Multi-level Clustering on Metric Spaces Using a Multi-GPU Platform*

Ricardo J. Barrientos[1], José I. Gómez[1], Christian Tenllado[1], Manuel Prieto Matias[1], and Pavel Zezula[2,**]

[1] Architecture Department of Computers and Automatic, ArTeCS Group, Complutense University of Madrid, Madrid, España
ribarrie@ucm.es
[2] Faculty of Informatics, Masaryk University, Brno, Czech Republic
zezula@fi.muni.cz

Abstract. The field of similarity search on metric spaces has been widely studied in the last years, mainly because it has proven suitable for a number of application domains such as multimedia retrieval and computational biology, just to name a few. To achieve efficient query execution throughput, it is essential to exploit the intrinsic parallelism in respective search algorithms. Many strategies have been proposed in the literature to parallelize these algorithms either on shared or distributed memory multiprocessor systems. More recently, GPUs have been proposed to evaluate similarity queries for small indexes that fit completely in GPU's memory. However, most of the real databases in production are much larger. In this paper, we propose multi-GPU metric space techniques that are capable to perform similarity search in datasets large enough not to fit in memory of GPUs. Specifically, we implemented a hybrid algorithm which makes use of CPU-cores and GPUs in a pipeline. We also present a hierarchical multi-level index named *List of Superclusters* (*LSC*), with suitable properties for memory transfer in a GPU.

Keywords: Similarity Search, Metric Spaces, GPU, Range queries.

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 [6]. 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.

Efficient range searches, which are usually dominated by expensive distance evaluations, are crucial for the success of many applications. In fact, the range

* This research was funded by the Spanish government's research contracts TIN2008-005089, TIN2012-32180 and the Ingenio 2010 Consolider ESP00C-07-20811.
** Supported by the grant GACR P103/12/G084.

F. Wolf, B. Mohr, and D. an Mey (Eds.): Euro-Par 2013, LNCS 8097, pp. 216–228, 2013.
© Springer-Verlag Berlin Heidelberg 2013

search operation can be considered as a basic search kernel since it is a commonly used component of more complex search operations, such as the *nearest neighbors search*. In the current technological context, one of the most promising alternatives for the acceleration of this operation is the exploitation of its intrinsic parallelism on Graphics Processing Units (GPUs). Range searches exhibit different levels of parallelism: we can process in parallel many queries, many distances from a given query or even exploit the parallelism in the distance operation itself. This feature matches well with the architecture of the GPU and Multi-GPU systems. However, these architectures have complex memory hierarchies and it has been empirically shown that their efficient exploitation is one of the key elements for the acceleration of many applications.

Previous related work, which focuses 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 [8]. 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 benefits of keeping the database objects smartly indexed.

In this paper we propose the development of two efficient pipeline strategies for coordinating the CPU and the GPU, which is able to hide most of the CPU-GPU data transfer latency. We also propose a new hierarchical index (LSC), which fits very well into these pipelined strategies: the CPU discards elements at the top level of hierarchy and the GPU completes the work using the low level of the index hierarchy. In addition, we have analyzed the impact of the index structure that we used, and the size and nature of the database.

The remaining of the paper is as follows. Section 2 gives some background on similarity search and metric-space databases, and summarizes some previous related work. In Section 3 we describe our proposals to deal with large databases on a multi-GPU platform. Section 4 present the experimental results of our analysis, and finally Section 5 summarizes the main conclusions of this work.

2 Similarity Search Background and Related Work

Searching similar objects from a database 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 [13]: 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 [6]: 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): The goal is to retrieve the set $kNN(q) \subseteq \mathbb{U}$ such that $|kNN(q)| = k$ and $\forall u \in kNN(q), v \in \mathbb{U} - kNN(q), d(q, u) \leq d(q, v)$.

The solution of range queries are used as basis to solve kNN queries, and because of this, the present paper is focused on solving range queries. To avoid as many distance computations as possible, many indexing approaches have been proposed. We have focused on the *List of Clusters* (LC) [5] index, since (1) it is one of the most popular non-tree structures that are able to prune the search space efficiently and (2) it holds its index on dense matrices which are very convenient data structures for mapping algorithms onto GPUs. We are not affirming that this index is the most suitable for GPU, but its properties make it a good candidates to become it. Besides, finding the best metric index for GPU is not a target of this paper; we mainly want to show the high performance achieved using a metric index on GPU compared to sequential and traditional multi-core approaches.

