering these characteristic and the possibility of solving many queries in parallel, the solution is simple. If X queries are answered in parallel, we can compute *All-k*-NN in $\frac{N}{X}$ iterations, the value of X depends of GPU characteristics and the size of database.

5 Experimental Results

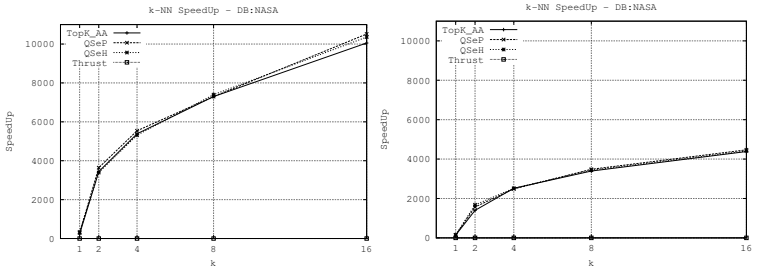
In this section we show and analyze the experimental results for the solutions of k-NN and *All-k*-NN problems. Please notice that range queries are trivial since it naturally avoids crosstalk between threads. We have not included this experiments, but they are faster than 1-NN queries. We selected two widely used benchmark databases from the SISAP Metric Library (`www.sisap.org`).

- *NASA images:* a set of 40,700 20-dimensional feature vectors, generated from images downloaded from NASA. The distance is euclidean.
- *COLORS histograms:* a set of 112,682 feature vectors of dimension 112, obtained from color histograms from an image database. Any quadratic form can be used as a distance, so we chose Euclidean distance as the simplest meaningful alternative.

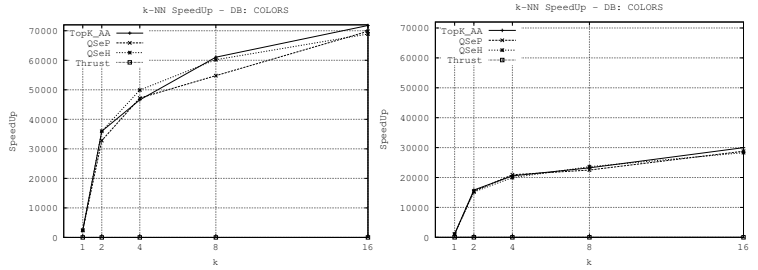
Each reported value is the average of many executions of the corresponding algorithm. We use for both databases k values of: 1, 2, 4, 8, and 16. The analysis was made for two generations of GeForce GPU whose characteristics (Global Memory, SM, SP, Clock rate, Compute Capability) are GTX330: (512MB,6,8,1.04GHz,1.2), GTX470: (1280MB, 14, 32, 1.2GHz, 2.0) and GTX550Ti: (1024MB, 4, 48, 1.96GHz,2.1). The CPU is an Intel core i3, 2.13 GHz and 3 GB of memory. The results are expressed in Speed up ($Sp = \frac{Time_{Sec}}{Time_{Par}}$). For lack of space, the shown results are only on architectures GTX330 and GTX470.

To evaluate the behavior of our solutions against a good sequential solution, we did choose the SAT^+ metric index. It is a new version of *Spatial Approximation Tree* (*SAT*) [21], improved by a new selection order of the neighbors in the tree (distal nodes). SAT^+ has shown to be one of the most competitive exact proximity searching index in the literature, and it is very scalable and resistant to the curse of dimensionality [22]. It is implemented in the *C* language.

Figure 2 shows the obtained speed up by k-NN queries for two before mentioned metric spaces. For all GPUs, we achieve very good results. The best speed up is obtained by our implementation, it can noticed that the *Thrust* solution reaches a significantly lower speedup compared to the others three proposed solutions.

