kNN Query Processing in Metric Spaces Using GPUs

Ricardo J. Barrientos[1], José I. Gómez[1], Christian Tenllado[1], Manuel Prieto Matias[1], and Mauricio Marin[2]

[1] Architecture Department of Computers and Automatic, ArTeCS Group, Complutense University of Madrid, Madrid, España
ribarrie@fdi.ucm.es
[2] Yahoo! Research Latin America, Santiago, Chile
mmarin@yahoo-inc.com

Abstract. Information retrieval from large databases is becoming crucial for many applications in different fields such as content searching in multimedia objects, text retrieval or computational biology. These databases are usually indexed off-line to enable an acceleration of on-line searches. Furthermore, the available parallelism has been exploited using clusters to improve query throughput. Recently some authors have proposed the use of Graphic Processing Units (GPUs) to accelerate brute-force searching algorithms for metric-space databases. In this work we improve existing GPU brute-force implementations and explore the viability of GPUs to accelerate indexing techniques. This exploration includes an interesting discussion about the performance of both brute-force and indexing-based algorithms that takes into account the *intrinsic dimensionality* of the element of the database.

1 Introduction

Similarity search has been widely studied in recent years and it is becoming more and more relevant due to its applicability in many important areas. Efficient kNN search, namely k nearest-neighbors search, is useful in multimedia information retrieval, data mining or pattern recognition problems. In general, when similarity search is undertaken by using metric-space database techniques, this problem is often featured by a large database whose objects are represented as high-dimensional vectors. A distance function operates on those vectors to determine how similar the objects are to a given kNN query object. The distance between any given pair of objects (i.e. high-dimensional vectors) is known to be an expensive operation to compute and thereby the use of parallel computation techniques can be an effective way to reduce running times to practical values in large databases.

In this paper we propose and evaluate efficient metric-space techniques to solve kNN search on GPUs. Obtaining efficient performance from this hardware can be particularly difficult in our application domain since metric-space solutions developed for traditional shared memory multiprocessors and distributed systems [16] cannot be implemented efficiently on GPUs.

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 380–392, 2011.

Our focus is on search systems devised to solve large streams of queries. Conventional parallel implementations for clusters and multicore systems that exploit corse-grained inter-query parallelism are able to improve query throughput by employing index data structures constructed off-line upon the database objects. On GPUs we are able to exploit fine-grained parallelism and it can be more efficient to just resort to brute-force algorithms, especially for high-dimensional metric-spaces. The interesting problem to solve in this case is how to reduce the amount of work required to keep track of the current objects making into the kNN set. We propose a realization of this approach that outperforms alternative approaches to brute-force based on global ordering of the distances of database objects to the query. Our proposal keeps a partial ordering of objects which results from applying a novel strategy based on parallel priority queues.

We also experimented with a couple of metric-space index data structures that we have found amenable for GPU parallelization as they allow matrix-like computations. Finding a way of mapping these indexes onto GPUs resulted quite complex and tricky (the main difficulty is to exploit the available memory bandwidth), and thereby a second contribution of this paper is the proposal of a GPU based metric-space index data structure for similarity search.

The remaining of this paper is organized as follows. Section 2 gives some background information on similarity search and metric-space databases. Section 3 describes the main features of our computing platform and summarizes some previous related work. In Section 4 and 5 we introduce our proposals and discuss the most important finding from a performance evaluation against baseline strategies. We conclude in Section 6 highlighting our main contributions.

2 Similarity Search Background and Related Work

A *metric space* (X, d) is composed of an universe of valid objects $\mathbb{X}$ and a *distance function* $d : \mathbb{X} \times \mathbb{X} \rightarrow \mathbb{R}^+$ defined among them. The distance function determines the similarity between two given objects and holds several properties such as strict positiveness, symmetry, and the triangle inequality The finite subset $\mathbb{U} \subset \mathbb{X}$ with size $n = |\mathbb{U}|$, is called the database and represents the collection of objects of the search space.

