Query Processing in Metric Spaces using GPUs

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

Abstract— **Similarity search has been a problem 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 usually indexed off-line to enable acceleration of on-line searches. Even with indexed databases, it is essential to parallelize the on-line query solving process. In the past, many strategies have been proposed to parallelize this problem in distributed and shared memory multicore systems. Lately, GPUs have also been used to implement brute-force approaches instead of using indexing structures. In this work we propose a GPU based metric-space index data structure for similarity search that outperforms previous OpenMP and GPU brute-force based implementations. We also validate our implementation in the context of real-time systems, when it is not affordable to wait for thousands of queries to fill the system before processing them all in parallel.**

Keywords— **Range Queries, Metric Spaces, Metric Databases, Similarity Search, GPU.**

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. Efficient similarity search is useful in multimedia information retrieval, data mining or pattern recognition problems. Range search also enables other relevant operations such as nearest neighbors search. 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. There exists a distance function that operates on those vectors to determine how similar are objects to a given query object. The distance between any given pair of objects 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 range search queries using GPUs. We have found that obtaining efficient performance from this hardware, in this application domain, can be particularly difficult since many of the metric-space solutions developed for traditional shared memory multiprocessors and distributed systems cannot be implemented efficiently on GPUs.

Our focus is on search systems devised to solve large streams of queries. Previous related work has shown that conventional parallel implementations for clusters and multicore 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. In contrast, on GPUs it is necessary to exploit both coarse and fine grained parallelism where the cost of data transfers such as pieces of index can hide the benefits of keeping smartly indexed the database.

We studied a number of alternative sequential metric-space index data structures and realized that two candidates, namely *LC* and *SSS-Index* (details below), were best suited for GPUs as their data organization and operations resemble computations on two-dimensional matrices.

Interestingly enough, the LC and SSS indexes have been shown to achieve efficient performance in shared memory multi-core and distributed memory cluster processors by previous work. This allowed us to expose a comparative study of our proposal against optimized implementations of the same indexes both sequentially and in parallel for shared memory using OpenMP [1].

II. 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* [2] and *The k nearest neighbors (kNN)* [3], [4]. 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* [6] strategies since (i) they are two of the most popular non-tree structures that are able to prune the search space efficiently and (ii) they hold their indexes on dense matrices which are very convenient for mapping algorithms onto GPUs [7].

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

A. 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

[1]ArTeCS Group, Complutense University, e-mail: ribarrie@fdi.ucm.es

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.

B. Sparse Spatial Selection (SSS-Index)

During construction, this pivot-based index [6] 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 $(\mathbb{X},)$ 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 ([7]). 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.

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 [8][9] we assume a high frequency of incoming queries and exploit coarse-grained inter-query parallelism, i.e. we always solve nq queries in parallel. 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

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

threads (from 128 to 512, 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 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, which is the first step. 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 function that performs the distance between elements and the query. 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.

B. List of Clusters (LC)

The data structure that holds the LC index consists of 3 matrices 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 $D \times N_{clu}$ 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.

Each CUDA Block processes a different query, which is transferred from device memory to shared memory since it is accessed by all its threads when performing distance evaluations. Once the query has been saved into the shared memory, a *for* loop 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 that performs the distance between elements and the query. If distances are lower than *range*, the respective centers cluster are appended to the list of results. Clusters are marked for exhaustive search 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, then we do not consider the subsequent clusters delimiting the number of clusters.

Finally, a *for* loop processes all the elements of the selected clusters as in the Brute Force technique.

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

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 a 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 the synchronization ensures the query has been copied before access it, 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, that test if their respective elements of the database can be discarded. For every non discarded element, a distance evaluation is performed.

In [6], authors have found empirically that around $\alpha = 0.4$ 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, the best performance is obtained with just one pivot for vector databases. 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, and this increases 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.

V. Experimental Results

All our GPU experiments were carried out on a NVIDIA GeForce GTX 280 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, composed by 2xIntel Quad-Xeon (2.66 GHz), and in each core 32KB of L1 Cache for instructions and 32KB for datas, each two cores shares the L2 Cache of 4MB, and the RAM is of 16 GB.

We have used two different reference databases:

Spanish **:** A Spanish dictionary with 51,589 words and we used the *edit distance* [10] 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.

