Metric Data Structures Supported by Heterogeneous Systems

Metric Data Structures Supported by Heterogeneous Systems

by

Roberto Uribe-Paredes, Enrique Arias,
José L. Sánchez, Diego Cazorla

Technical Report **#DIAB-13-05-2** May, 2013.-

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS
ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE CASTILLA-LA MANCHA
Campus Universitario s/n
Albacete - 02071 - Spain
Phone +34.967.599200, Fax +34.967.599224

Metric Data Structures Supported by Heterogeneous Systems

May 14, 2013

Abstract

In real applications, when dealing with high volume of data, it is necessary the use of parallel platforms in order to obtain results in a reasonable time. Nowadays, GPU/MultiGPU devices are widely used to get this reasonable time at a low price. The GPU/MultiGPU are managed by a CPU/core/multicore host. That is why these kinds of systems are called heterogeneous systems. However, in most cases the CPU/core/multicore is idle when the GPU/MultiGPU devices are processing. As a consequence, the theoretical peak performance of the underlying architecture decreases and, of course, the term heterogeneous platform becomes more a conventional term than a real term.

In this paper, we have carried out a real heterogeneous implementation focusing on similarity search. Similarity search is becoming a field of increasing interest because these kinds of methods can be applied to different areas in science and engineering, such as pattern recognition, information retrieval, etc. This search is carried out over metric indexes that allow to decrease the number of distance evaluations during the search process, improving the efficiency of the overall process. Our heterogeneous platform consists of a 8 core processor and 4 GPUs. Then, our algorithms exploit the computational resources executing at the same time on cores and on the GPUs. The scheduling is carried out by OpenMP considering dynamic scheduling. In order to fix the best chunk or block size of the queries to be processed by each processing element, different approaches have been considered.

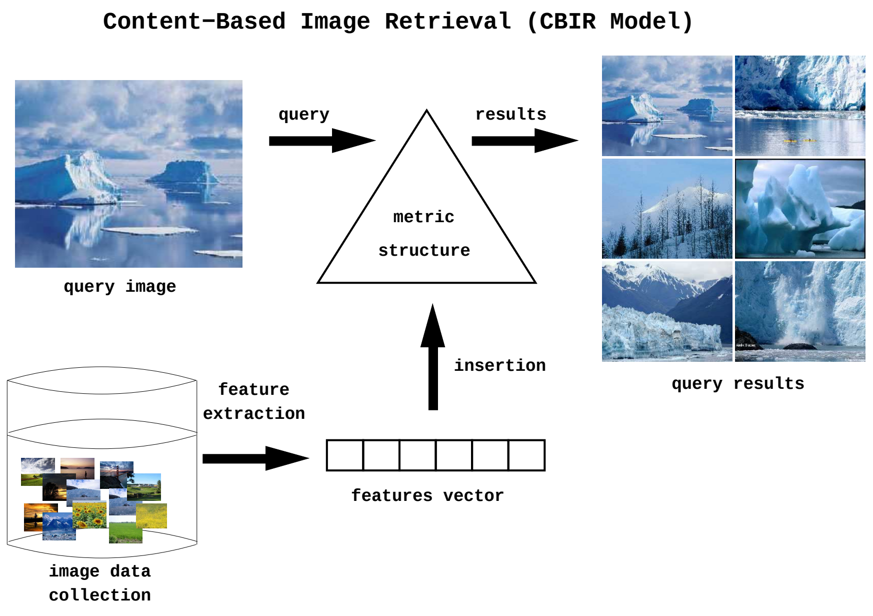


Figure 1: Similarity Search applied on Content-Based Image Retrieval System.

Keywords: Similarity search, metric spaces, range queries, GPU platforms, multi-core platforms.

1 Introduction

In the last decade, 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, data mining, plagiarism 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. Figure 1 represents a typical content-based image retrieval scheme.