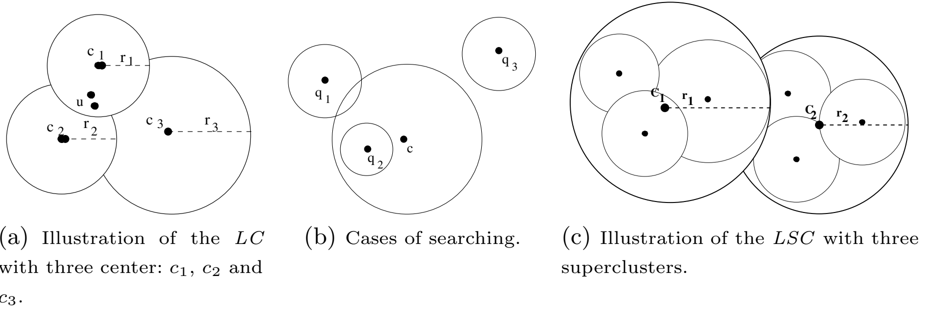
In [11,2] the authors propose solutions for similarity search using a GPU card. All these papers take the initial assumption that the whole index fits on GPU memory, with capacity of a few GiB. In this paper we propose solutions to deal with large databases, which is usually the real case, where the databases fit just partially on the GPU memory.

In the following subsections we explain the construction of the LC index and is described how range queries are solved using it.

2.1 List of Clusters (LC)

This index [4,5] can be implemented dividing the space in two different ways: taking a fixed radius for each partition or using a fixed size. In this paper, to ensure good load balance in a parallel platform, we consider partitions with a fixed size of K elements, thus the radius r_c of a cluster with center c is the maximum distance between c and its K-nearest neighbor.

The LC data structure is formed from a set of centers (objects). The construction procedure (illustrated in Figure 1(a)) is roughly as follows. We (randomly) chose an object $c_1 \in \mathbb{U}$ which becomes the first center. This center determines a cluster (c_1, r_1, I_1) were I_1 is the set $k\text{NN}_{\mathbb{U}}(c_1, K)$ of K-nearest neighbors of c_1 in $\mathbb{U}$ and r_1 is the distance between the center c_1 and its K-nearest neighbor in $\mathbb{U}$ (r_1 is called *covering radius*). Next, we choose a second center c_2 from the set $E_1 = \mathbb{U} - (I_1 \cup \{c_1\})$. This second center C_2 determines a new cluster (c_2, r_2, I_2) where I_2 is the set $k\text{NN}_{E_1}(c_2, K)$ of K-nearest neighbors of c_2 in E_1 and r_2 is the distance between the center C_2 and its K-nearest neighbor in E_1. Let $E_0 = \mathbb{U}$, the process continues in the same way choosing each center c_n ($n > 2$) from the set $E_{n-1} = E_{n-2} - (I_{n-1} \cup \{c_{n-1}\})$, till E_{n-1} is empty.



(a) Illustration of the *LC* with three center: c_1, c_2 and c_3.

(b) Cases of searching.

(c) Illustration of the *LSC* with three superclusters.

Fig. 1. List of Cluster (LC)

Note that, a cluster created first during construction has preference over the following ones when their corresponding covering radius overlap. All the elements that lie inside the cluster corresponding to the first center c_1 are stored in it, despite that they may also lie inside the subsequent clusters (Figure 1(a)). This fact is reflected in the search procedure. Figure 1(b) illustrates all the situations that may arise between a range query (q, r) and a given cluster.