There are two main queries of interest: ***Range Search*** [6] and ***The k nearest neighbors (kNN)*** [1,8]. In the former, the goal is to retrieve all the objects $u \in \mathbb{U}$ within a radius r of the query q (i.e. $(q, r)_d = \{u \in \mathbb{U} / d(q, u) \leq r\}$), whereas in the latter, the goal is to retrieve the set $kNN(q) \subseteq \mathbb{U}$ such that $|kNN(q)| = k$ and $\forall u \in kNN(q), v \in \mathbb{U} - kNN(q), d(q, u) \leq d(q, v)$.

For solving both kind of queries and to avoid as many distance computations as possible, many indexing approaches have been proposed. We have focused on the *List of Clusters* (LC) [5] and *SSS-Index* [2] strategies since (1) they are two of the most popular non-tree structures that are able to prune the search space efficiently and (2) they hold their indexes on dense matrices which are very convenient data structures for mapping algorithms onto GPUs [9].

In the following subsections we explain the construction of both indexes and describe how range queries are solved using them (range searches are simpler than kNN, but many kNN searches are built on them).

2.1 List of Clusters (LC)

This index [5] is built by choosing a set of centers $c \in U$ with radius r_c where each center maintains a bucket that keeps tracks of the objects contained within the ball (c, r_c). Each bucket holds the closest k-elements to c. Thus the radius r_c is the maximum distance between the center c and its k-nearest neighbor.

The buckets are filled up sequentially as the centers are created and thereby a given element i located in the intersection of two or more center balls remains assigned to the first bucket that hold it. The first center is randomly chosen from the set of objects. The next ones are selected so that they maximize the sum of the distances to all previous centers.

A range query q with radius r is solved by scanning the centers in order of creation. For each center $d(q, c)$ is computed and only if $d(q, c) \leq r_c + r$, it is necessary to compare the query against the objects of the associated bucket. This process ends up either at the first center that holds $d(q, c) < r_c - r$, meaning that the query ball (q, r) is totally contained in the center ball (c, r_c), or when all centers have been considered.

2.2 Sparse Spatial Selection (SSS-Index)

During construction, this pivot-based index [2] selects some objects as *pivots* from the collection and then computes the distance between these pivots and the rest of the database. The result is a table of distances where columns are the pivots and rows the objects. Each cell in the table contains the distance between the object and the respective pivot. These distances are used to solve queries as follows. For a range query (q, r) the distances between the query and all pivots are computed. An object x from the collection can be discarded if there exists a pivot p_i for which the condition $|d(p_i, x)d(p_i, q)| > r$ does hold. The objects that pass this test are considered as potential members of the final set of objects that form part of the solution for the query and therefore they are directly compared against the query by applying the condition $d(x, q) \leq r$. The gain in performance comes from the fact that it is much cheaper to effect the calculations for discarding objects using the table than computing the distance between the candidate objects and the query.

A key issue in this index is the method that calculates the pivots, which must be good enough to drastically reduce total number of distance computations between the objects and the query. An effective method is as follows. Let $(\mathbb{X}, d)$ be a metric space, $\mathbb{U} \subset \mathbb{X}$ an object collection, and M the maximum distance between any pair of objects, $M = max\{d(x, y)/x, y \in \mathbb{U}\}$. The set of pivots contains initially only the first object of the collection. Then, for each element $x_i \in \mathbb{U}$, x_i is chosen as a new pivot if its distance to every pivot in the current set of pivots is equal or greater than αM, being α a constant parameter. Therefore,

an object in the collection becomes a new pivot if it is located at more than a fraction of the maximum distance with respect to all the current pivots.

3 Graphic Processing Units (GPU)

GPUs have emerged as a powerful cost-efficient many-core architecture. They integrate a large number of functional units following a SIMT model. We develop all our implementations using NVIDIA graphic cards and its CUDA programming model ([9]). A CUDA *kernel* executes a sequential code on a large number of threads in parallel. Those threads are grouped into fixed size sets called *warps*[1]. Threads within a *warp* proceed in a lock step execution. Every cycle, the hardware scheduler of each GPU multiprocessor chooses the next warp to execute (i.e. no individual threads but warps are swapped in and out). If the threads in a warp execute different code paths, only those that follow the same path can be executed simultaneously and a penalty is incurred.