The increasing size of databases and the emergence of new data types create the need to process large volumes of data. In order to be able to deal with this amount of data and get results in a reasonable time, the use of parallel platforms is necessary. For example, when a query (search a word) is made to a retrieval information system in a database (dictionary), the response time has to be short in order to accomplish the user requirements.

Nowadays, a typical server site is comprised by a cluster of multicores and multiGPUs [26, 25]. In most applications, data is distributed among the different processors in a cluster, and is processed in a multicore/multiGPU platform. Therefore, the algorithms have to be optimized to deal with the underlying heterogeneous architecture. Moreover, the algorithms have to be able to be executed on cores and GPUs at the same time in order to obtain the best performance.

In this paper the authors present two different implementations considering two different scenarios. The first one takes into account that the queries are completely processed in a unique kernel. Under this scenario more objects of the database can be allocated to the global memory of the GPU. Evidently, the consequence is that the time spent in data transfer is increased and more calls to the unique kernel are needed. Then, the second scenario corresponds to a block-oriented implementation transferring a chunk of queries at a time and processing it in the unique kernel, reducing the time spent in data transfer and calls to the kernel. However, less objects of the database can be allocated in the global memory of the device. Moreover, a bigger data structure is required to store the results of the queries.

This paper is structured as follows. First, the following subsections introduce a description of the GPU architecture and the programming model, then the concept of similarity search in metric spaces is introduced and finally related work is presented. Thinking on the GPU implementation, Section 2 compares well-known data structures pivot-based against a generic structure in the sequential implementation in order to achieve a suitable structure for GPU platforms, presenting the experimental platform, experimental results and discussion. From the previous starting point, the two heterogeneous implementations considering the two scenarios previously described are detailed in Sections 4.1 and 4.2. Finally, in Section 5 the conclusions and future work are outlined.

1.1 Graphics Processing Units

Graphics Processing Units (GPUs) are considered as serious challengers for high-performance computing solutions because of its suitability for massively parallel processing. Their high number of computing cores and high-speed memory access have facilitated their application to many real-world applications such as bioinformatics, computational finance, numerical computing, image/video processing, engineering simulations, physics and chemistry, etc.

In order to adapt to the available hardware and obtain good performance by exploiting the full potential of the GPU, the main manufacturers, such as NVIDIA and AMD/ATI, proposed new languages or extensions for the most commonly-used high-level programming languages. NVIDIA proposed CUDA, a software platform for programming massively parallel high-performance. The NVIDIAs CUDA Programming Model [1] considers the GPU as a computational device capable to execute a high number of parallel threads. CUDA includes C/C++ software development tools, function libraries, and a hardware abstraction mechanism that hides the GPU hardware to the developers.

To take advantage of the GPU computational capacity, hundreds of threads are launched simultaneously to execution. In order to hide memory access latency, an efficient usage of the memory hierarchy must be achieved. A CUDA kernel executes a sequential code in a large number of threads in parallel. The threads are organized into grids of thread blocks. Threads within a block can work together efficiently exchanging data via a local shared memory and synchronize low-latency execution through synchronization barriers (where threads in a block are suspended until they all reach the synchronization point). By contrast, the threads of different blocks in the same grid can only coordinate their execution through high-latency accesses to global memory (the graphic board memory). The programmer arranges parallelism by declaring the number of thread blocks, the number of threads per thread block and their distribution, subject to the program constraints.

1.2 Similarity Search in Metric Spaces

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 ($d(x,y) + d(y,z) \geq (d(x,z)$).

In a given metric space $(\mathbb{X}, d)$ and a finite data set $\mathbb{Y} \subseteq \mathbb{X}$, a series of queries can be made. The basic query is the *range query* (x, r), a query being $x \in \mathbb{X}$ and a range $r \in \mathbb{R}$. The range query around x with range r (or radius r) is the set of objects $y \in \mathbb{Y}$ such that $d(x,y) \leq r$. A second type of query that can be built using the range query is *k nearest neighbors* (*kNN*), the query being $x \in \mathbb{X}$ and object k. k nearest neighbors to x are a subset $\mathbb{A}$ of objects $\mathbb{Y}$, such that if $|\mathbb{A}| = k$ and an object $y \in \mathbb{A}$, there is no object $z \notin A$ such that $d(z,x) \leq d(y,x)$.