During the processing of a range query (q, r), the idea is that if the first cluster is (c_1, r_1, I_1), we evaluate $d(q, c_1)$ and add c_1 to the result set if $d(q, c_1) \leq r$. Then, we scan exhaustively the objects in I_1 only if the range query (q, r) intersects the cluster with center c_1 and radius r_1, i.e. only if $d(q, c_1) \leq r_1 + r$ (q_1 in Figure 1(b)). Next, we continue with the remaining set of clusters following the construction order. However, if a range query (q, r) is totally contained in a cluster (c_i, r_i, I_i), i.e. if $d(q, c_i) \leq r_i - r$, we do not need to traverse the remaining clusters, since the construction process of the *LC* ensures that all the elements that are inside the query (q, r) have been inserted in I_i or in a previous clusters in the building order (q_2 in Figure 1(b)). In [5], authors analyzed different heuristics for selecting the centers, and showed experimentally that the best strategy is to choose the next center as the element that maximizes the sum of distances to previous centers. This is the heuristic used in our work.

2.2 List of Superclusters (LSC)

We propose a hierarchical multi-level *LC*, named *List of Superclusters* (*LSC*) that takes into account the organization of the GPU memory.

The construction of the *LSC* has two steps. First, based on the construction procedure of the *LC* with fixed size of K elements, we get N clusters of size K. Each i-th cluster is composed by its center C_i, covering radius r_i and the K nearest elements to C_i ($k\text{NN}_{\mathbb{U}}(C_i, K)$). These N clusters are named *superclusters* and integrate the first level of the hierarchy. In the second step, we create a *LC* index into each supercluster with their own elements following the construction procedure of the *LC* (Section 2.1).

To process a range query (q, r) we have to calculate the distances $d(C_i, q), i \in [1, N]$ between the centers of the superclusters and the query. We add the center

C_i to the result set if $d(C_i, q) \leq r$. Using triangle inequality we try to discard each supercluster (i.e. if $d(C_i, q) \leq r_i + r$), and for each non-discarded supercluster, we apply the searching procedure of the LC over its elements.

If we think in the clusters as unit of transfers to device memory in the GPU, the LSC makes better use of the bandwidth, because each non-discarded supercluster is a set of clusters that must be processed. The Figure 1(c) shows an example of a LSC index.

3 Strategies to Process Similarity Queries

In this section we describe our proposed methods to process range queries on a multi-GPU platform. All the following strategies are designed assuming that the database does not fit in device memory, i.e. just a subset of the clusters can be loaded at a time. Thus, for every batch of queries, the whole database may be (potentially) loaded to the GPU to perform the search.

In all the following strategies the kernels are launched with one CUDA Block per query. Each CUDA Block processes a different query, which has several advantages, such as, to be able to synchronize the threads that solve the same query, to exploit coarse-grained parallelism solving a batch of queries in parallel, or to exploit fine-grained parallelism solving a query with a set of threads.

3.1 1-Stage Strategy

In [2], assuming that the whole dataset fits into device memory, a multi-GPU strategy was proposed, using the LC index, and called *1-Stage*. We used the *1-Stage* strategy as baseline, but in this case we load in device memory just a percentage of the clusters at a time. The aim of this strategy is to solve each query in just one *kernel* in the GPU, avoiding to launch consecutive kernels and copying data to communicate them.

We used one CPU-thread per GPU, each one controls a different GPU. The centers, covering radius and their respective clusters are distributed among the GPUs (in a circular manner), and because of this, each query must be processed by all the GPUs.

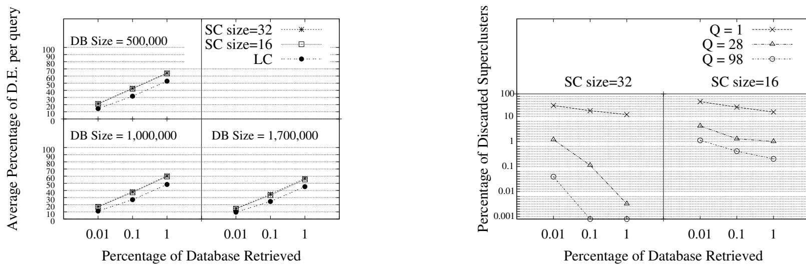
The discard of clusters and searching on them is performed inside the kernel, composed by two steps. (1) Each thread performs a distance evaluations between a different center and the query (corresponding to the current CUDA Block), and stores in shared memory a variable indicating if the cluster is discarded. (2) According to the variables in shared memory, all the non-discarded clusters are distributed (in a circular manner) among the threads, and each thread calculates the distance between an element and the query in the same kernel.

