Pipeline strategies to accelerate range query processing on a multi-GPU environment

Ricardo J. Barrientos
Department of Computer Architecture, ArTeCS Group,
Complutense University of Madrid, Spain.
Email: ribarrie@ucm.es

***Abstract*—Nowadays, similarity search is becoming a field of increasing interest because these kinds of methods can be applied to different areas in computer science and engineering, such as voice and image recognition, text retrieval, and many others. However, when processing large volumes of data, query response time can be quite high. In this case, it is necessary to apply mechanisms in order to significantly reduce the average query response time. In this sense, the parallelization of the metric structures processing is an interesting field of research. Currently, most of the previous and current works developed in this area are carried out considering classical distributed or shared memory platforms. However, modern GPU/Multi-GPU systems offer a very impressive cost/performance ratio as compared to multiprocessor or multicomputer platforms that are usually more expensive gaining in significance and popularity within the scientific computing community. More recently, GPUs have been proposed to evaluate similarity queries for indexes that remains statically stored in GPU's memory. In this paper we propose two different pipelines to accelerate the process of similarity queries in datasets large enough not to fit in memory of the GPUs. The first pipeline makes use of CPU-cores and GPUs in a hybrid algorithm, and the second one is implemented into the GPU. The results show that the best performance is achieved with both pipelines at the same time.**

***Index Terms*—Similarity search, range queries, metric spaces, GPU, multi-GPU.**

I. Introduction

Currently, the search of similar objects in a large collection of stored objects in a metric database has become a most interesting problem. This kind of search can be found in different applications such as voice and image recognition, bioinformatics, plagiarism detection and many others. A typical query for these applications is the *range search* which consists in obtaining all the objects that are at some given distance from the consulted object. Basically, similarity is modeled in many interesting cases through metric spaces, and the search of similar objects through *range search* or *nearest neighbors*.

A metric space $(\mathbb{X}, d)$ is a set $\mathbb{X}$ and a distance function $d : \mathbb{X}^2 \rightarrow \mathbb{R}$, so that $\forall x, y, z \in \mathbb{X}$ fulfills the properties of positiveness $[d(x, y) \geq 0,$ and $d(x, y) = 0$ iff $x = y]$, symmetry $[d(x, y) = d(y, x)]$ and triangle inequality $[d(x, y) + d(y, z) \geq (d(x, z)]$. This concept of similariy is associated to the concept of Metric space data structures, which can be grouped into two classes [1]: *clustering*-based (*BST* [2], *GHT* [3], *M-Tree* [4], *GNAT* [5], and many others), and *pivots*-based methods (*LAESA* [6], *FQT* and its variants [7], *Spaghettis* and its variants [8], *FQA* [9], *SSS-Index* [10] and others).

On the other hand, the increasing size of databases and the emergence of new data types create the need to process large volumes of data, and as a consequence, it makes necessary to develop new algorithms that deal with this amount of data. In this context, the use of parallel resources becomes essential. To address this problem, typical solutions are based on distributed memory platforms (cluster of PCs), shared memory platforms (multicore) or both (cluster of multicores).

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.

Metric data structures are used to perform an efficient filtering on the database and reduce the search space. However, their use could introduce 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. The first one coordinates the CPU and the GPU, being able to hide most of the CPU-GPU data transfer latency. The second one, combines the process of the GPU with the transfers between CPU ang GPU.

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

II. 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:

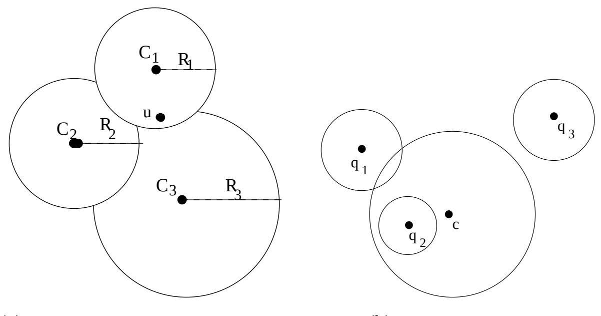
Metric Space [11]: A *metric space* (X, d) is composed of an universe of valid objects $\mathbb{X}$ and a *distance function* $d : \mathbb{X} \times \mathbb{X} \rightarrow \mathbb{R}^+$ defined among them. The distance function determines the similarity between two given objects and holds several properties 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]: 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*) [12] 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.

