Range query processing in a multi-GPU environment

Ricardo J. Barrientos, José I. Gómez,
Christian Tenllado and Manuel Prieto Matias
Department of Computer Architecture, ArTeCS Group,
Complutense University of Madrid, Spain.
Email: ribarrie@fdi.ucm.es

Mauricio Marin
DIINF, University of Santiago of Chile
Yahoo! Research Latin America, Santiago, Chile.
Email: mmarin@yahoo-inc.com

***Abstract*—Similarity search has been widely studied in the last years, as it can be applied to several fields such as searching by content in multimedia objects, text retrieval or computational biology. These applications usually work on very large databases that are often indexed off-line to enable the acceleration of on-line searches. However, to maintain an acceptable throughput, it is essential to exploit the intrinsic parallelism of the algorithms used for the on-line query solving process, even with indexed databases. Therefore, many strategies have been proposed in the literature to parallelize these algorithms, both on shared and distributed memory multiprocessor systems. Lately, GPUs have also been used to implement brute-force approaches instead of using indexing structures, due to the difficulties introduced by the index in the efficient exploitation of the GPU resources. In this work we propose a Multi-GPU metric-space technique that efficiently exploits index data structures for similarity search in large databases, and show how it outperforms previous OpenMP and GPU brute-force strategies. Furthermore, our analysis covers the effects of the database size and its nature.**

I. 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 [1]. It is often undertaken by using metric-space techniques on large databases whose objects are represented as high-dimensional vectors. A distance function exists and operates on those vectors to determine how similar the objects are to a given query object. A range search with radius r for a query q, represented as (q, r), is the operation that obtains from the database the set of objects whose distance to the query object q is not larger than the radius r. However, the performance of this kind of searches is largely dominated by the computation of the distance function, which is known to be an expensive operation. The development of techniques that provide efficient range searches becomes crucial to promote the success of many applications that involve, among others, multimedia information retrieval, data mining or pattern recognition problems. Moreover, range search itself can be considered as a basic search kernel, as it enables other search operations (for instance, the *nearest neighbors search*), which reinforces the necessity of its performance boost.

In the current technological context, one of the most promising alternatives for the acceleration of range search operations is the exploitation of its intrinsic parallelism on Graphics Processing Units (GPUs). Range searches provide different levels of parallelism: we can process several queries in parallel, several distance computations in parallel for a given query or even exploit parallelism in the distance operation itself. This scheme matches well with the architecture of the GPU, that can execute a bunch of threads on a group of multiprocessors, which execute the threads in small groups in lock-step (similar to SIMD processors). Moreover, we currently have Multi-GPU systems available, providing an addition level for parallelism exploitation. However, these architectures have complex memory hierarchies, in which some of the levels can be controlled by software. It has been empirically shown that it is crucial to efficiently exploit this memory system to achieve a significant performance improvement when using GPUs to accelerate a given application.

Previous related work, which focusses on search systems devised to solve large streams of queries, has shown that conventional parallel implementations for clusters and multi-core systems, that exploit coarse-grained inter-query parallelism, are able to improve query throughput by employing index data structures constructed off-line upon the database objects [2]. These index structures are used to perform an efficient filtering on the database and reduce the search space. However, their use introduces a complex and irregular memory access pattern in the search algorithm, making it very inefficient for the GPU memory system. The cost of the additional data transfers introduced by using the index can hide the can hide the benefits of keeping the database objects smartly indexed.

To the authors knowledge, this paper presents the one of the first evaluations of GPUs as accelerators for metric-space range searches on large databases. In our study, we analyze the impact of the index structure used and the size and nature of the database. We first target the case in which the full database can be stored in the GPU memory (several GBs), and propose an efficient mapping scheme for two relevant index structures: *List of Clusters* [3] and *SSS-index* [4]. The performance achieved with these single-GPU range search methods is compared with optimized implementations of the same index based searches, both sequentially and in parallel, on leading edge shared memory multiprocessors. Once the best single-GPU strategy has been selected, we study its scalability to Multi-GPU systems and consider databases that do not fit in the device memory of a single GPU. The results show that Multi-GPU systems can be efficiently used to significantly improve throughput in metric-space range searches.