Due to the memory restrictions of space in the GPU, we load N centers and Q queries in device memory, and we process them iteratively. In the first iteration we process a batch of Q queries with N clusters, in the second iteration we load the next N cluster and process the same Q queries, and so on, until all clusters were loaded. The same process is repeated with all the batches of queries.

3.2 List of Superclusters on GPU

We created the *LSC* index with the method described in Section 2.2. We set the parameters of the *LSC* to create versions with N and $N/2$ clusters per supercluster, where N is the maximum number of cluster allowed in device memory. After loading a complete supercluster, a kernel is launched with Q CUDA Blocks (Q is the quantity of queries of the current query batch) to search into it.The kernel is composed of three steps as follows. (1) The first D threads cooperate to get the distance d between the center of the supercluster and the query, where D is the dimension of the elements. (2) As we described in Section 2.2, we use the distance d and the triangle inequality property to try to discard the supercluster. (3) If the supercluster is not discarded, then we search in the *LC* index inside the supercluster, with the method used by 1-Stage strategy (Section 3.1).

The Figure 2 shows results in sequential computation of the *LSC* against the *LC*, using the datasets described in Section 4. The Figure 2(a) compares the average of distance evaluations between the *LSC* and *LC*, where the *LC* always takes advantage. To try to know the efficiency for discarding of the *LSC*, the Figure 2(b) exposes the average of the percentage of discarded superclusters, processing query batches of different sizes (represented by Q in the graph). In this figure a supercluster is considerd discarded just if its covering radius does not intersect any of the Q queries of the current batch query. Therefore, the larger the Q, the less the probability of discarding a cluster. We observe that trying to discard superclusters taking account query batches of size 98 or higher, the *LSC* is able to discard less than 2% of the superclusters. The Figure 2(a) represents values with Q=1, and we can observe in the Figure 2(b) that the *LSC* reaches 69% of discard for Q=1 with the database of 500,000 elements, but even with this, the *LC* performs less distance evaluations. This is because the low discard of elements in the non-discarded superclusters by the *LSC*, close to 17%.



(a) Average of the distance evaluations (D.E.) per query, between *LSC* of 16 and 32 clusters per supercluster and the *LC* index.

(b) Discarding of supercluster in the *LSC* with 16 and 32 clusters per supercluster, using the database of 500,000 elements.

Fig. 2. Results in sequential computation of *LSC* and *LC*

Despite the total number of distance evaluations increases, in Section 4 we show that our implementation of *LSC* in GPU outperforms the *LC* one. This counterintuitive behavior is largely explained due to the higher transfer efficiency of the *LSC*. The minimum unit of discarding in the *LC* is a cluster and in the *LSC* is a supercluster, which also are the minimum unit of transfer. Therefore, the layout of the data to be transfered from CPU to GPU gets much more irregular when using *LC*, and the available bandwidth is poorly exploited.

3.3 Building a CPU-GPU Pipeline

To minimize the number of transfers to GPU and in order to increase the degree of parallelism, we developed a hybrid pipeline between CPU and GPU, where the CPU helps to discard some elements to avoid them to be transfered to the GPU. We used P CPU-threads, where P is the quantity of CPU-cores of the machine, and from those P the first G threads ($G < P$) manage a different GPU.

Considering that N is the allowed quantity of clusters in device memory, and Q is the quantity of the current batch query, the steps of the pipeline are as follows. (1) In CPU, we try to discard N clusters of the *LC* just using the center and covering radius of the clusters. For this we distribute (circularly) the clusters among the threads, and each thread discards its cluster if its covering radius does not intersect with any of the Q queries. (2) We load in GPU just the non-discarded clusters according to the previous step, and we process the queries with them. (3) While the second step (with the first G threads) is in execution, the first step (with the rest of the threads) is in execution too, but attempting to discard the next N clusters.

We implemented this pipeline for both *LC* and *LSC* indexes. In the case of the *LC* the threads that run on CPU cores try to discard clusters, and in the *LSC* they try to discard superclusters. As result, with this pipeline we get to load less quantity of clusters (or superclusters) in GPU.