There are several metric data structures (or metric access methods) aimed to minimize the amount of distance evaluations made to solve the query and thereby reduce the processing time. Searching methods for metric spaces are mainly based on dividing the space using the distance to one or more selected objects. As they do not use particular characteristics of the application, these methods work with any type of objects [8].

Some structures are based in clustering and others in pivots. The clustering-based structures divide the space into areas, where each area has a so-called centre. Some data is stored in each area, which allows easy discarding the whole area by just comparing the query with its centre. Some clustering-based indexes are BST [15], GHT [21], M-Tree [9], GNAT [6], EGNAT [19] and many others.

In the pivots-based methods, a set of pivots is selected and the distances between the pivots and database elements are precalculated. When a query is made, the query distance to each pivot is calculated and the triangle inequality is used to discard the candidates. Its objective is to filter objects during a request through the use of a triangular inequality, without really measuring the distance between the object under request and the discarded object. Some pivots-based indexes are LAESA [18], FQT and its variants

[3], Spaghettis and its variants CMBY99, FQA [7], SSS-Index [20] and others.

Array-type structures implement these concepts directly. The difference among different array-type structures lies on extra structures used to reduce the computational cost to obtain the number of candidates keeping invariable the evaluation of distances. Although there are also tree-type structures, however, array-type are the best to be implemented on GPU-based platforms [24].

More details on current metric structures can be found in [8, 14].

1.3 Related Work

Currently, there are many parallel platforms for the implementation of metric structures. In this context, basic research has focused on technologies for distributed memory applications, using high level libraries for message passing such as MPI or PVM, and shared memory, using the language or directives of OpenMP [13].

Some studies have focused on different structures parallelized on distributed memory platforms using MPI or BSP. In these cases, the aim was not only the parallelization of the algorithms, but also the balanced distribution of data [27, 2, 12].

In terms of shared memory, some studies analyze data distribution on multicore nodes. Other works propose combining multithread queries processing totally asynchronous with massively synchronous, depending on traffic [11].

Most of the previous and current works developed in this area are carried out considering classical distributed or shared memory platforms, but few studies exist focus on GPU-based platforms. Some solutions considered till now developed on GPUs are based on kNN queries without using data structures. This means that GPUs are basically applied to exploit its parallelism only for exhaustive search [16, 10]. In general, in the previous works the parallelization is applied in two stages. The first one consists in building the distance matrix, and the second one consists in sorting this distance matrix in order to obtain the final result.

A particular variant of the above proposed algorithms is presented in [4] where the search is structured into three steps. In the first step each block solves a query. Each thread keeps a heap where it stores the kNN nearest elements processed by this thread. Secondly, a reduction operation is applied to obtain a final heap. Finally, the first k elements of this final heap are taken as a result of the query.

In [4, 5] a GPU version of the structure List of Clusters is presented. However, in this case, a single kernel is used and no restrictions about the size of the memory are made, i.e., they consider they can store in the memory of the device the whole structure, the database, the pivots and all the queries.

In [24] metric structures on a GPU were used and the results were compared with sequential versions, considering 2 and 3 kernels working on the GPU.

2 Preparing a Pivot-based Metric Data Structure

In metric spaces literature different data structures have been considered and classified into two categories: clustering-based methods and pivot-based methods. In Section 1.2 several references have been taken into account.

In this work, clustering-based methods as EGNAT or GHT have not been considered due to the fact that the tree-type structures do not suit well into the GPU architecture. Thus, we only have considered array-type data structures as SSS-Index, Spaghettis and LAESA [23].