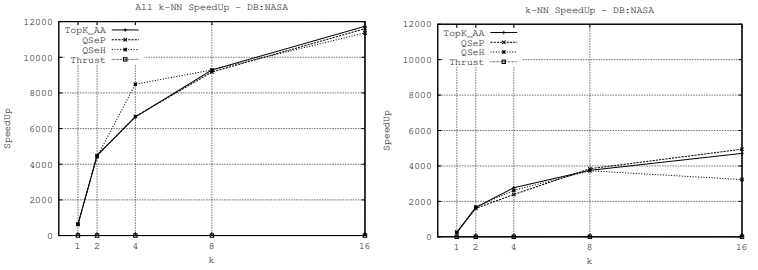


(a) GTX 470 (b) GTX 330M

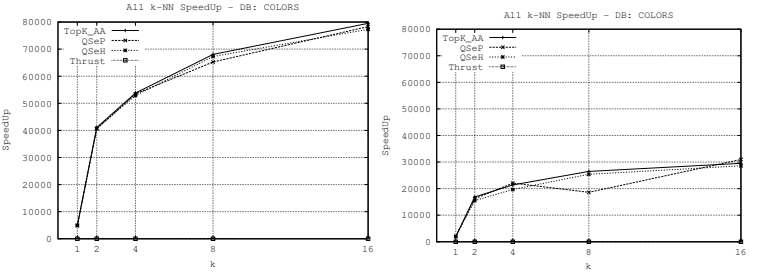


(c) GTX 470 (d) GTX 330M

Fig. 2. Speedup of k-NN query implementations for NASA and COLORS databases, on two different GPUs



(a) GTX 470 (b) GTX 330M



(c) GTX 470 (d) GTX 330M

Fig. 3. Speedup of All-k-NN query implementations for NASA and COLORS databases, on two different GPUs

The accelerations are different because they depend of GPU resources, but in all cases they are significant. Figure 3 resumes the behavior of four proposed solutions of *All-k*-NN query. Like k-NN solutions, the accomplished speedups by our three implementation are much higher than the one obtained using *Thrust*.

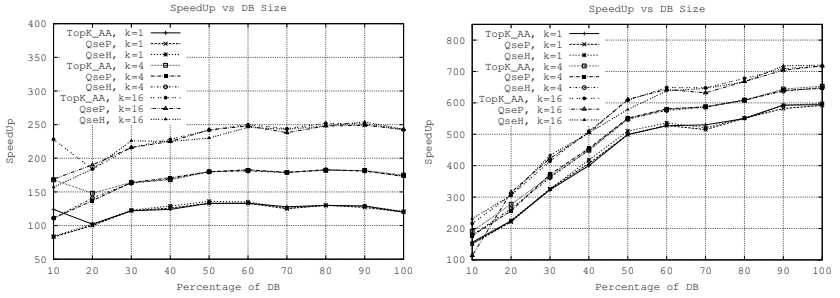
The speedup increase as both the database size and k, the number of neighbors, increase. Tables 1 and 2 show the obtained throughput (number of queries by second) by all implementations evaluated. Table 1 illustrates the throughput obtained for the sequential index used SAT^+. For lack of space, we only report the results obtained for $k = 1$ and $k = 16$. The results clearly show the benefits for all GPU architectures. In every case and k value, the number of queries by second is impressively high.

Table 1. Throughput of k-NN query for SAT^+

DB	k	PC
NASA	1	35160,14
	16	1219,08
COLORS	1	4608,80
	16	163,49

Table 2. Throughput of k-NN query, for four algorithms in two GPUs

Algorithm	k	NASA		COLORS	
		GTX 470	GTX 330M	GTX 470	GTX 330M
TopK_AA	1	6779661,02	3883495,15	6106870,23	3827751,20
	16	6851612,90	3919909,50	6666666,67	3903703,70
QSeP	1	9913258,98	4938271,60	9720534,63	4040404,04
	16	11428571,43	4993449,78	10624169,99	4444444,44
QSeH	1	7272727,27	3960396,04	6153846,15	3418803,42
	16	7666666,67	3982439,02	7272727,27	3448275,86
Thrust	1	94,35	41,31	94,35	41,31
	16	94,45	41,35	94,45	41,35



(a) SP without construction cost (b) SP with construction cost

Fig. 4. Speedup of k-NN query implementations for COLORS databases, considering increasing database sizes

6 Conclusions, Remarks and Future Work

We have shown a very fast algorithm for the k-Nearest Neighbors and the All k-Nearest Neighbors problems. Our proposal is several orders of magnitude faster than state of the art sequential and massively parallel approaches. We have not included other GPU implementations in our benchmark because they used different hardware, we can only deduct from the published results that the speedup obtained is very modest. Our dramatic speedup is due to a combination of faster memory, non blocking parallel processing and adherence to the design principles. A more detailed experimental evaluation is needed to know if the speedup can be considered super linear on the number of processing units.