3.4 Exploiting CUDA Asynchronous Copies

`cudaMemcpyAsync` allows to perform transfers to (and from) device memory while a kernel is in execution. This is possible by using CUDA *streams*, where each CUDA stream can contain a sequence of instructions. Copies and kernels from different streams can be executed at the same time.

Starting from the base non-pipelined implementation, we exploit the asynchronous copies for both *LC* and *LSC* indexes. If N is the quantity of clusters allowed in device memory, then we create two CUDA streams, and each stream is composed of the following instructions. (1) To copy $N/2$ clusters to device memory, and in the case of the *LSC* to copy one supercluster of $N/2$ clusters. (2) launch a kernel to process the queries with the loaded clusters (or supercluster). We create just two CUDA streams and not more, because this quantity makes a good balance in running time between copies and kernels, which effectively builds a two stage transfer - kernel pipeline.

We always copy the clusters of the LC or supeclusters of the LSC with just one `cudaMemcpyAsync` because the elements of a cluster or a supercluster are contiguous in the database; this is key to efficiently exploit the huge bandwidth between CPU and GPU, since short transfers cannot hide the initial latency.

3.5 Multi-pipeline Strategy

Our final proposal combined the two previous strategies in one *multi-pipeline strategy*. We use the LSC index (Section 3.2), and we create P CPU threads, one per CPU-core (Section 3.3), leaving G threads in charge of G GPUs ($G < P$). Each GPU create two CUDA streams to build a pipeline between copies and kernels (Section 3.4). The Figure 3 shows a scheme of this strategy, which is composed by three steps separated by OpenMP barriers, the steps are as follows. (1) Discard of superclusters with threads running on CPU-cores. (2) To copy the ID of the non-discarded superclusteres to be read by the threads in charge of GPUs. (3) Each GPU create two CUDA streams, and each stream copy to device memory one supercluster per `cudaMemcpyAsync`, and after a supercluster is loaded in device memory, immediately is launched a kernel to search on it. The steps 1 and 3 are executed on the same time, because the pipeline method described in Section 3.3.

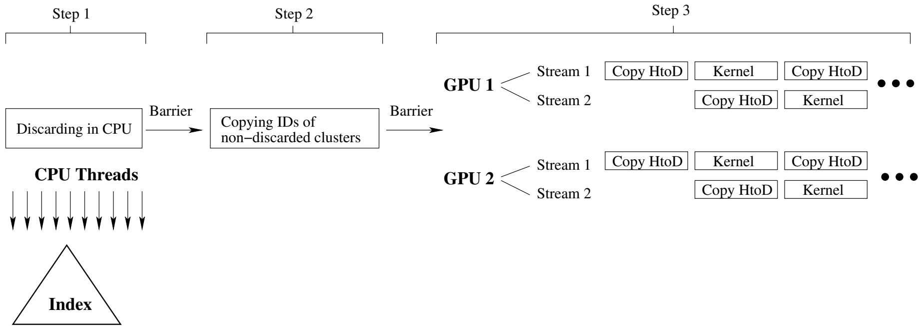


Fig. 3. Scheme of the multi-pipeline strategy

4 Experimental Results

All our GPU experiments were carried out on two NVIDIA Tesla M2070, and each one is shipped with 14 multiprocessors, 32 cores per multiprocessor, 48KB of shared memory and 5GB of device memory. The host CPU is a 2xIntel Quad-Xeon processor of 2.66GHz with 16 GB of RAM.

We have used as reference database the *CoPhIR* (Content-based Photo Image Retrieval) dataset [3]. This consists of metadata extracted from the Flickr photo sharing system. It is a collection of 106 million images containing for each image five MPEG-7 visual descriptors, specifically Scalable Colour, Colour Structure, Colour Layout, Edge Histogram, and Homogeneous Texture. For the purposes