Warps are further organized into a grid of *CUDA Blocks*: threads within a block can cooperate with each other by (1) efficiently sharing data through a shared low latency local memory and (2) synchronizing their execution via barriers. In contrast, threads from different blocks can only coordinate their execution via accesses to a high latency global memory. Within certain restrictions, the programmer specifies how many blocks and how many threads per block are assigned to the execution of a given kernel. When a kernel is launched, threads are created by hardware and dispatched to the GPU cores.

According to NVIDIA the most significant factor affecting performance is the bandwidth usage. Although the GPU takes advantage of multithreading to hide memory access latencies, having hundreds of threads simultaneously accessing the global memory introduces a high pressure on the memory bus bandwidth. The memory hierarchy includes a large register file (statically partitioned per thread) and a software controlled low latency shared memory (per multiprocessor). Therefore, reducing global memory accesses by using local shared memory to exploit inter thread locality and data reuse largely improves kernel execution time. In addition, improving memory access patterns is important to allow coalescing of warp loads and to avoid bank conflicts on shared memory accesses.

4 GPU Mapping of *k*-Nearest Neighbor Algorithms

In this section we describe the mapping of three k-NN algorithms onto CUDA-enabled GPUs: a brute-force approach and two index-based search methods.

All of them exploit two different levels of parallelism. As in some previous papers [11][15][7] we assume a high frequency of incoming queries and exploit coarse-grained inter-query parallelism. However, we also exploit the fine-grained parallelism available when solving a single query. Overall, each query is processed by a different CUDA Block that contains hundreds of threads (from 128 to 512,

[1] Currently, there are 32 threads per *warp*.

depending of the specific implementation) that efficiently cooperate to solve it. Communication and synchronization costs between threads within the same CUDA Block are rather low, so this choice looks optimal to fully exploit the enormous parallelism present in k-NN algorithms.

Another common point of our three implementations is the usage of priority queues (implemented using a heap [12]) to keep track of the potential candidates found by the threads. This avoids the sorting of the full vector of distances at the cost of a final reduction stage (described in subsection4.1), as well as increasing the irregularity of the accesses. Nevertheless, data locality is optimized as much as possible holding queries and heaps in shared memory whenever possible.

4.1 Exhaustive Search Algorithm

k-NN is typically implemented on GPUs using brute force methods applying a two-stage scheme. First, all the distances from the target query to the elements on the database are evaluated and then a second stage sorts these distances to obtain the nearest elements. In [10], the final stage is implemented with a modified parallel insertion sort in order to just get the k closest elements, whereas in [13] authors used an improved Radix-sort. In both cases, full GPU resources are employed to solve a single query. As mentioned above, we assume a high frequency of incoming queries and also exploit a coarser level of parallelism by solving several queries simultaneously. In this case, the first stage becomes a matrix multiplication ($Q * U$, where Q rows hold the queries and U columns store the database), which can be implemented very efficiently on GPUs [19].

Our implementation of the *sorting* phase is based on a *logarithmic reduction* algorithm that consists of three steps. First, the distance vector is evenly distributed across the CUDA Block threads. To fulfill CUDA alignment and coalescing requirements, elements are assigned in a round-robin fashion such that concurrent accesses of threads within a warp are performed to consecutive memory addresses. Each thread keeps its own private heap to store its partial K results (i.e. the K minimal values found in its part of the vector). For real size problems, the size of hte distance vector is higher than K; thus, the time to fill each heap is almost negligible. In the *steady state*, each thread must compare the new distance with the top of the heap. Just in case this distance is lower than the current top, a heap insertion is performed. As computation evolves, is less and less likely to find smaller elements. At that point, memory accesses become very regular and threads within a warp almost never diverge.