SSS-Index, Spaghettis and LAESA can be considered as a bidimensional array, that is, a distance table. These kinds of structures are appropriate to be mapped into a GPU architecture, benefiting the execution of simple instructions by several threads over different data and avoiding jumps on the device memory. In the case of the tree-type structures, mapping the structure on the GPU device implies jumps along the structure reducing the performance. This is the reason why we assess that tree-type structures do not suit well into the GPU architecture.

In this work, the authors considered a Generic Metric Structure (GMS) that does not take into account a pivot selection, or, the pivots are randomly selected, as opposite to SSS-Index and LAESA data structure. Besides, Spaghettis data structure needs to be reordered before the search process. In GMS the structure does not need to be ordered, and then it is possible to obtain better performance on a GPU due to the fact that the ordering process is computationally expensive in this kind of device.

Therefore, this section tries to put in context different array-type data structures before selecting one of them thinking of the GPU implementations.

2.1 Pivot-based and Array-type Data Structures

In the literature it is possible to find different array-type metric data structures. In this section, we describe those taken into account in this work.

In particular, the considered metric data structures are:

Spaghettis: It is an array-type structure based on pivots and does not assume any pivot selection method. However, each entry in the array, that represents distances between an element in the database and the pivots, is sorted with respect to this distance, obtaining a reduction on the execution time by means of a binary search. In this work, the array is sorted considering only the first pivot.

SSS-Index: SSS-Index (*Sparse Spatial Selection*) [20] is basically the generic structure varying the way in which the pivots are selected. The selection methods will be introduced later.

LAESA: Like *SSS-Index*, it is a structure similar to the generic one, but the selection of pivots is carried out using a method called *Maximun Sum of Distances* (*MSD*).

The choice of these metric structures is motivated because they are representative of this field of knowledge, and we have considered structures based on pivots and array-type.

With respect to the choice of the pivot selection method, we have considered the following:

Randomly: As the name suggests, this method consists in selecting randomly the set of pivots.

Sparse Spatial Selection (SSS): Sparse Spatial Selection [20] is a method to select a dynamic set of pivots or centres distributed in the space. Let $(\mathbb{X}, d)$ be a metric space $\mathbb{U} \subset \mathbb{X}$ and M the largest distance between all pairs of objects, i. e. $M = max\{d(x,y)/x,y \in \mathbb{U}\}$. Initially, the set of pivots contains the first element of the collection. After that, an element $x_i \in \mathbb{U}$ is selected as a pivot if and only if the distance between it and the rest of selected pivots is greater than or equal to $M * \alpha$, being α a constant whose optimum values are close to 0.4 [20].

Maximun Sum of Distances (MSD): MSD is used in *LAESA* (*Linear Approximating Search Algorithm*) [18, 17]. The underlying idea is to select pivots considering that the distance between them is always the maximum. Starting with a base pivot arbitrarily selected, the distance between the objects and the selected pivot is calculated, and then the new base pivot to be selected is the one located to the maximum distance. The distances are added in a vector to calculate the next base pivot. This is an iterative process that ends when the required number of base pivots is obtained.

In general terms, Spaghettis metric data structure, SSS-Index and LAESA, could be considered as a generic bidimensional array data structure. The difference between these structures is the way of obtaining the pivots or the way in which the structure is stored.

From this initial analysis, we have extracted some conclusions that provide us some clues about what kind of metric structure could be suitable to be implemented in a GPU-based platform. Thus, in next section we introduce a structure called Generic Metric Structure as an alternative to the previously presented. The idea is to simplify the structures and the processes thinking of the GPU architecture and programming model in order to obtain the best performance.

2.2 Generic Metric Structure

Considering that we are going to work on a GPU-based hardware platform, a pivot-based Generic Metric Structure (GMS) has been considered [23]. During the construction of this metric structure, a set of pivots $p_1, ..., p_k$, which may or may not belong to the database, are selected. A GMS can be seen then as a table of distances between the pivots and all the elements of the database, i. e., each cell stores the distance $d(y_i, p_j)$, where y_i is an element of the database and p_i the pivots.