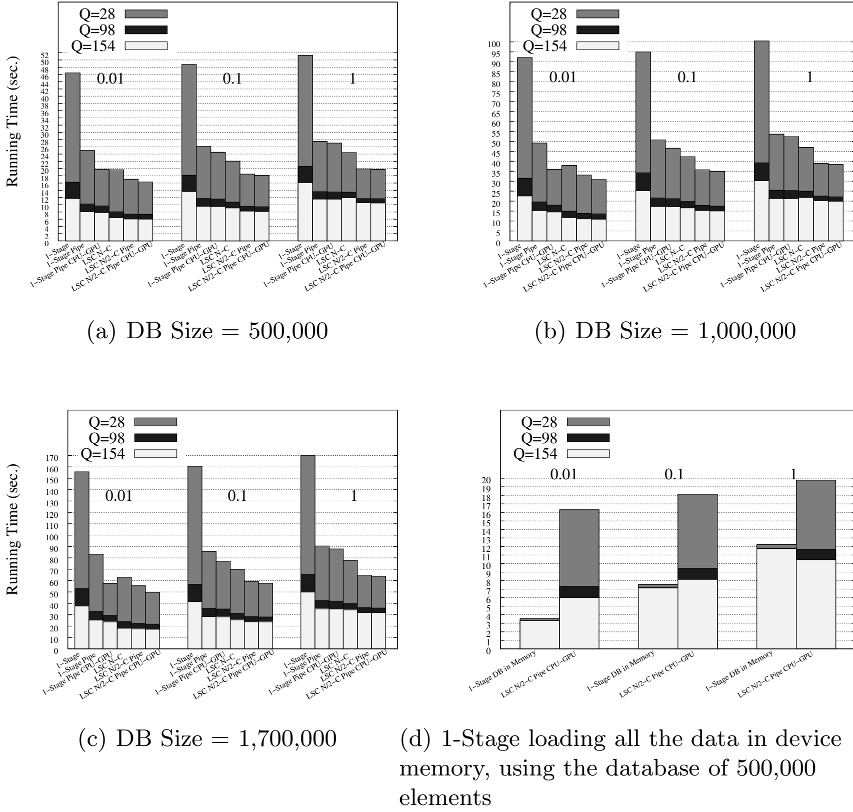
of this paper, we just used the *Colour Structure* MPEG-7 image feature, which represents a 64 dimensional vector for each image. We used the *Euclidean distance* as a distance measure. As in previous papers [7,9], the radii used were those that retrieve on average the 0.01%, 0.1% and 1% of the elements of the database per query.

To our knowledge, there is not a public and real query log for similarity search in images. But recently, a public website was presented in [10]. It applies the MUFIN [12] search engine for images of CoPhIR dataset and is used by many users all round the world. From this website, we got our query log, which represents the processed queries by several days. We used 30,000 queries that are represented by its *Colour Structure* MPEG-7 image feature of dimension 64. We have made this query log public [1].

The Figures 4(a), 4(b) and 4(c) presents the running time of all the strategies described in Section 3. The first column stands for the 1-Stage strategy (Section 3.1), which is implemented using the *LC* index. After loading N clusters a kernel is launched to search on them (N is the number of clusters allowed in device memory). The second column (*1-Stage Pipe*) stands for the 1-Stage strategy, but using two CUDA streams (Section 3.4), therefore after loading $N/2$ clusters in device memory we launch a kernel to search on them. The third column (*1-Stage Pipe CPU-GPU*) is similar to the second one, but implementing the pipeline CPU-GPU (Section 3.3), where the threads that run on CPU-cores try to discard clusters of the *LC* in parallel with the GPUs processing of the previous batch query. The fourth column (*LSC N-C*) stands for the *LSC* index (Section 3.2), with N clusters per supercluster, and after loading a supercluster a kernel is launched. The fifth column (*LSC N/2-C Pipe*) stands for the *LSC* index with $N/2$ clusters per supercluster, and using two CUDA streams (Section 3.4), therefore after loading a supercluster by a stream, a kernel is launched by using the same stream to search on it. The last column (*LSC N/2-C Pipe CPU-GPU*) stands for the *Multi-pipeline Strategy* described in Section 3.5.

In all our experiments we always set the cluster size equal to 256, because it has been empirically proved a good parameter. Therefore each supercluster of the *LSC* index is composed by clusters of size 256. We set the clusters allowed in device memory in N=32, and we just copy the results from GPU when a batch query is completely processed. In all the strategies, we copy a cluster of the *LC* (or supercluster in case of *LSC*) with one `cudaMemcpy`, or one `cudaMemcpyAsync` in the columns labeled with *Pipe*. All the strategies that implement the asynchronous copies pipeline (Section 3.4), use page-locked (pinned) memory to transfer data. This memory allows copies to device memory in parallel with kernel processing, and also decrease the time of the copies. Therefore, to be fair we use pinned memory for the transfers in all the strategies.