The other two stages implement the reduction of the local heaps and are common to all our implementations as mentioned above. The second step starts once all the threads in the CUDA Block have finished its assigned computations. A synchronization barrier is needed to ensure that all threads have finished before starting this step. The input to this stage is a set of *tpb* heaps, each filled with K elements (*tpb* stands for the number of threads per CUDA Block). Now, just one *warp* from the whole CUDA Block becomes active. At this point, we sacrifice parallelism to guarantee a better memory exploitation, which reveals to be much more relevant for final performance. Heaps are statically assigned, again

in a round-robin fashion, to each individual thread within the *warp*. Each thread will traverse its set of heaps, keeping the K minimal values found. Once again, a heap is used to store temporal results and the output of this step consists of 32 (the warp size) heaps filled with K elements. For the investigated values of K, these heaps can be allocated on the shared memory.

In the last step, a single thread performs the reduction of the K smaller values out of the 32 heaps. There is no need of explicit synchronization between steps in this case, due to the locked-step execution of threads in the same warp during the second step. The final results are also stored on a heap, allocated in shared memory (in order to keep the memory footprint as low as possible, in-place mapping is performed if the set of 32 heaps from step 2 were also allocated in shared memory). Both in the second and third step we exploit the fact that input elements are already organized in heaps, which allows us to heavily reduce the number of comparisons and insertions in the new output heap.

4.2 LC

The data structure that holds the LC index consists of 3 arrays denoted as *CENTER*, *RC* and *CLUSTERS*. *CENTER* is a $D \times N_{cen}$ matrix (D is the dimension of the elements[2] and N_{cen} is the number of centers), where each column represents the center of a cluster, *RC* is an array that stores the covering radius of each cluster, and *CLUSTERS* is a $D \times (B_{size} \cdot N_{clu})$ matrix (N_{clu} is the number of clusters and B_{size} is the number of elements per clusters) that holds the elements of each cluster. Index information is stored column-wise to favor coalesce memory accesses.

kNN queries are solved with LC indexes using auxiliary range queries with increasing or decreasing radius. The former starts performing a range query with a given initial range R_{ini} and repeat those range queries increasing the radius by Δ until the nearest K-elements are found[3]. The latter starts with $R_{ini} = \infty$ and performs a single iteration in which the radius is reduced dynamically. Sequential implementations usually employ the decreasing alternative since it provides better performance [6]. However the increasing radius strategy exhibits higher inherent parallelism and we have analyzed both of them. In both cases, parallelism comes from the distribution of distance evaluations.

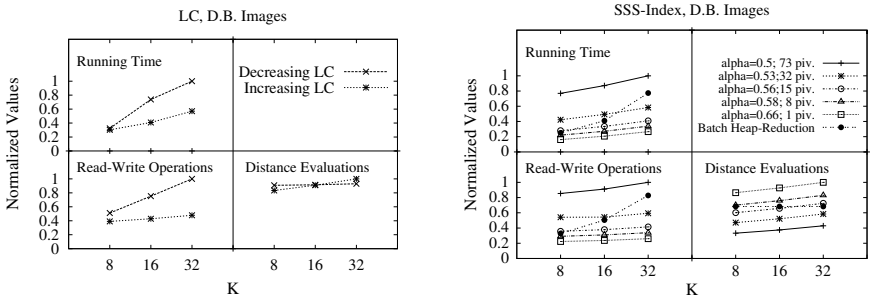
In our implementations, all threads evaluate first the distance between the assigned query q and a subset of the elements of *CENTER* following a Round-Robin distribution of this matrix. Based on these distances some centers and their respective clusters are discarded using triangle inequality. For clusters cannot be discarded, more distances are evaluated in parallel following again Round-Robin distribution of the *CLUSTERS* matrix. As in the exhaustive approach, each thread holds potential candidates in a local heap, which are finally reduced using the same logarithmic strategy. For the decreasing radius strategy we start

[2] For the *Spanish* database, D is the maximum size of a word.

[3] Both parameters R_{ini} and Δ are usually set empirically with an off-line analysis of the database using a small sample of its elements. In our experiments we have used less than 1% of their elements to set them.

with $R_{ini} = \infty$ and perform some initial updates of this radius when threads fill their local heaps for the first time. At these events, the radius is adjusted with the longest distance processed so far by the thread (i.e. the root of its local heap) using CUDA `atomicMin`($radius_{current}$, $local_{longest-distance}$) function. Later on, the radius is adjusted similarly whenever a new element is inserted on a local heap. The *current radius* is always used to test if the elements of a given cluster can be discarded using the triangle inequality.