For this generic metric data structure, the searching process, given a query q and a range r, is carried out according to the following steps:

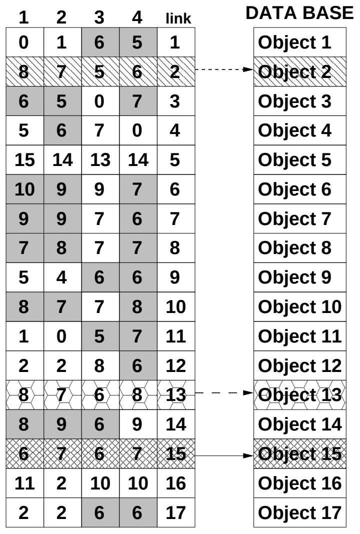
1. The distance between q and all the pivots $p_1, \ldots, p_k$ is calculated in order to obtain k intervals in the form $[a_1, b_1], ..., [a_k, b_k]$, where $a_i = d(p_i, q)$ - r and $b_i = d(p_i, q)$ + r.
2. The objects in the intersection of all intervals are considered as candidates to the query q.
3. For each candidate object y, the distance $d(q,y)$ is calculated, and if $d(q,y) \leq r$, then the object y is a solution to the query.

Details of the process are shown in Algorithm 1.

Algorithm 1 Generic Metric Index: Search Algorithm.

```
rangesearch(query $q$, range $r$)
 1: {Let $\mathbb{Y} \subseteq \mathbb{X}$ be the database}
 2: {Let $P$ be set of pivots $p_1, \ldots, p_k \in \mathbb{X}$}
 3: {Let $D$ be the table of distances associated $q$}
 4: {Let $S$ be Metric Index}
 5: for all $p_i \in P$ do
 6:     $D_i \leftarrow d(q, p_i)$
 7: end for
 8: for all $y_i \in \mathbb{Y}$ do
 9:     $discarded \leftarrow false$
10:     for all $p_j \in P$ do
11:         if $D_j - r > S_{ij} || D_j + r < S_{ij}$ then
12:             $discarded \leftarrow true$
13:             break;
14:         end if
15:     end for
16:     if !$discarded$ then
17:         if $d(y_i, q) \leq r$ then
18:             add to result
19:         end if
20:     end if
21: end for
```

Figure 2 represents a GMS built using 4 pivots. In this example, objects $2, 13, 15$ are candidates and their real distance to the query must be calculated.



1	2	3	4	link	DATA BASE
0	1	6	5	1	Object 1
8	7	5	6	2	Object 2
6	5	0	7	3	Object 3
5	6	7	0	4	Object 4
15	14	13	14	5	Object 5
10	9	9	7	6	Object 6
9	9	7	6	7	Object 7
7	8	7	7	8	Object 8
5	4	6	6	9	Object 9
8	7	7	8	10	Object 10
1	0	5	7	11	Object 11
2	2	8	6	12	Object 12
8	7	6	8	13	Object 13
8	9	6	9	14	Object 14
6	7	6	7	15	Object 15
11	2	10	10	16	Object 16
2	2	6	6	17	Object 17

Figure 2: Searching on GMS: Structure is built using 4 pivots. For a query q with distances to pivots $d(q, p_i) = \{8, 7, 4, 6\}$ and a search range $r = 2$, define the intervals $\{(6, 10), (5, 9), (2, 6), (4, 8)\}$ over which the searching is going to be carried out. The cells within the intervals are marked with dark gray. The cells hatched with lines indicate candidates (objects $2, 13, 15$).

Considering the metric data structures introduced till this moment and the algorithm described here, in next subsections we introduce the case studies as well as the experimental results and a discussion about them, in order to obtain some valuable conclusions for the GPU implementation.