Each bar in the figures represents the running time of the corresponding strategy. For example, in the first bar of Figure 4(a), the running time of the 1-Stage strategy, processing the queries in batches of Q=28 is 46.4 seconds, with Q=98 is 16.2 seconds, and with Q=154 is 11.7 seconds. We process the queries in batches of 28, 98 and 154, because these numbers are multiples of 14, which is the



(a) DB Size = 500,000

(b) DB Size = 1,000,000

(c) DB Size = 1,700,000

(d) 1-Stage loading all the data in device memory, using the database of 500,000 elements

Fig. 4. Running time of the *LC* and *LSC* indexes combined with the pipelines described in Section 3

number of multiprocessors in our GPUs, and taking account that we are processing each query with a different CUDA Block, a multiple of 14 improves the load balance of CUDA Blocks across multiprocessors. Note that, in the worst case (if no discard is performed at the CPU level), the complete DB must be transferred from the CPU to the GPU for every batch query: 195 times for Q=154, 307 times for Q=98 and 1072 times for Q=28. It is imperative to efficiently hide this latency to attain good results.

Our baseline implementation, labeled as *1-Stage* strategy and described in section 3.1, achieves the worst performance in all the databases for all Q. The *1-Stage Pipe* strategy outperforms the previous one, because it reduces latency of the copies to device memory by using the pipeline described in Section 3.4, implemented with CUDA streams. The *1-Stage Pipe CPU-GPU* strategy outperforms the previous two, because the reduction in the quantity of clusters copied to device memory. This reduction is made by the threads running on CPU cores that calculate the distances between the centers and the batch query, avoiding to copy the discarded clusters.

The next three bars show the results for the proposed index, *List of superclusters*. The *LSC N-C* corresponds with the strategy explained in section 3.3 applied to the *LSC* index. As shown previously, this index finally performed more distance evaluations than the simpler *LC*; however, its final execution time is most of the times because the bandwidth is more efficiently managed: in each copy to device memory is transferred contiguous data of size N clusters to device memory (N is the number of clusters allowed in device memory). Next ,the *LSC N/2-C Pipe* strategy achieves better performance than the previous ones, because the implementation of the pipeline using CUDA streams, and the unit of transfers is a supercluster of $N/2$ cluster. This strategy is copying data while a kernel is in execution, hiding transfers latency. Finally, the strategy labeled as *LSC N/2-C Pipe CPU-GPU* (named as *multi-pipeline strategy* in Section 3.5) achieves the best performance. This strategy is similar to the previous one, but the threads running on CPU cores try to discard superclusters while the GPUs are processing the previous batch query.

The advantages of using the CPU-GPU pipeline (Section 3.3) in the *LSC* is more evident with Q=28, because the larger Q, the less is the discard of clusters (Figure 2(b)). This seems to indicate a certain degree of locality in the query log, which is lost when the batch is made too large. However, the much larger number of transfers due to a reduce Q does mitigate the benefits of this locality.

To complete our study, we consider the case where the whole DB actually fits in the GPUs main memory (i.e. the whole DB must just be copied once at the beginning of the process). Our smallest DB may be distributed amongst the two available GPUs, so we could compare our best implementation with this unrealistic scenario.Figure 4(d) shows the results.

Not surprisingly, the *all-fit* implementation outperforms our proposal when searching with small radius. Also as expected, in this ideal version, there is almost no penalty when reducing the Q: it does not entail further transfers, so there is not huge penalty form that side. However, it is very noticeable that, for the larger search radius (1% of the DB retrieved) our implementation actually outperforms the *all-fit* version for Q=154. With such a large radius, the discard efficiency is quite low. Thus, kernel execution times are always able to completely hide transfers penalties. Then, we only pay the latency of transferring one *supercluster* per batch of queries. The higher the Q, the lesser the number of batches and, for this example, we are able to be competitive with the *all-fit* version. For the smallest Q value, the number of not-hidden transfers increase too much (and the kernel work decreases), largely degrading performance.