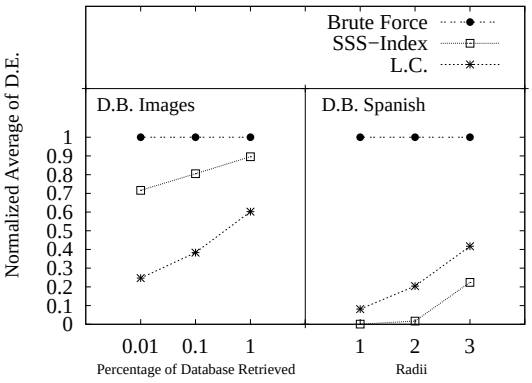
In the vector database (*Images*) the radius used were those that retrieve on average the 0.01%, 0.1% and 1% of the elements of the database per query. In the *Spanish* database the radius were 1, 2 and 3. Similar values have been also used in previous papers [9][8]. In all the proposed methods, the 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 64 pivots ($\alpha = 0.5$).

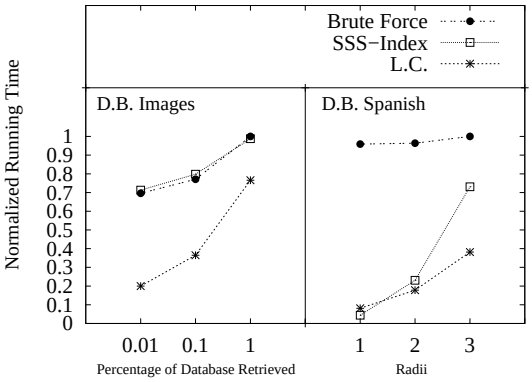
Figure 1 illustrates the performance characteristics of our GPU implementations. *Brute Force* stands for the exhaustive-search algorithm. LC and *SSS-Index* show the results for the two implemented indexing mechanisms with the parameters indicated above. All figures are normalized to the largest value of each version.

We first place our attention on the total number of distance evaluations (Figure 1(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* typically outperforms LC when checking distance evaluations. And it does it if we consider the *Spanish* database. However, since we just used one pivot for the *Images*, LC becomes the *winner* in this category. One would expect that running times mimic the trend exhibited by the distance evaluations but results in Figure 1(b) partially contradicts this intuition: the brute force search algorithm behaves better than expected in the *Images* database. It equals or even improves *SSS-Index* performance. For the *Spanish* database changes are not so drastic, but LC becomes the best implementation even if it performs more distance evaluations.

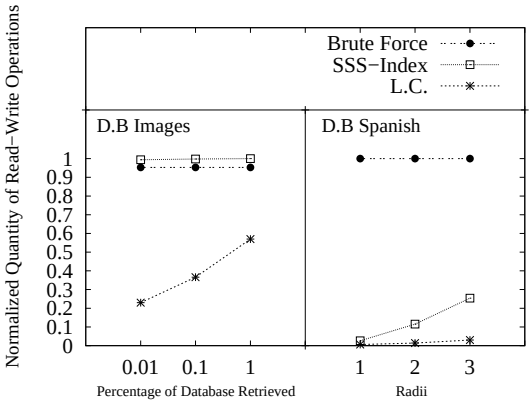
Figure 1(c) has the clue: the brute force technique



(a) Number of Distance Evaluations



(b) Running time



(c) Number of read/write operations

Fig. 1. Normalized **a)** Distance evaluations per query (average) **b)** Running time and **c)** Read-write Operations (of 32, 64 o 128 bytes) to *device memory*.

has a slightly better memory access pattern over *SSS-Index* when dealing with the *Images* database. The 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 up to 32 separate accesses. The LC shows the best results on this aspect on both databases, which explains its superior performance previously reported.

A. Performance of parallel implementations

Figure 2 shows the performance speed-ups of our GPU indexed implementations over optimized sequential implementations. Results are very impressive for *SSS-Index* (up to 248x for 1% elements re-

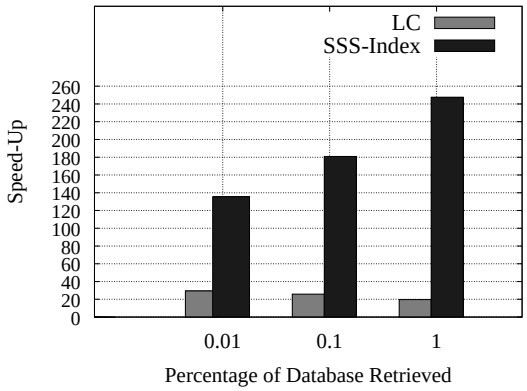


Fig. 2. Speed-Up of GPU versions of *LC* and *SSS-Index* over sequential counterparts with DB *Images*.

trieved) due to the poor CPU performance of this indexing mechanisms. But even for the lighter[3] index (*List of Clusters*) our implementation achieves a 29x speedup, which seems hard to attain with a medium-size multicore server.