The remaining of this paper is organized as follows. Section II gives some background on similarity search and metric-

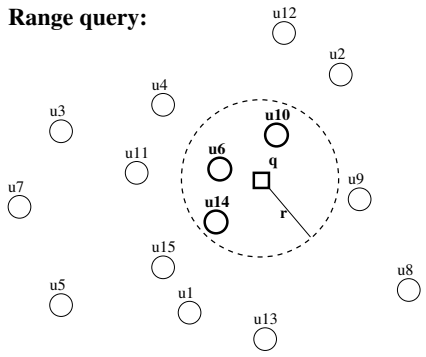


Fig. 1. Example of *range query* (*q*,*r*)

space databases, and summarizes some previous related work. Section III describes the main features of our computing platform and its programming model, while Sections IV and V describe the implementations of the range search methodologies under analysis in this paper, both on Single-GPU and shared memory multi-core systems. Section VI present the results of the analysis on single-GPU systems and Section VII analyzes its scalability to multi-GPU platforms. Finally, Section VIII summarizes the main conclusions of this work.

II. Similarity Search Background and Related Work

Searching objects from a database which are similar to a given query object is a problem that has been widely studied in recent years. The solutions are based on the use of a data structure that acts as an index to speed up the processing of queries. Similarity can be modeled as a metric space as stated by the following definitions:

Metric Space [5]: 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 ($d(x, y) > 0$ and if $d(x, y) = 0$ then $x = y$), symmetry ($d(x, y) = d(y, x)$), and the triangle inequality ($d(x, z) \leq d(x, y) + d(y, z)$). 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, kNN and *range* queries.

Range Query [1]: As shown in figure 1, the goal is to retrieve all the objects $u \in \mathbb{U}$ within a radius r of the query q (i.e. $(q, r)_d = \{u \in \mathbb{U} / d(q, u) \leq r\}$).

The k Nearest Neighbors (kNN) [6]: The goal is to retrieve the set $kNN(q) \subseteq \mathbb{U}$ such that $|kNN(q)| = k$ and $\forall u \in kNN(q), v \in \mathbb{U} - kNN(q), d(q, u) \leq d(q, v)$.

To solve both kind of queries and to avoid as many distance computations as possible, many indexing approaches have been proposed. In this paper we have focused on the *List of Clusters* (*LC*) [3] and *SSS-Index* [4] indexes 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. We are not affirming that these indexes are the only suitable ones for the GPU, but their properties make them good candidates to be it. Besides, finding the best metric index for GPU is not the target of this paper; we mainly want to show the great performance achievable using a metric index on GPU compared to sequential and traditional multi-core approaches.

In [7] the authors propose a range query solution on a GPU platform, using the *Spaghettis* metric index [8]. This index is based on pivots, and they use a database of words. In the present work we also use an index based on pivots called *SSS-Index* [4], and also we propose a solution using the LC index [3] (which is based on clustering), that outperforms the previous indexes based on pivots. Also, it is important to say that although we get best results in real time, we do not use exactly the same databases and graphic cards.

[9] proposes a *heap* based implementation of *LC* and *SSS-Index* to solve kNN queries on GPU. Due to the different nature between kNN and *range* queries we cannot rightly reuse the same implementations for our purposes. In this paper we present *ad-hoc* GPU implementations to solve range queries, and besides we propose and compare different strategies to efficiently exploit a multi-GPU platform.

In the following subsections we explain the construction of the two implemented indexes (*LC* and *SSS-Index*) and describe how range queries are solved using them.

A. List of Clusters (LC)

This index [3] 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 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 query q with radius r is solved by scanning the centers in order of creation. For each center c, the distance $d(q, c)$ is always 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.

B. Sparse Spatial Selection (SSS-Index)

During construction, this pivot-based index [4] 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 uses the table to discard objects instead of 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.

III. 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 ([10]). 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.

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

IV. Range Queries

In this section we describe the mapping of three range search 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] [12] we assume a high frequency of incoming queries and exploit coarse-grained inter-query parallelism, i.e. we always solve nq queries in parallel. Moreover, 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, depending on 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 range search algorithms.

We introduced a brute force algorithm which is used as point of comparison with the indexed methods.

A. Brute Force Algorithm

The overall idea is that each CUDA Block processes a different query and within a CUDA Block, each thread computes the distance between the query and a subset of the elements of the database. The database is a $D \times E$ matrix, where D is the dimension of its elements and E is the size of the database, which has been uploaded previously to *device memory*. Queries are also uploaded into device memory but the threads of each CUDA Block cooperate to transfer their associated query to the *shared memory* to accelerate its access (this is the first step of the algorithm). Afterwards, threads compute the distance between the query and the elements of the database following a Round-Robin distribution. Most work is performed within the device distance function. Database elements are stored column-wise to increase the chances of coalesce memory accesses when computing these distances since that way consecutive threads have to access adjacent memory locations.