As shown in Figure 1(a), although both methods are able to discard a similar proportion of the database collection, the increasing strategy outperforms the decreasing counterpart. Note that in a sequential setting, the decreasing approach scans the database elements in order and may adjusts the current radius after processing every single element. However, in a parallel setting these updates are performed at a higher granularity since many elements are processed in parallel. Therefore, the current radius decreases *more slowly* and less clusters are discarded. Furthermore, warps divergences are more expensive in the decreasing strategy. They occur when new elements are inserted into local heaps and in the decreasing method involve an additional atomic instruction in order to adjust the radius. As a consequence of these penalties, the decreasing method exhibits a worse memory behavior since costly divergences prevent coalesce memory accesses.



(a) (b)

Fig. 1. Performance of kNN queries using a) *LC* with both increasing and decreasing radius range queries and b) *SSS-Index* with different number of pivots.

4.3 SSS-Index

We have used 3 matrices to implement *SSS-Index*: *PIVOTS*, *DISTANCES* and *DB*. *PIVOTS* is a $D \times N_{piv}$ matrix (D is the dimension of the elements and N_{piv} is the number of pivots) where each column represents a pivot. *DISTANCES* is a $N_{piv} \times N_{DB}$ matrix (N_{DB} = number of elements of the database) where each row holds the distance vector between pivots and an element of the database. *DB* is the reference database. As in the *LC*, the index information is stored column-wise to favor coalesce memory accesses.

As centers in LC, pivots are distributed across threads following a round-robin distribution to evaluate their distance with the query. On a later stage, the rows of *DISTANCES* are distributed across threads, that test if their respective elements of the database can be discarded. For every non discarded element, a distance evaluation is performed and a set of heaps is filled accordingly (as in previous implementations).

In [2], authors have found empirically that $\alpha = 0.4$ yields the minimal number of distance evaluations. Our own experiments on GPUs confirm this behavior: the more pivots are used (up to a certain threshold), the less distance evaluations are performed. However, as shown in Figure 1(b), the best performance is obtained with just one pivot. Indeed the more pivots used, the worst the execution time becomes. *Irregularity* explains this apparent contradiction: when using more pivots, threads within a warp are more likely to diverge. Moreover, memory access pattern becomes more irregular and hardware cannot coalesced them. This leads to the observed increase in the number of Read/Write operations. Summarizing, less distance evaluations do not pay off due to the overheads caused by warp divergences and irregular access patterns. Overall, just one pivot provides the optimal performance for many of our reference databases.

5 Experimental Results

As computing platforms we have used a NVIDIA GeForce GTX 280 GPU (30 multiprocessors, 8 cores per multiprocessor, 16K of shared memory) equipped with 4GB of device memory and an Intel's Clovertown processor with 16 GB of RAM. We have use three different reference databases (described below) and the parameter K (the number of nearest neighbors) has been set to 8, 16 and 32. Similar values have been also used in previous papers [13][10][3].

***Spanish*:** A Spanish dictionary with 51,589 words and we used the *edit distance* [14] to measure similarity. On this metric-space we processed 40,000 queries selected from a sample of the Chilean Web which was taken from the TODOCL search engine. This can be considered a low dimensional metric space.

***Images*:** We took a collection of images from a NASA database containing 40,700 images vectors, and we used them as an empirical probability distribution from which we generated a large collection of random image objects containing 120,000 objects. We built each index with the 80% of the objects and the remaining 20% objects were used as queries. In this collection we used the *euclidean distance* to measure the similarity between two objects. Intrinsic dimensionality of this space higher than in the previous database, but it is still considered low.