To prove that last point, we run the same experiments on the multicore system using the OpenMP implementation presented in [9]. The OpenMP parallel version scales worse than expected with the number of cores: our experimental framework consists of 8 cores but the parallel implementation is just 4 times faster than the sequential counterpart, as shown in figures 2 and 3.

Given the synchonization-free parallelism exploited, we expected almost linear scaling with the number of cores. Indeed, with no aggressive compilation flags, the parallel version scales linearly. Table I shows this behavior for the *List of Clusters* implementation when retrieving 1% of the database for each query: to increase the optimization level leads to faster implementations but worsens the speedup metric when compared with the sequential implementation. All the others experiments followed the same trend.

After aggressive compiler optimizations, the memory system becomes even more critical since *density* of accesses increases. Even if no inter-thread communication is present in our implementation, certain levels of the memory system are shared. The effect of L2 sharing can be estimated when launching 4 threads instead of 8. Default thread-to-core assignment yields a x2.03 speedup factor over the sequential implementation. Making the assignment more *L2-cache friendly* increases the speedup factor up to x2.5. The common memory controller is another bottleneck, since accesses from the 8 cores are issued concurrently. Conflicting accesses are then serialized, thus decreasing potential performance gains. This resource sharing explains the sub-linear speedup factor obtained.

Figure 3 shows the speedups obtained by the GPU implementation over the 8-core OpenMP version. As expected from the above discussion, it mimics the

[3]The required space to store the *LC* is equal to the 6% of the space required by the *SSS-Index*.

LC	No optimization flags	With optimization flags
Sequential Time	89.46s	21.94s
8 cores	11.2 s	4.95s
speedup	7.98	4.43

TABLE I

Execution times of sequential and parallel version (OpenMP based, 8 cores) for *List of Clusters*.

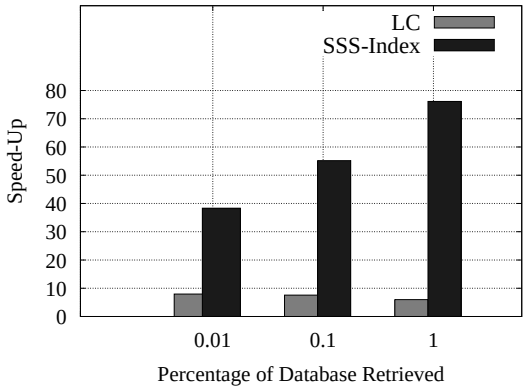


Fig. 3. Speedup of GPU versions of *LC* and *SSS-Index* over corresponding OpenMP implementations with DB *Images*.

trends of figure 2. *SSS-Index* benefits much more from the two level of parallelism exploited (both inter- and intra- query parallelism). We achieved a maximum speedup of x76.104 when searching with a radius $r = 0.73$ (this radius implies to retrieve a 1% of the elements of the reference database). *LC* performance gains are more modest, but still relevant: from 5.96x to 7.92x depending on the selected *radius*. This speedup factor may not look 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.

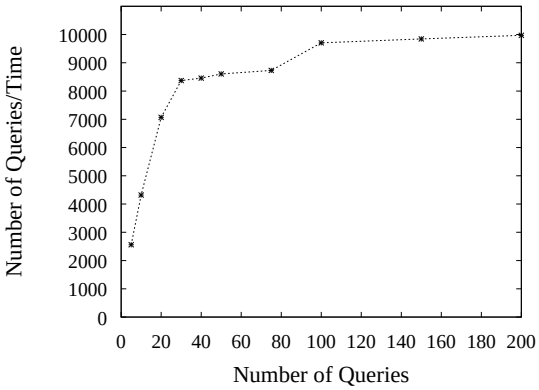
B. *Solving queries* on-line

All results reported so far assume that we know all the queries to be solved in advanced. For the *Image* database, that means that we assume a total of 23831 queries in the system before we start solving them all in parallel. While this assumption could be admissible for certain use cases, it could be unaffordable in on-line *real-time* systems like web searching for multimedia contents [11].