5 Conclusions

In this paper we have presented a hierarchical multi-level structure, built on the *LC* index named *List of Superclusters* (*LSC*), to solve similarity queries in metric spaces. The *LSC*, which is composed by *superclusters*, has been designed to perform well on GPUs. A supercluster is made by a center, a covering radius and elements, but with the elements of each supercluster is created a *LC* index.

Grouping clusters in superclusters allows for a fast discard at CPU level and, using it as the minimal CPU-GPU transfer unit, ensures that the bandwidth is always efficiently exploited.

To study the index efficiency, we have implemented a pipelined hybrid CPU-GPU version of both the *LC* and *LSC* indexes. The CPUs perform a first round of discards for a batch query Q_i while the GPUs are finishing the processing of the previous batch, Q_{i-1}. Moreover, the CPU-GPU transfers and the GPU kernels execution is further pipelined using *streams* and asynchronous copies. The transfer latency is almost completely hidden that way; indeed, even if the complete list of clusters is copied for each batch query (except those clusters discarded by the CPU), the total exposed latency may be even lower than the experienced when transferring the complete DB just once.

Our study with a real query log for similarity search in images, shows that there exits a locality amongst queries: i.e. the sets of clusters accessed by two consecutive queries have a non null intersection. This motivates further exploration to reduce transfers by carefully scheduling queries.

References

1. Query log, `http://kataix.umag.cl/~ribarrie/Programs.html`
2. Barrientos, R., Gómez, J., Tenllado, C., Prieto, M., Marin, M.: Range query processing in a multi-gpu environment. In: 10th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA 2012), pp. 419–426 (2012)
3. Bolettieri, P., Esuli, A., Falchi, F., Lucchese, C., Perego, R., Piccioli, T., Rabitti, F.: Cophir: a test collection for content-based image retrieval. CoRR abs/0905.4627 (2009), `http://cophir.isti.cnr.it`
4. 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 (2000)
5. Chávez, E., Navarro, G.: A compact space decomposition for effective metric indexing. Pattern Recognition Letters 26(9), 1363–1376 (2005)
6. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquín, J.L.: Searching in metric spaces. ACM Computing Surveys 33(3), 273–321 (2001)
7. Costa, V.G., Barrientos, R.J., Marín, M., Bonacic, C.: Scheduling metric-space queries processing on multi-core processors. In: Danelutto, M., Bourgeois, J., Gross, T. (eds.) PDP, pp. 187–194. IEEE Computer Society (2010)
8. Marin, M., Ferrarotti, F., Gil-Costa, V.: Distributing a metric-space search index onto processors. In: 39th International Conference on Parallel Processing, ICPP 2010, pp. 433–442. IEEE Computer Society, San Diego (2010)
9. Navarro, G., Uribe-Paredes, R.: Fully dynamic metric access methods based on hyperplane partitioning. Information Systems 36(4), 734–747 (2011)
10. Novak, D., Batko, M., Zezula, P.: Generic similarity search engine demonstrated by an image retrieval application. In: 32nd ACM SIGIR Conference on Research and Development in Information Retrieval, p. 840. ACM, Boston (2009)

11. Uribe-Paredes, R., Arias, E., Sánchez, J.L., Cazorla, D., Valero-Lara, P.: Improving the performance for the range search on metric spaces using a multi-GPU platform. In: Liddle, S.W., Schewe, K.-D., Tjoa, A.M., Zhou, X. (eds.) DEXA 2012, Part II. LNCS, vol. 7447, pp. 442–449. Springer, Heidelberg (2012)
12. Zezula, P.: Multi feature indexing network mufin for similarity search applications. In: Bieliková, M., Friedrich, G., Gottlob, G., Katzenbeisser, S., Turán, G. (eds.) SOFSEM 2012. LNCS, vol. 7147, pp. 77–87. Springer, Heidelberg (2012)
13. Zezula, P., Amato, G., Dohnal, V., Batko, M.: Similarity Search: The Metric Space Approach. Advances in Database Systems, vol. 32. Springer (2006)