In [13], [14] 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



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

(b) Cases of searching.

Fig. 1. List of Cluster (LC).

propose solutions to deal with large databases, which is usually the real case, where the databases fits just partially on the GPU memory.

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

A. List of Clusters (LC)

This index [15], [12] 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.

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 [12], 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.

III. Graphic Processing Units (GPU)

This section presents an overview of the architecture used by NVIDIA's GPUs [16] and the programming model offered by their CUDA drivers, in order to expose the challenges that compilers have to face to produce efficient codes for these devices.

A GPU is a device that can be used as a high performance coprocessor, suitable for accelerating data parallel codes. The program running on the CPU (the host) must explicitly manage data transfers from host memory to device memory and vice versa, and can control the execution of programs on the device.

Figure 2 gives an schematic view of the actual hardware of modern NVIDIA's GPU. Physically, GPU cores (processors) are organized into several *multiprocessors*. Each multiprocessor is composed of several scalar processors that share a single instruction unit. The processors within a multiprocessor execute in lock-step, all the same instruction each cycle, but on different data. Each multiprocessor can maintain hundreds of threads in execution. These threads are organized in sets, called *warps*.[1] Every cycle, the hardware scheduler of each multiprocessor chooses the next warp to execute (i.e., no individual threads but warps are swapped in and out), using fine grain simultaneous multithreading to hide memory access latencies. This execution model is called Single Instruction Multiple Thread (SIMT) by Nvidia.

Regarding the memory hierarchy, all multiprocessors can access the same on-board DRAM memory (global memory in CUDA parlance) through a high bandwidth bus. This global memory is banked, which allows the hardware to coalesce several simultaneous memory accesses to adjacent positions into a single memory transaction. In addition, each multiprocessor contains a smaller SRAM memory. In more recent GPUs (starting from the *Fermi* architecture [16]) this SRAM can be configured as scratch pad (i.e., a software controlled memory) and hardware controlled cache memory. The user can decide, with certain restrictions, the amount of cache and scratch pad needed. These newer GPUs also incorporate a L2 cache common to all multiprocessors. Finally, GPU multiprocessors can also access the global memory through a special read-only two level hierarchy of so called *texture* caches, that can be configured to capture 2D locality.

This model is exposed to the programmer by the CUDA driver. It allows to control the execution of a *kernel* on the device. A *kernel* consists of a sequential piece of code that has to be executed by a large set of threads on the GPU multiprocessors. Those threads are grouped into warps. Threads within a warp are simultaneously executed on the scalar processors of a single multiprocessor in lock step. 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 are all executed in the same multiprocessor, and are then able to cooperate with each other by (1) efficiently sharing data through

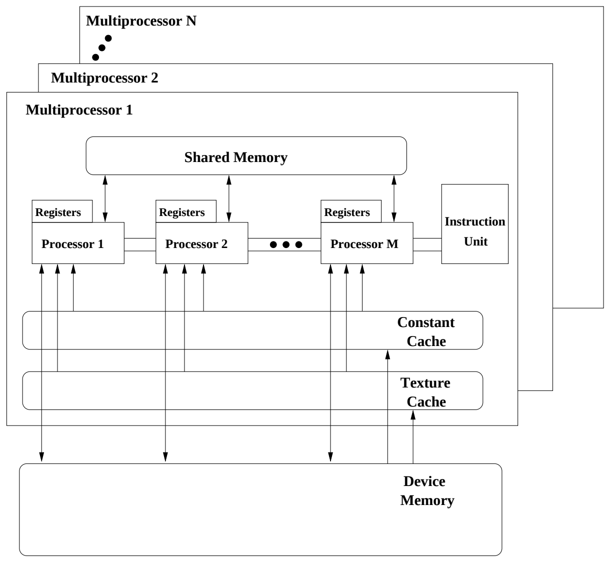


Fig. 2. The CUDA programming model is designed for compute. It represents the GPU as a coprocessor that integrates several multiprocessors and a complex memory hierarchy.