Depending on the application, when an element is found as a result, different information may be retrieved. For the purposes of this work, a counter is increased to track the number of elements found by each thread, but no further processing or transfers are performed.

B. List of Clusters (LC)

The data structure that holds the LC index consists of 3 matrices denoted as *CENTER*, *RC* and *CLUSTERS* in algorithm 1. *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

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

each column represents the center of a cluster, *RC* is an array that stores the covering radius of each cluster, and *CLUSTERS* is a $D \times N_{clu}$ matrix (N_{clu} is the number of elements in all the clusters) that holds the elements of each cluster. Index information is stored column-wise to favor coalesce memory accesses as in the Brute Force Technique.

Algorithm 1 shows the pseudocode of the main CUDA kernel that solves a range query (*query*, *range*) using the *LC* index. Each CUDA Block processes a different query, which is transferred from device memory to shared memory (line 6) since it is accessed by all its threads when performing distance evaluations. Once the query has been saved into the shared memory, the *for* loop starting at line 11 iterates over the different clusters. Each thread computes the distance between q and a subset of elements of *CENTER* following a Round-Robin distribution. Most work is performed again within the device function `distance()`. If distances are lower than $range$, the respective centers cluster are appended to the list of results in `found()` (line 14). Clusters are marked for exhaustive search at line 16 only if their respective center balls have some intersection with the query ball. A property of this index (given by its construction) is that the exhaustive search over a cluster can be pruned if the query ball is totally contained in a given center ball. If this is the case (line 17), then we do not consider the subsequent clusters and delimit the number of clusters at line 18.

Finally, the *for* loop starting at line 24 processes all the elements of the selected clusters as in the Brute Force technique.

C. SSS-Index

The *SSS-Index* consists of 3 matrices denoted as *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 element is the distance between a pivot and an element of the database, and *DB* is a $D \times N_{DB}$ matrix where each column represents an element of the database. As in the *LC*, the index information is stored column-wise to favor coalesce memory accesses.

As in the *LC*, each CUDA Block transfers its associated query to shared memory due to its frequent access. Once a synchronization ensures the query has been copied before being accessed, each thread performs the distance evaluations between the query and a subset of pivots following a Round-Robin distribution. And finally, the rows of *DISTANCES* are distributed across threads to test if their respective elements of the database can be discarded. For every non discarded element, a distance evaluation is performed.

In [4], 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 2, 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 coalesce them. This leads to the observed increase in the number of Read/Write operations. In summary, 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.

Algorithm 1 *LC* search algorithm CUDA kernel.

{Each row of *Queries* is a query.}
{D is the dimension of the elements in the database and *bsize* is the number of elements of each cluster.}

__global__ range_SearchLC(float **CENTERS*, float ***RC*, float **CLUSTERS*, float **Queries*, float *range*)

```
 1: __shared__ float query[D]
 2: __shared__ int exhaustive[N_cen]
 3: __shared__ int minC=N_cen
 4:
 5: for (i=threadIdx.x; i<D; i+=blockDim.x) do
 6:     query[i] = Queries[blockIdx.x][i]
 7: end for
 8:
 9: __syncthreads()
10:
11: for (i=threadIdx.x; i<minC; i+=blockDim.x) do
12:     dist = distance(CENTERS, i, query)
13:     if dist <= range then
14:         found()
15:     end if
16:     exhaustive[i] = dist <= RC[i] + range
17:     if dist < RC[i] - range then
18:         atomicMin(&minC, i)
19:     end if
20: end for
21:
22: __syncthreads()
23:
24: for (i=threadIdx.x; i<N_clu && i/bsize<=minC; i+=blockDim.x) do
25:     if exhaustive[i/bsize] == 1 then
26:         if distance(CLUSTERS, i, query) <= range then
27:             found()
28:         end if
29:     end if
30: end for
```

V. OpenMP and Sequential Implementations

We have compared our GPU-based implementations against sequential and OpenMP-based counterparts. As a basis we have taken the proposals for implementation on multi-core systems introduced on [12]. In the so-called *Local* method, queries are distributed across threads following a round-robin distribution. This is the best alternative for high query traffic since its synchronization overheads are very low. Authors have also investigated additional methods for exploiting fined-grained parallelism when query traffic is low at the expense of higher synchronization cost. In those cases, queries are further split into a number of tasks, which are distributed across cores (instead of distributing queries). In both cases the index is constructed prior to the searching process, and all threads have