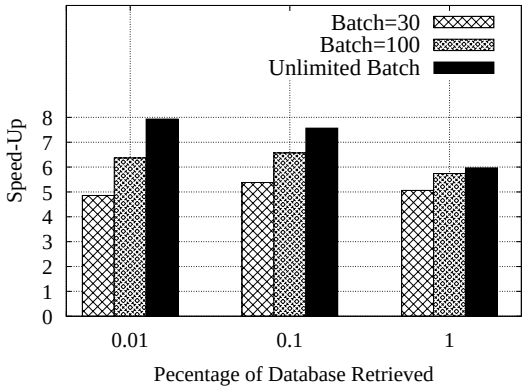
We performed a productivity test in function of the number of queries issued in parallel. Figure 4(a) shows the results for the *List of Cluster* implementation. The x-axis indicates how many queries are launched in parallel (starting in five queries at a time). The y-axis shows the productivity of the system measured in the number of queries processed by second. Not surprisingly, the productivity rapidly

increases up to the point where we launch 30 queries in parallel (remind that the GPU used in the experiments includes 30 *multiprocessors*). Below that point the GPU is underused and the constant penalty of launching a kernel weights too much. There is a knee at 30 but productivity still slowly increases due to the GPU multithreading capabilities which allows to hide long memory latencies effectively. Launching queries in batch of 200 is almost at the maximum productivity achievable with our implementation (anyhow, it is always advisable to launch as many queries in parallel as possible since it reduces the number of *kernel* invocations).

Figure 4(b) reviews the speed-up figures for the *LC* implementation. The bar labeled *Unlimited Batch* corresponds to the *LC* speed-up bar in figure 3. It is the upper bound for GPU performance, since all queries are solved with a single *kernel* invocation. Bars labeled *Batch=30* and *Batch=100* corresponds with a scenario where, as soon as we have 30 (resp. 100) queries in the system, we launch a kernel to solve them. Even if the productivity is not at its highest point, the GPU implementation always outperforms the OpenMP version. Please note that OpenMP based implementations are still assuming their *best-case*, i.e. all queries are known from the beginning, so no dispatching overhead is present. A speedup higher than 5x (in average) is achieved when launching just 30 queries in parallel, which represents a very low frequency traffic scenario. Thus, we can conclude that GPUs can be used for *on-line* query processing in metric spaces as a low-cost high performance alternative to traditional multi CPU implementations.



(a) Productivity solving different number of queries



(b) Speed-Up over corresponding OpenMP implementation using different size of batch

Fig. 4. **a)** Productivity (number of queries divided by its corresponding running time) solving different number of queries of the *LC* over *Images* database. **b)** Speed-Up of *LC* solving a batch of queries (of 30 and 100) at a time over corresponding OpenMP implementation with DB *Images*.

VI. Conclusions

In this paper we have presented efficient implementations of suitable indexing mechanisms which are mapped on CUDA based GPUs. We compared them against optimized OpenMP and sequential implementations, overcoming both of them.

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* uses a single pivot to prune the search space, which shows that the SSS algorithm is inefficient 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 29x over the sequential counterpart, and 7.9x over an optimized 8-thread OpenMP implementation.

In the context of the processing of stream of queries, based on the productivity of our algorithms, we found that solving batches of queries whose size equals the number of GPU multi-processors is a strategy able to achieve good speed-up.

References

[1] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas, *Using OpenMP: portable shared memory parallel programming*, The MIT Press, 2008.

[2] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José L. Marroquín, "Searching in metric spaces," in *ACM Computing Surveys*, September 2001, pp. 33(3):273–321.

[3] D. W. Aha and D. Kibler, "Instance-based learning algorithms," in *Machine Learning*, 1991, pp. 37–66.

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

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

[6] Nieves R. Brisaboa, Antonio Fariña, Oscar Pedreira, and Nora Reyes, "Similarity search using sparse pivots for efficient multimedia information retrieval," in *ISM*, 2006, pp. 881–888.

[7] "CUDA: Compute Unified Device Architecture. ©2007 NVIDIA Corporation.," .

[8] R. Uribe and G. Navarro, "Egnat: A fully dynamic metric access method for secondary memory," in *Proc. 2nd International Workshop on Similarity Search and Applications (SISAP)*. 2009, pp. 57–64, IEEE CS Press.

[9] Veronica Gil Costa, Ricardo J. Barrientos, Mauricio Marín, and Carolina Bonacic, "Scheduling metric-space queries processing on multi-core processors," in *PDP*, Marco Danelutto, Julien Bourgeois, and Tom Gross, Eds. 2010, pp. 187–194, IEEE Computer Society.

[10] V.I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet Physics Doklady*, 1966, vol. 10, pp. 707–710.

[11] Mauricio Marin, Veronica Gil-Costa, Carolina Bonacic, Ricardo Baeza-Yates, and Isaac D. Scherson, "Sync/async parallel search for the efficient design and construction of web search engines," *Parallel Computing*, vol. 36, no. 4, pp. 153 – 168, 2010.