[1]Currently, there are 32 threads per warp

the shared low latency local SRAM memory and by, (2) synchronizing their execution via barriers. In contrast, threads from different blocks can be (potentially) scheduled on different multiprocessors and thus they can only coordinate their execution via accesses to the 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. 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-wise memory accesses and to avoid bank conflicts on shared memory accesses.

To summarize, we can draw some conclusions on the challenges being faced when mapping code to this kind of devices. First the programmer needs to partition the code into host and GPU code. The parallel code pieces for the GPU must be mapped onto the CUDA model of blocks and threads. Here we have two different levels of parallelism, independent threads that are assigned to different blocks and cooperating threads, that are assigned to the same block forming warps. The latter should exhibit SIMD parallelism to avoid warp divergences and to minimize the number of non-coalesced memory accesses (threads in the same warp should access adjacent memory addresses). In addition, to reduce bandwidth requirements, data locality should be efficiently exploited in the register file and the local shared memories. This implies that the programmer should explicitly consider, schedule and express data transfers between the different memories available[2], trying to reduce the accesses to the global memory, and also take account the existence of the new caches, which introduce an additional variable that should be considered.

IV. 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.

[2]This programmer control is larger when using the SRAM mainly as a software controlled memory, but hardware controlled cache must also be taken into account during the mapping

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.

A. *1-Stage Strategy*

The authors in [14], assuming that the whole dataset fits into device memory, a multi-GPU strategy was proposed, using the *LC* index, and called *1-Stage*. It widely outperformed sequential and multi-core versions. 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.

B. *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.

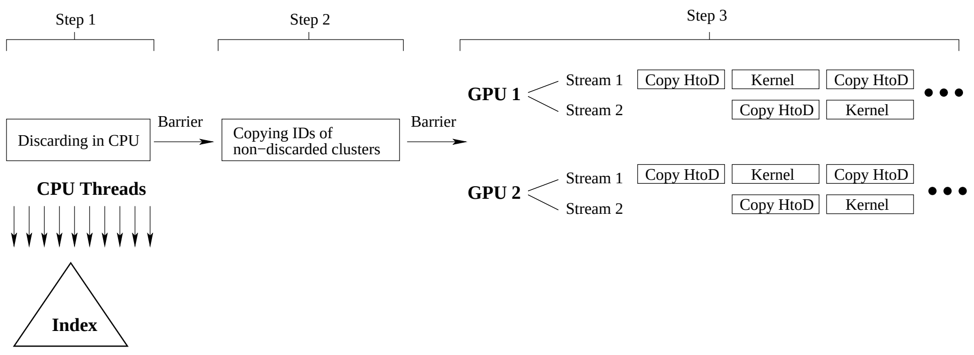


Fig. 3. Scheme of the multi-pipeline strategy.

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 the latter 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) The non-dicarded cluster are processed with the GPUs. While the third 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.

C. 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.

We exploit the asynchronous copies starting from the base non-pipelined implementation. 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 no 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* with just one `cudaMemcpyAsync` because the elements of a cluster 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.

D. Multi-pipeline Strategy

Our final proposal combined the two previous strategies in one *multi-pipeline strategy*. We create P CPU threads, one per CPU-core, leaving G threads in charge of G GPUs ($G < P$) to build the pipeline described in Section IV-B. Each GPU creates two CUDA streams to build the pipeline described in Section IV-C between copies and kernels. 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 cluster per `cudaMemcpyAsync`, and after a cluster 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.

V. 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 [17]. 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

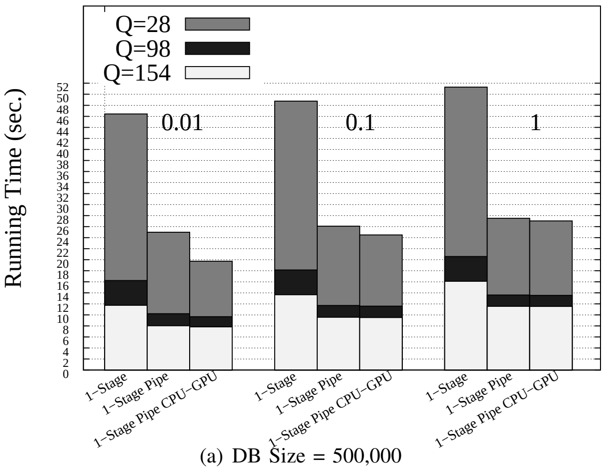
Euclidean distance as a distance measure. As in previous papers [18], [19], the radii used were those that retrieve on average the 0.01%, 0.1% and 1% of the elements of the database per query.