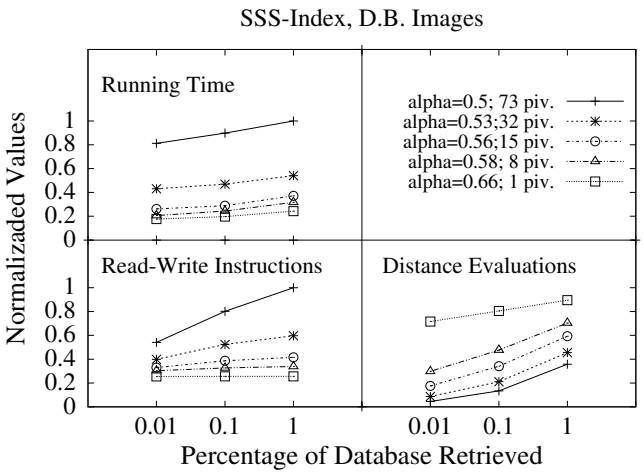


Fig. 2. Range search algorithm using *SSS-Index* with different number of pivots.

TABLE I
MAIN FEATURES OF THE COMPUTING PLATFORM FOR SEQUENTIAL AND OPENMP IMPLEMENTATIONS.

Processor	2xIntel Quad-Xeon (2.66 GHz)
L1 Cache	4x32KB + 4x32KB (inst.+data) 8-way associative, 64byte per line
L2 Unified Cache	2x4MB (4MB shared per 2 procs) 16-way associative, 64 byte per line
Memory	16GBytes, (4x4GB) 667MHz DIMM memory, 1333 MHz system bus
Operating System	GNU Debian System Linux kernel 2.6.26-SMP for 64 bits
Intel C/C++ Compiler v11.0 (icc)	-O3 -march=pentium4 -xW -ip -ipo Parallelization with OpenMP: -openmp

access to it (it is allocated on main memory). The number of threads is always equal to the number of cores, and each thread is mapped onto a different core.

For the purpose of this paper we have focused on algorithms for high query traffic, and therefore we have chosen the Local method as a reference for our implementations. The sequential implementation is based also on this code, but removing OpenMP primitives and pragmas (essentially the same implementation but just with one thread).

VI. SINGLE GPU EVALUATION

All our GPU experiments were carried out on a NVIDIA Tesla C1060 which is shipped with 30 multiprocessors, 8 cores per multiprocessor, 16KB of shared memory and 4GB of device memory. The host CPU is an Intel's Clovertown processor with 16 GB of RAM. Table I details the experimental environment used with the OpenMP implementations.

We have used two different reference databases:

Spanish **:** a spanish dictionary with 51,589 words. We used the *edit distance* [13] to measure similarity. 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 120,000 random image objects. We built each index with the 80% of the objects and the remaining 20% objects were used as queries (23,831 queries). In this collection we used the *euclidean distance* to measure the similarity between two objects. Intrinsic dimensionality of this space is higher than the dimentionality of the previous database, but it is still considered low.

In the vector database (*Images*) the radii used were those that retrieve on average the 0.01%, 0.1% and 1% of the elements of the database per query. In the *Spanish* database the radii were 1, 2 and 3. Similar values have been also used in previous papers [12], [14]. In all the proposed methods, the set of queries are previously copied to device memory.

Regarding the GPU implementation, we performed a wide exploration to obtain the best parameters for each indexed structure. Regarding *LC* we found that 64 elements per cluster is the best option for the vector database, while 32 performs the best in the *Spanish* database. We already discussed *SSS-Index* tuning in Section IV-C. The conclusions there drawn hold for the *Images* database, so a single pivot ($\alpha = 0.66$) is used. However, for the *Spanish* database it is better to use 68 pivots ($\alpha = 0.5$).

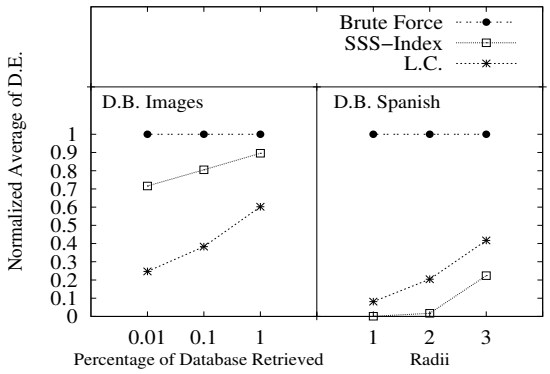
Figure 3 illustrates the performance characteristics of our GPU implementations. *Brute Force* stands for the exhaustive-search algorithm. *LC* and *SSS-Index* show the results for the two implemented indexing mechanisms with the parameters indicated above. All figures are normalized to the largest value of each version.