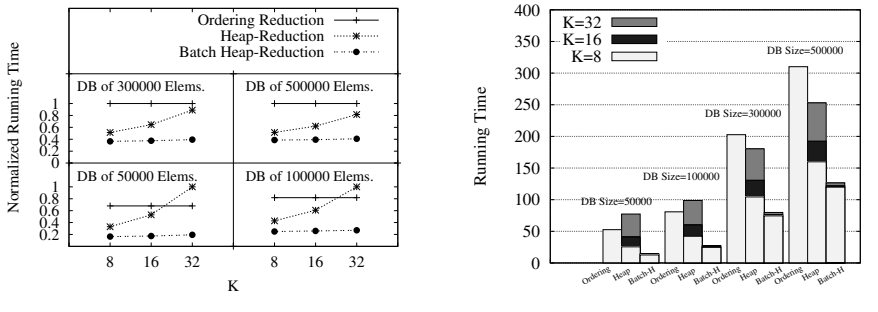
***Faces*:** This database was created from a collection of 8480 face images obtained from Face Recognition Grand Challenge [17]. We apply the *Eigen Face Method* [18] to obtain a projection matrix, that can be used to generate a feature vector from any face image. We used this collection as an empirical probability distribution from which we generated a large collection of random face image objects containing 120,000 objects. We used the *euclidean distance* to measure the similarity between two objects. Each *eigenface* consist in a vector of 254 elements.

Even if the intrinsic dimensionality of the space may be lower than 254, it is large enough to ruin traditional indexing benefits.

Figure 2 compares different exhaustive search methods. *Ordering reduction* stands for the state-of-the-art solution: all the distance evaluations are performed first. Next, the whole GPU is devoted to sort the obtained distances per query (i.e., queries are solved one at a time, and full resources are employed to sort a single vector of distances). A very efficient parallel version of the *quicksort* algorithm is employed at that step [4]. *Batch-Heap Reduction* corresponds with the technique explained in Section 4.1: one CUDA Block solves a single query and multiple queries are solved in parallel. Finally, we include a third version labeled *Heap-Reduction* that follows our implementation but solves just one query at a time. Figure 2(a) compares normalized running times for different reference database size and different values of K. The points are normalized to the largest value of the experiment. Figure 2(b) shows the absolute running time of the same set of experiments.

Our proposals are able to outperform the *Ordering* counterpart in most experiments. Even if we solve one query at a time (*Heap-Reduction* in the Figure) we outperform sorting-based algorithms for large databases and small values of K due to a better memory management. When we exploit the full strength of GPU launching as many CUDA Blocks as queries the difference increases and, more relevant, our implementation becomes much less sensitive to K. Note the performance of any sorting-based implementation is independent of K, since they sort the full distance vector).

We now turn our attention to the proposed indexing algorithms. For LC, the preliminary analysis presented in Section 4.2 motivated us to pick 64 as the number of elements per cluster in the high-dimensional database, while lowering it to 32 in the *Spanish* database. Similarly, we restrict ourselves to the increasing radius approach since it always perform better than the decreasing counterpart. Regarding *SSS-Index*, and following the conclusions drawn in Section 4.3, we

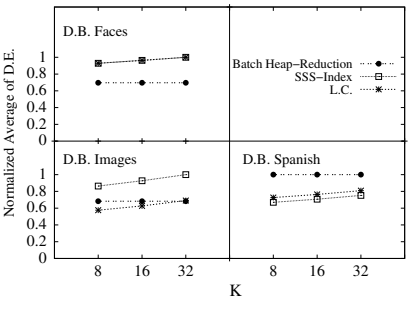


(a) (b)

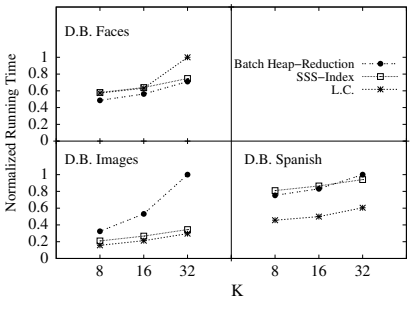
Fig. 2. Normalized (a) and absolute (b) running times of the investigated exhaustive search algorithms for different K and number of elements using *Faces* database

use just a single pivot ($\alpha = 0.66$) for vector high-dimensional databases and 68 pivots ($\alpha = 0.5$) for *Spanish* database.