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 [20]. It applies the MUFIN [21] search engine for images of CoPhIR dataset and is used by many users all around 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 [22].

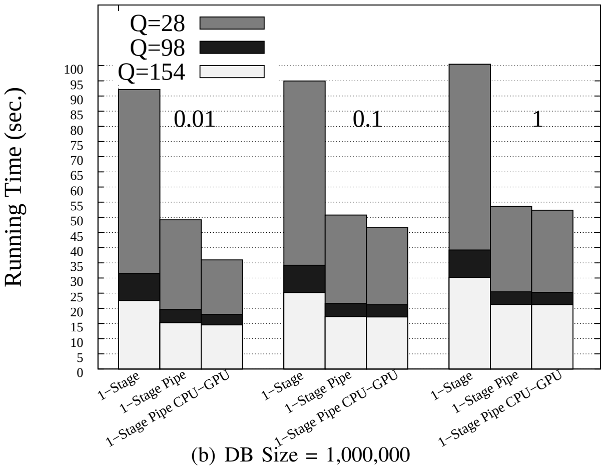
The Figures 4(a), 4(b) and 4(c) presents the cumulative running time of all the strategies described in Section IV implemented with the *LC* index, solving the queries in batches of $Q = 28, 98, 154$. The first column of the figures stands for the 1-Stage strategy (Section IV-A), where after loading 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 IV-C), 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*) stands for the multi-pipeline strategy (Section IV-D) which implements both pipelines.

In all our experiments we always set the cluster size equal to 256, because it has been empirically proved a good parameter. 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* with one `cudaMemcpy`, or one `cudaMemcpyAsync` in the columns labeled with *Pipe*. All the strategies that implement the asynchronous copies pipeline (Section IV-C), 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.

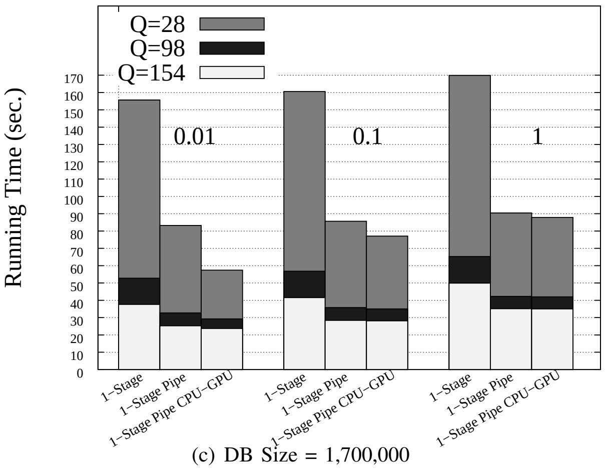
Each bar in the figures represents the running time of the corresponding strategy. For example, in the first bar of Figure 4(a), the running time of the 1-Stage strategy, processing the queries in batches of Q=28 is 46.4 seconds, with Q=98 is 16.2 seconds, and with Q=154 is 11.7 seconds. We process the queries in batches of 28, 98 and 154, because these numbers are multiples of 14, which is the number of multiprocessors in our GPUs, and taking account that we are processing each query with a different CUDA Block, and each CUDA Block is completely processed in just one multiprocessor



(a) DB Size = 500,000



(b) DB Size = 1,000,000



(c) DB Size = 1,700,000

Fig. 4. Running time of the *LC* index combined with the pipelines described in Section IV.

(Section III), 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 database 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 IV-A, 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 IV-C, 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 advantages of using both pipelines is more evident with Q=28, because the larger Q, the less the discard of clusters. 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.