We first place our attention on the total number of distance evaluations (Figure 3(a)). Both database behaves as expected: indexing mechanisms do significantly decrease the number of distance evaluations when compared to the brute force search method. *SSS-Index* outperforms LC in the number of distance evaluations performed with the *Spanish* database. However, since we just used one pivot for the *Images* to maximize performance, its filtering efficiency is hugely penalized.

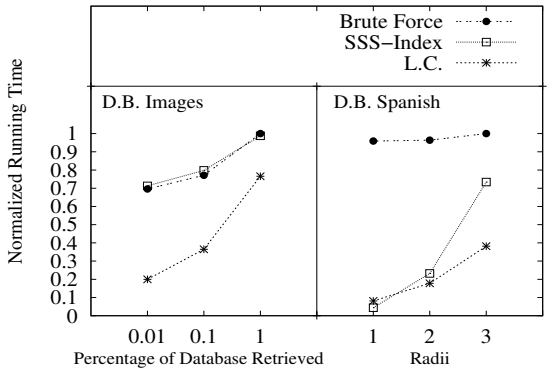
One would expect that running times mimic the trend shown by the distance evaluations but results in Figure 3(b) partially contradicts this intuition: the brute force search algorithm behaves better than expected in the *Images* database. It equals and even improves *SSS-Index* performance. With *Spanish* database changes are not so drastic, but *LC* becomes the best approach even if it performs more distance evaluations.

Figure 3(c) gives the clue: the regularity of the brute force algorithm leads to a more GPU aware access pattern. Coalescing and alignment of memory access heavily influences performance on current GPUs. As stated in Section III, when a warp launches misaligned or non-consecutive memory accesses, hardware is not able to coalesce it and a single reference may become several separate accesses. *LC* code is also quite regular and, moreover, it performs much less distance evaluations thus reducing the number of memory accesses. This explain the sustained superior performance of *LC* over the other implementations

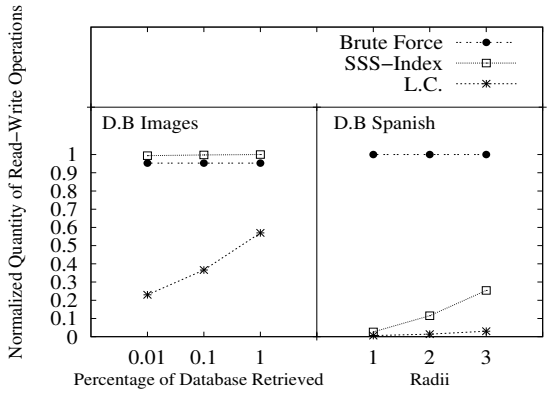
Focusing on overall performance, Figure 4 shows the speed-ups of all our implementations taking as reference the perfor-



(a) Number of Distance Evaluations



(b) Running time



(c) Number of read/write operations

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*.

mance of a sequential brute force version executed in a single CPU. We show results for the *Spanish* database because it can discuss different scenarios not present in the *Images* database. Table I describes the CPU nodes used for the sequential and OpenMP experiments.

The first two columns of Figure 4 are the speed-ups of sequential implementations of the two indexes,*LC* and *SSS-Index* . The next two columns are the speed-ups considering the OpenMP implementation run on 8 core server. Finally, the last two columns of this figure are the speed-ups of our single GPU implementations described in Section IV. Please note that we tune each implementation to attain the maximum performance, so index parameters may vary across implementations. We run the search with three different ranges. Given the large speed-up variations, we separate the results in two figures.

Overall, GPU approaches largely outperforms OpenMP approaches which was expected given the superior capabilities of the GPU. One common and expected trend is that, the smaller the radius employed, the larger the performance benefits of indexing (sequential or parallel) techniques. Remind that the reference case is the exhaustive search (i.e. all the possible distance evaluations are performed). If we increase the range of the search, the number of distance evaluations gets increased for the indexed implementations, thus narrowing the gap with brute-force implementation.

OpenMP implementations scale worse than GPU versions when compared with their respective sequential implementations, getting speed-ups of 4.4x, 4.0x and 3.8x for *LC* and 3.4x, 3.6x and 3.9x for *SSS-Index*. The common memory controller is a bottleneck for the multi-core sever, since accesses from the 8 cores are issued concurrently. Conflicting accesses are then serialized, thus decreasing potential performance gains. Current graphic processor units overcome this limitations offering an enormous bandwidth between processing elements and the DDR memory. Access coalescing plays a crucial role in the right exploitation of this feature. Moreover, fine grained multithreading helps to partially hide the unavoidable and long memory latencies.