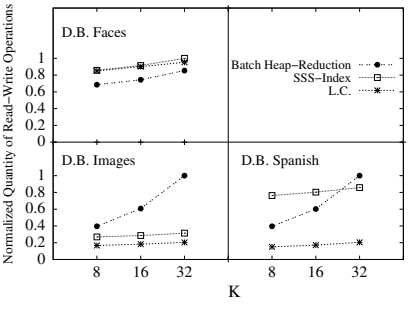
Figure 3 shows different set of results to illustrate several important findings of our three implementations. We first place our attention on the total number of distance evaluations (Figure 3(a)). The *Spanish* database behaves as expected: indexing mechanisms do significantly decrease the number of distance evaluations when compared to the exhaustive search method. However, as space dimensionality increases, that is no longer the case. Indeed the opposite behavior is observed: indexing mechanisms perform more distance evaluations than the exhaustive algorithm. This is specially true for *SSS-Index* with just one pivot. Obviously, this fact implies that some evaluations are performed more than once with the indexing mechanism, which is possible due to the increasing radius approach. It must be noted that we intentionally decided not to reuse distance evaluations to avoid repeated computations since it introduces an enormous source of irregularity. On GPUs, decreasing the amount of work in this way, does not pay off.



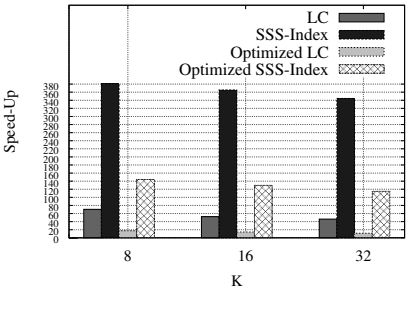
(a) Number of Distance Evaluations



(b) Running time



(c) Number of read/write operations



(d) Speed-Up of *LC* and *SSS-Index*

Fig. 3. Normalized a) Distance evaluations per query (average) b) Running time and c) Read-write Operations (of 32, 64 o 128 bytes) to *device memory*. d) Speed-Up of *LC* and *SSS-Index* over sequential counterparts with DB *Images*.

One would expect that running times mimic the trend exhibited by the distance evaluations but results in Figure 3(b) partially contradicts this intuition: the exhaustive search algorithm behaves worse than expected, specially for the *Images* database. Figure 3(c) has the clue: memory access pattern, which heavily influences performance on current GPUs, behaves better for the indexing mechanism, specially for *LC*. As stated in Section 3, when a warp launches misaligned or non-consecutive memory accesses, hardware is not able to coalesce it and a single reference may become up to 32 separate accesses. In all our implementations, heap insertions usually imply warp divergences and lack of locality, thus increasing the number of read/write operations. Indexed algorithms performs more distance evaluations but, since many distance evaluations are evaluated several times, the number of heap insertions is significantly reduced. However, as dimensionality increases the higher cost of these evaluations starts to tradeoff the difference in heap insertions. There, indexed mechanisms perform poorly and our exhaustive-search implementation outperforms both of them. The *Faces* database, the one with largest dimensionality in our experiments, illustrates this situation (see Figure 3(b)).

Finally, Figure 3(d) shows the performance speed-ups of our indexed implementations over its corresponding sequential implementations. For each of the two algorithms, we implemented a naive *out-of-the-box* version and a heavily tuned and compiler optimized one. Results are very impressive for *SSS-Index* due to the poor CPU performance of this indexing mechanisms. Regarding the optimized sequential version, we obtain up to 144x speedup for $k = 8$. But even for the lighter index (*List of Clusters*) our implementation achieves reasonable speedups of a 17x. Our experiments show that this speed-up is not easy to attend with OpenMP versions running on medium-sized clusters.

6 Conclusions

In this paper we have presented efficient implementations of typical indexing mechanisms together with an exhaustive-search version which are mapped on CUDA based GPUs.

We may highlight the following findings after our exploration: 1) when performing *kNN* search based on *range searches* in parallel, an *increasing radius* strategy becomes more efficient than the traditional *decreasing radius*. This is specially true for GPUs given their memory system restrictions. 2) Optimal parameters for both, *List of Clusters* and *SSS-Index* metrics are extremely different than those found on distributed implementations. In particular, the best GPU implementation found for *SSS-Index* uses a single pivot to prune the search space, which is highly inefficient since this pivot is found randomly. 3) In such a context, considering running time, our exhaustive-search proposal outperforms the indexed versions whereas for the lower dimension datasets our index strategies outperform exhaustive-search.