VI. Conclusions

In this paper we have presented two different pipelines to accelerate similarity search on metric spaces using a multi-GPU platform, and we have used the *LC* index because its suitable features for the GPU.

The first pipeline is a hybrid CPU-GPU version of the *LC* index, where 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}. The second one is implemented into the GPUs using CUDA streams and asynchronous copies, where the CPU-GPU transfers and the GPU kernels are executed in parallel. The transfer latency is almost completely hidden that way; indeed, even if the complete list of clusters is copied for each batch query, the total exposed latency may be even lower than the experienced when transferring the complete database just once.

We outperformed the baseline (*1-Stage*) strategy proposed in a previous paper, where its performance outperformed sequential and multi-core versions.

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] 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] I. Kalantari and G. McDonald, "A data structure and an algorithm for the nearest point problem," *IEEE Transactions on Software Engineering*, vol. 9, no. 5, 1983.

[3] J. Uhlmann, "Satisfying general proximity/similarity queries with metric trees," in *Information Processing Letters*, 1991, pp. 40:175–179.

[4] P. Ciaccia, M. Patella, and P. Zezula, "M-tree: An efficient access method for similarity search in metric spaces," in *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*. Morgan Kaufmann, 1997, pp. 426–435.

[5] S. Brin, "Near neighbor search in large metric spaces," in *the 21st VLDB Conference*. Morgan Kaufmann Publishers, 1995, pp. 574–584.

[6] L. Micó, J. Oncina, and E. Vidal, "A new version of the nearest-neighbor approximating and eliminating search (aesa) with linear preprocessing-time and memory requirements," *Pattern Recognition Letters*, vol. 15, pp. 9–17, 1994.

[7] R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu, "Proximity matching using fixedqueries trees." in *5th Combinatorial Pattern Matching (CPM'94)*, ser. LNCS 807, 1994, pp. 198–212.

[8] E. Chávez, J. 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'99)*. Los Alamitos, USA: IEEE CS Press, 1999, pp. 38–46.

[9] E. Chávez, J. Marroquín, and G. Navarro, "Fixed queries array: A fast and economical data structure for proximity searching," *Multimedia Tools and Applications*, vol. 14, no. 2, pp. 113–135, 2001.

[10] 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.

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

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

[13] R. Uribe-Paredes, E. Arias, J. L. Sánchez, D. Cazorla, and P. Valero-Lara, "Improving the performance for the range search on metric spaces using a multi-gpu platform," in *23rd International Conference on Database and Expert Systems Applications (DEXA 2012)*, ser. LNCS, vol. 7447. Springer, 2012, pp. 442–449.

[14] R. Barrientos, J. Gómez, C. Tenllado, M. Prieto, and M. Marin, "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.

[15] E. Chavéz and G. Navarro, "An effective clustering algorithm to index high dimensional metric spaces," in *The 7th International Symposium on String Processing and Information Retrieval (SPIRE'2000)*. IEEE CS Press, 2000, pp. 75–86.

[16] NVIDIA, "Nvidia's next generation cuda compute architecture: Fermi," Tech. Rep., 2010.

[17] P. Bolettieri, A. Esuli, F. Falchi, C. Lucchese, R. Perego, T. Piccioli, and F. Rabitti, "Cophir: a test collection for content-based image retrieval," *CoRR*, vol. abs/0905.4627, 2009. [Online]. Available: http://cophir.isti.cnr.it

[18] V. Gil-Costa, R. J. Barrientos, M. Marin, and C. Bonacic, "Scheduling metric-space queries processing on multi-core processors," in *18th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP 2010)*. Pisa, Italy: IEEE Computer Society, February 2010, pp. 187–194.

[19] 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.

[20] D. Novak, M. Batko, and P. Zezula, “Generic similarity search engine demonstrated by an image retrieval application,” in *32nd ACM SIGIR Conference on Research and Development in Information Retrieval*. Boston, MA, USA: ACM, 2009, p. 840.

[21] P. Zezula, “Multi feature indexing network mufin for similarity search applications,” in *38th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2012)*, ser. LNCS, vol. 7147. Springer, 2012, pp. 77–87.

[22] “Query log. *http://kataix.umag.cl/∼ribarrie/Programs.html*.”