For some readers, GPU speed-up factors may not look so impressive taken into account that our GPU has 30 *multiprocessors* on-board, compared with the 8-core Xeon based server used for OpenMP experiments. However, it is important to remind that each of this NVIDIA *multiprocessor* is extremely simpler than the Core/Nehalem michroarchitecture based Intel CPUs; instruction level parallelism is almost not exploited while it represents the main source of performance for complex out-of-order processors.

When we look the details of the index methods, it is relevant to note that, for the smallest *radii SSS-Index* runs faster than *LC* in all platforms. This just confirms the results shown in Figure 3(b).

When *radii* becomes 2, the GPU trend is reversed (again announced in Figure 3(b)). However, both the sequential and the OpenMP implementations of *SSS-Index* still outperforms their *LC* counterparts. It sounds reasonable since, as shown in Figure 3(a), *SSS-Index* performs much less distance evaluations than *LC*. Regarding the biggest *radii* the previous trend is confirmed and *LC* now always outperforms *SSS-Index*: independently of the platform, the huge irregularity inherent to *SSS-Index* does counteract the benefits of the saved distance evaluations and overall performance degrades very fast with large *radii*.

Regarding *Images* database, the exhibited behavior match that of the largest *radii*: *LC* consistently outperforms *SSS-Index* for any search range. Motivated by this finding, we decide to adapt our GPU *LC* implementation to a multi-GPU environment. Implementation details and results are shown in

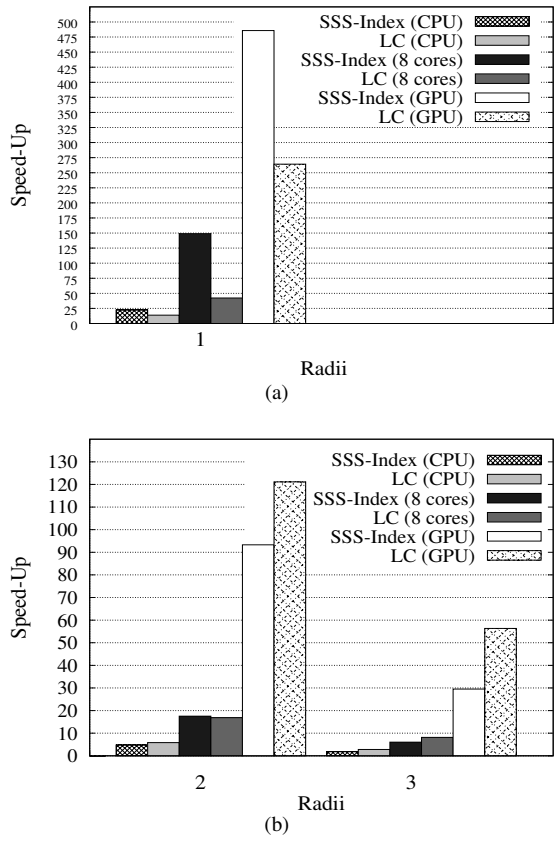


Fig. 4. Speed-ups of the *SSS-Index* and *LC* using different platforms (on *Spanish* database), over sequential brute force algorithm, with radius **(a)** 1, **(b)** 2 and 3.

the next section.

VII. Multi-GPU Strategies

In this section we propose and compare strategies using the LC (because its good results on Section VI) on a multi-GPU platform. [11] and [2] have shown several strategies to distribute the LC on a cluster of processors connected by an Infiniband 1000 MB/s network. Due to the differences between this platform and the multi-GPU server, the same results cannot be applied in this case. But, the work shown on these publications was taken as basis for this section.

As we are able to manage all the GPUs memories, which allows us to store a bigger database. To exploit this property, we just take account strategies that create a global index distributed among the device memories of the GPUs. Below we present and compare two different strategies, the first (*2-Stages Strategy*) process a percentage of the searching on CPU, whereas the second (*1-Stage Strategy*) process all the searching on GPU.

A. 2-Stages Strategy

The main idea is to divide the searching process of a query batch in two stages. The first stage establish which GPU must process which cluster, and the second calculates the distances between the cluster and the queries.

The clusters are distributed in a circular manner, and all the GPUs have a map that indicates the ID of the GPU where each cluster is stored. In order that any GPU could decide which clusters must be compared with a particular query, all the GPUs have a copy of all the centers and its covering radius. Because the latter, is possible to distribute the queries among the GPUs, each query to a single GPU.