We do not need to build an index beforehand, widening the applications of our algorithms. Also notice that our approach can easily solve range queries, much faster than nearest neighbor queries because Top-k selection is skipped. We are working on a randomized (exact) version of the k-NN algorithm using repeated calls to a range query routine. Others properties of our approach is the absence of the curse of dimensionality, and the possibility to query non-metric (dis)similarity databases.

References

1. Zezula, P., Amato, G., Dohnal, V., Batko, M.: Similarity Search: The Metric Space Approach. Advances in Database Systems, vol. 32. Springer (2006)
2. Samet, H.: Foundations of Multidimensional and Metric Data Structures. The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling. Morgan Kaufmann Publishers Inc., San Francisco (2005)
3. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquín, J.: Searching in metric spaces. ACM Comput. Surv. 33(3), 273–321 (2001)
4. Benjamin, B., Navarro, G.: Probabilistic proximity searching algorithms based on compact partitions. Discrete Algorithms 2(1), 115–134 (2004)
5. Kirk, D., Hwu, W.: Programming Massively Parallel Processors, A Hands on Approach. Elsevier, Morgan Kaufmann (2010)
6. Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., Phillips, J.: GPU Computing, pp. 879–899. IEEE (2008)
7. NVIDIA. Nvidia cuda compute unified device architecture, programming guide version 4.2. In: NVIDIA (2012)
8. Barrientos, R., Gomez, J., Tenllado, C., Prieto, M.: Heap based k-nearest neighbor search on gpus. In: XXI Jornadas de Paralelismo, pp. 559–566 (September 2010)
9. Garcia, V., Debreuve, E., Barlaud, M.: Fast k nearest neighbor search using GPU. In: CVPR Workshop on Computer Vision on GPU (CVGPU), Anchorage, Alaska, USA (June 2008)
10. Garcia, V., Debreuve, E., Nielsen, F., Barlaud, M.: k-nearest neighbor search: fast GPU-based implementations and application to high-dimensional feature matching. In: IEEE International Conference on Image Processing, Hong Kong (September 2010)
11. Kato, K., Hosino, T.: Solving k-nearest neighbor problem on multiple graphics processors. In: ACM (ed.) 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGRID, pp. 769–773 (2010)

12. Kuang, Q., Zhao, L.: A practical gpu based knn algorithm. In: International Symposium on Computer Science and Computational Technology (ISC-SCT), pp. 151–155 (2009)
13. Liang, S., Liu, Y., Wang, C., Jian, L.: Design and evaluation of a parallel k-nearest neighbor algorithm on CUDA-enabled GPU. In: IEEE 2nd Symposium on Web Society (SWS), p. 53 (2010)
14. Rozen, T., Boryczko, K., Alda, W.: Gpu bucket sort algorithm with applications to nearest-neighbour search. Journal of WSCG 16(1-3), 161–167 (2008)
15. Barrientos, R., Gomez, J., Tenllado, C., Prieto, M.: Query processing in metric spaces using gpus. In: XXII Jornadas de Paralelismo (2011)
16. Barrientos, R.J., Gómez, J.I., Tenllado, C., Matias, M.P., Marin, M.: kNN Query Processing in Metric Spaces Using GPUs. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part I. LNCS, vol. 6852, pp. 380–392. Springer, Heidelberg (2011)
17. Zhou, K., Hou, Q., Wang, R., Guo, B.: Real-time kd-tree construction on graphics hardware. In: ACM SIGGRAPH Asia, Papers, SIGGRAPH Asia 2008, pp. 126:1–126:11. ACM, New York (2008)
18. Brown, S., Snoeyink, J.: Gpu nearest neighbors using a minimal kd-tree. In: Second Workshop on Massive Data Algorithmics (MASSIVE 2010), Snowbird, Utah, USA (June 2010)
19. Qiu, D., May, S., Nüchter, A.: Gpu-accelerated nearest neighbor search for 3d registration. In: Fritz, M., Schiele, B., Piater, J.H. (eds.) ICVS 2009. LNCS, vol. 5815, pp. 194–203. Springer, Heidelberg (2009)
20. Djinevski, S.R.L., Gusev, M.: Superlinear speedup for matrix multiplication in gpu devices. In: ICT Innovations 2012, pp. 285–294 (2013)
21. Navarro, G.: Searching in metric spaces by spatial approximation. The Very Large Databases Journal (VLDBJ) 11(1), 28–46 (2002)
22. Chavez, E., Ludueña, V., Reyes, N., Roggero, P.: Faster proximity searching with the distal sat (2012) (submitted, draft)