Note that from sequential computing literature we can learn that in metric-spaces with very high dimensions, indexing strategies are no longer useful as they

lose their ability to reduce running time. Here, the only option is to compare the query against the whole set of database objects. For that case, the exhaustive search strategy proposed in this paper is clearly the most efficient alternative for GPU based metric-space query processing.

References

1. Aha, D.W., Kibler, D.: Instance-based learning algorithms. In: Machine Learning, pp. 37–66 (1991)
2. Brisaboa, N.R., Fariña, A., Pedreira, O., Reyes, N.: Similarity search using sparse pivots for efficient multimedia information retrieval. In: ISM, pp. 881–888 (2006)
3. Bustos, B., Deussen, O., Hiller, S., Keim, D.A.: A graphics hardware accelerated algorithm for nearest neighbor search. In: Alexandrov, V.N., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2006. LNCS, vol. 3994, pp. 196–199. Springer, Heidelberg (2006)
4. Cederman, D., Tsigas, P.: Gpu-quicksort: A practical quicksort algorithm for graphics processors. J. Exp. Algorithmics 14, 1.4–1.24 (2009)
5. Chavéz, E., Navarro, G.: An effective clustering algorithm to index high dimensional metric spaces. In: The 7th International Symposium on String Processing and Information Retrieval (SPIRE 2000), pp. 75–86. IEEE CS Press, Los Alamitos (2000)
6. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquín, J.L.: Searching in metric spaces. In: ACM Computing Surveys, pp. 273–321 (September 2001)
7. Costa, V.G., Barrientos, R.J., Marín, M., Bonacic, C.: Scheduling metric-space queries processing on multi-core processors. In: Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP 2010), pp. 187–194. IEEE Computer Society, Pisa (2010)
8. Cover, T., Hart, P.: Nearest neighbor pattern classification. IEEE Transactions on Information Theory 13(1), 21–27 (1967), `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1053964`
9. CUDA: Compute Unified Device Architecture. ©2007 NVIDIA Corporation, `http://developer.nvidia.com/object/cuda.html`
10. Garcia, V., Debreuve, E., Barlaud, M.: Fast k nearest neighbor search using gpu. In: Computer Vision and Pattern Recognition Workshop, pp. 1–6 (2008)
11. Gil-Costa, V., Marin, M., Reyes, N.: Parallel query processing on distributed clustering indexes. Journal of Discrete Algorithms 7(1), 3–17 (2009)
12. Knuth, D.E.: The Art of Computer Programming, vol. 3. Addison-Wesley, Reading (1973)
13. Kuang, Q., Zhao, L.: A practical gpu based knn algorithm, Huangshan, China, pp. 151–155 (2009)
14. Levenshtein, V.: Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Doklady 10, 707–710 (1966)
15. Marin, M., Gil-Costa, V., Bonacic, C.: A search engine index for multimedia content. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 866–875. Springer, Heidelberg (2008)

16. Marin, M., Ferrarotti, F., Gil-Costa, V.: Distributing a metric-space search index onto processors. In: 39th International Conference on Parallel Processing, ICPP 2010, San Diego, California, pp. 13–16 (2010)
17. Phillips, P.J., Flynn, P.J., Scruggs, T., W., K., Bowyer, J.C., Hoffman, K., J., Marques, J.M., Worek, W.: Overview of the face recognition grand challenge. In: IEEE Computer Society Conference on Computer Vision and Pattern Recognition, CVPR 2005, vol. 1, pp. 947–954 (June 2005)
18. Turk, M., Pentland, A.: Eigenfaces for recognition. Journal of Cognitive Neuroscience 3(1), 71–86 (1991)
19. Volkov, V., Demmel, J.W.: Benchmarking gpus to tune dense linear algebra. In: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, SC 2008, pp. 31:1–31:11. IEEE Press, Piscataway (2008), http://portal.acm.org/citation.cfm?id=1413370.1413402