In the first stage, the centers are distributed in a circular manner among the threads, to get the discarded clusters and the clusters that must be compared with the queries. This would correspond to the process made by lines 11-20 of the (1-GPU) Algorithm 1. After this, each GPU indicates (in a matrix) to the rest of them what of their clusters must be compared with what of the queries. This information must be stored in device memory because it is an input parameter for the next stage.

In the second stage, each GPU launches P kernels (P: number of GPUs). The kernel i performs the distance evaluations between clusters and queries established by the i-th GPU. After that, is required to read from device memory the input parameter, that indicates which cluster must be accessed. This would correspond to the lines 24-30 of the (1-GPU) Algorithm 1.

B. 1-Stage Strategy

The aim is to solve each query in just one step, avoiding making a scheduling for each query. The centers and their respective clusters are distributed among the GPUs, and because of this, each query must be processed by all the GPUs.

Each GPU perform the distance evaluations between centers and queries. By each non-discarded cluster, their elements are compared against the query in the same step.

One advantage of this strategy is that it avoids readings from device memory, because the clusters that must be accessed are known in the same step. But a disadvantage, is that it is not as efficient as the 2-Stage strategy which stops a searching when a query is contained in a cluster (line 17 in Algorithm 1).

C. Experimental Results on multi-GPU Strategies

In the experiments we used a multi-GPU server with 4 GPUs. Each one of these, is a NVIDIA Tesla C1060 with the same features of the GPU used in Section VI. In this section we used bigger databases than Section VI to try to exploit the memory capacity of the platform. They are described below.

***Words Database*:** This database is a merge of dictionaries from United Kingdom. From this database we chose equidistant words to create new databases of 100000, 200000, 500000 and 1 million of elements. The query file of 40000 elements was the same as that used in *Spanish* database (Section VI).

***Vectors Database*:** As we did in *Images* database, we took a NASA database containing 40700 images vectors, and we used them as an empirical probability distribution to generate databases of size 100000, 200000, 500000 and 1 million. The query file contains 23831 elements.

2-Stage strategy is designed to better balance the work amongst the available nodes. Even if it succeeds, it introduces extra CPU-to-GPU communication that spoils the expected

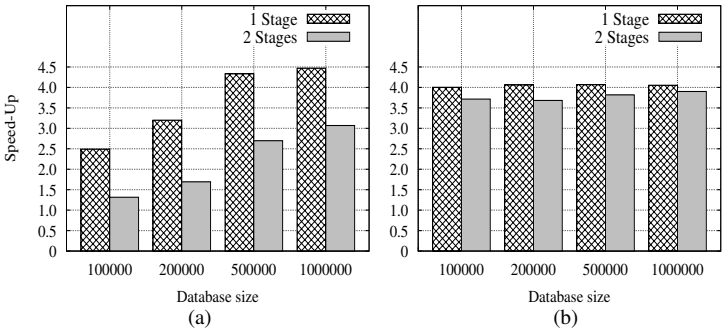


Fig. 5. Speed-up of the multi-GPU Strategies on the **a)** vectors database, and **b)** words database

TABLE II
REAL TIME IN SECONDS FOR THE LC ON VECTORS DATABASE (RECOVERING 1% OF THE DB) USING THE DIFFERENT PLATFORMS. THE QUERY FILE WAS A LOG WITH 23831 QUERIES.

LC	multi-core (8 cores)	1 GPU	4 GPUs (1-Stage strategy)
100000 elems.	14.6	1.6	0.64
200000 elems.	42.6	3.24	1.01
500000 elems.	109.7	9.16	2.11
1000000 elems.	224.5	17.74	3.97

benefits. Moreover, due to the nature of the parallelization followed in the *1-Stage* strategy, there is no inter-GPU communication required and the only relevant communication penalty is the potential unneeded copies of queries to certain GPU nodes. This explains the sustained better performance of *1-Stage* strategy over the *2-Stage* one. However, we can observe in the *words* database (see 5(a)), performance differences are significantly lower than that of the *vectors* database case. As we showed in Section VI, LC makes much more reduction of the number of distance evaluations in the *words* database; the *2-Stage* strategy benefits more from this extra filtering since it better balances the work load.

It is also worth noting that, as Figure 5, we observe a super-lineal speedup (up to x4.5 speedup with just 4 GPUs) for the *1-Stage strategy* as the size of the problem increases. This can be explained through the *occupancy* of each version. As stated in line 2 of Algorithm 1, the amount of shared memory required by our implementation is proportional to the size of the index stored in the node. For the single GPU implementation, the whole index (i.e. the centers of all the clusters) are stored in the node, limiting the potential *occupancy* of the node up to 50%. Since the index itself is distributed across nodes, using 4 GPUs results in 100% *occupancy* and thus better exploits the available resources by launching more thread blocks concurrently.

Finally, Table II shows the real execution times of all our implementations for the LC strategy over the *vectors* database.

VIII. CONCLUSIONS

In this paper we have presented efficient implementations of suitable indexing mechanisms which are mapped on CUDA based GPUs. In the experiments, they outperform both optimized OpenMP and sequential implementations.

We found that the optimal parameters in the context of the GPU, for both *List of Clusters* and *SSS-Index*, are extremely different than those found on the sequential and OpenMP implementations. In particular, the best GPU implementation found for *SSS-Index* with vectors databases uses a single pivot to prune the search space, which shows that the SSS algorithm is inefficient under this context, since this pivot is selected at random among the database objects.

The *List of Cluster* is the index with best performance on GPU, achieving a speed-up of 264x over the sequential brute force algorithm with the words database, and 138x with the vector database.

Finally, we compared and proposed different strategies for the *LC* using a multi-GPU platform, exposing the difficulties of dealing with this kind of platform. We obtained a super-linear speed-up over the single GPU version.

REFERENCES

[1] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín, "Searching in metric spaces," in *ACM Computing Surveys*, September 2001, pp. 33(3):273–321.

[2] M. Marin, F. Ferrarotti, and V. Gil-Costa, "Distributing a metric-space search index onto processors," in *39th International Conference on Parallel Processing*, ser. ICPP 2010. San Diego, USA: IEEE Computer Society, September 2010, pp. 433–442.

[3] E. Chávez and G. Navarro, "A compact space decomposition for effective metric indexing," *Pattern Recognition Letters*, vol. 26, no. 9, pp. 1363–1376, 2005.

[4] N. R. Brisaboa, A. Fariña, O. Pedreira, and N. Reyes, "Similarity search using sparse pivots for efficient multimedia information retrieval," in *ISM*, 2006, pp. 881–888.

[5] P. Zezula, G. Amato, V. Dohnal, and M. Batko, *Similarity Search: The Metric Space Approach*, ser. Advances in Database Systems. Springer, 2006, vol. 32.

[6] T. Cover and P. Hart, "Nearest neighbor pattern classification," *Information Theory, IEEE Transactions on*, vol. 13, no. 1, pp. 21–27, 1967.

[7] R. U. Paredes, P. Valero-Lara, E. Arias, J. L. Sánchez, and D. Cazorla, "A gpu-based implementation for range queries on spaghettis data structure," in *Computational Science and Its Applications (ICCSA 2011)*, ser. Lecture Notes in Computer Science, vol. 6782. Santander, Spain: Springer, June 2011, pp. 615–629.

[8] E. Chávez, J. L. Marroquín, and R. Baeza-Yates, "Spaghettis: An array based algorithm for similarity queries in metric spaces," in *6th International Symposium on String Processing and Information Retrieval (SPIRE 1999)*, Los Alamitos, USA, 1999, pp. 38–46.

[9] R. Barrientos, J. Gómez, C. Tenllado, M. Prieto, and M. Marin, "knn query processing in metric spaces using gpus," in *17th International European Conference on Parallel and Distributed Computing (Euro-Par 2011)*, ser. Lecture Notes in Computer Science, vol. 6852. Bordeaux, France: Springer, September 2011, pp. 380–392.

[10] "CUDA: Compute Unified Device Architecture. ©2007 NVIDIA Corporation." [Online]. Available: http://developer.nvidia.com/object/cuda.html

[11] V. Gil-Costa, M. Marin, and N. Reyes, "Parallel query processing on distributed clustering indexes," *Journal of Discrete Algorithms*, vol. 7, no. 1, pp. 3–17, 2009.

[12] V. G. Costa, R. J. Barrientos, M. Marín, and C. Bonacic, "Scheduling metric-space queries processing on multi-core processors," in *PDP*, M. Danelutto, J. Bourgeois, and T. Gross, Eds. IEEE Computer Society, 2010, pp. 187–194.

[13] V. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet Physics Doklady*, vol. 10, no. 8, 1966, pp. 707–710.

[14] G. Navarro and R. Uribe-Paredes, "Fully dynamic metric access methods based on hyperplane partitioning," *Information Systems*, vol. 36, no. 4, pp. 734 – 747, 2011.