2.3 Experimental Environment

As case studies, we have considered two datasets: a subset of the Spanish dictionary and a color histograms database, obtained from the Metric Spaces Library (http://www.sisap.org). The Spanish dictionary is a space of words composed of $86,061$ words and the edit distance was used. Given two words, this distance is defined as the minimum number of insertions, deletions or substitutions of characters needed to make one of the words equal to the other. For each query, a range search between 1 and 4 was considered. The second space is a set of $112,682$ color histograms (112-dimensional vectors) from an image database. Any quadratic form can be used as a distance, thus we chose Euclidean distance as the simplest meaningful alternative. The radius used was that allowing to retrieve 0.01, 0.1 and 1% from the dataset.

For both databases we create the metric data structure with 90% of the dataset randomly chosen, and reserve the rest 10% for queries. We select 32 pivots to built the generic structure because this amount of pivots gives the best results [23]. Pivots are also randomly selected. The time spent during the construction of the structure is not computed, this is considered as a preprocessing time.

This experimental framework was chosen because it is the most common environment used to evaluate the kinds of algorithms presented in this paper.

The hardware platform used is a 2 Quadcore Xeon E5530 at 2.4GHz and 48GB of main memory with 4 Nvidia Tesla C1060 240 cores at 1.3GHz and 4 GB of global memory, using CUDA SDK v3.2 [1]. The compilation has been done using gcc 4.3.4 compiler and OpenMP library.

2.4 Discussion

We have considered this variety of structures in order to determine, experimentally, if the cost in the searching process compensates the complexity of the implementation, taking into account that the decision taken here will condition the future implementation on a GPU-based platform.

The relevant features considered in this work are:

Execution time. The execution time is a key factor in order to determine the best implementation. In the literature lot of papers are found talking about evaluation of distances [20, 19], but they do not consider execution time (floating point operations and I/O operations), memory accesses, etc.

Distance evaluations. In general, the reduction on evaluation of distances has been considered as the main goal of the new structures design, and evidently, it has a direct impact on the execution time. However, the high processing capacity of current computational platforms implies that distance evaluation is not always the operation with a higher computational cost. For instance, in GPU-based platforms, sorting operation affects to the execution time more than the evaluation of distances.

In order to compare all the structures under the same conditions, the number of pivots were previously selected according to the SSS-Index criteria, varying the parameter α. The reason was that for SSS-Index the pivots cannot be stablished a priori because they are dynamically generated. In this case, the number of pivots taken into account are: 26, 44, 82, 328, 665, 1362 for the dictionary case study and 30, 35, 44, 57, 74, 119, 155, 244 for the histogram case study. Moreover, for GMS, Spaghettis and LAESA data structures 32 and 500 pivots were added for the dictionary case study and 32 pivots the for histogram case study, with the aim of having a more complete study of the behavior of the different methods. In the case of SSS-Index, it is not possible to deal with this number of pivots because there is not any α that provides them.

Figures 3 and 4 show the results for the sequential implementation related to the selected methods previously described. The figures only show the range between 32 and 500 pivots for dictionary case study and between 32 and 119 for the histogram case study. Also, Figure 3 shows the results separately for search range 1, 2, 3 and 4, for the dictionary case study. In general, the idea of considering an interval or separating by search range is due to the fact that we want to focus on the relevant results and then to make easier their interpretation.

In general, the best execution time corresponds to the Spaghettis data structure when low search range is considered, independently of the number of pivots taken into account. The reason is that the binary search process used to find the exact range for the first pivot, optimizes the rest of the searching process. GMS data structure is the second one in terms of performance. Finally, the SSS and MSD methods have the worst behavior, being the random pivot selection method the best choice for these case studies. In this point, it is important to mention that searching in metric spaces depends on the real and intrinsic dimension of the space, and then all the methods could not have the same behavior. This is the case of SSS and LAESA, where better results were expected.

As the range or radius increases, the behavior of the Spaghettis-based method is degraded because the number of discarded objects with the first pivot is lower than

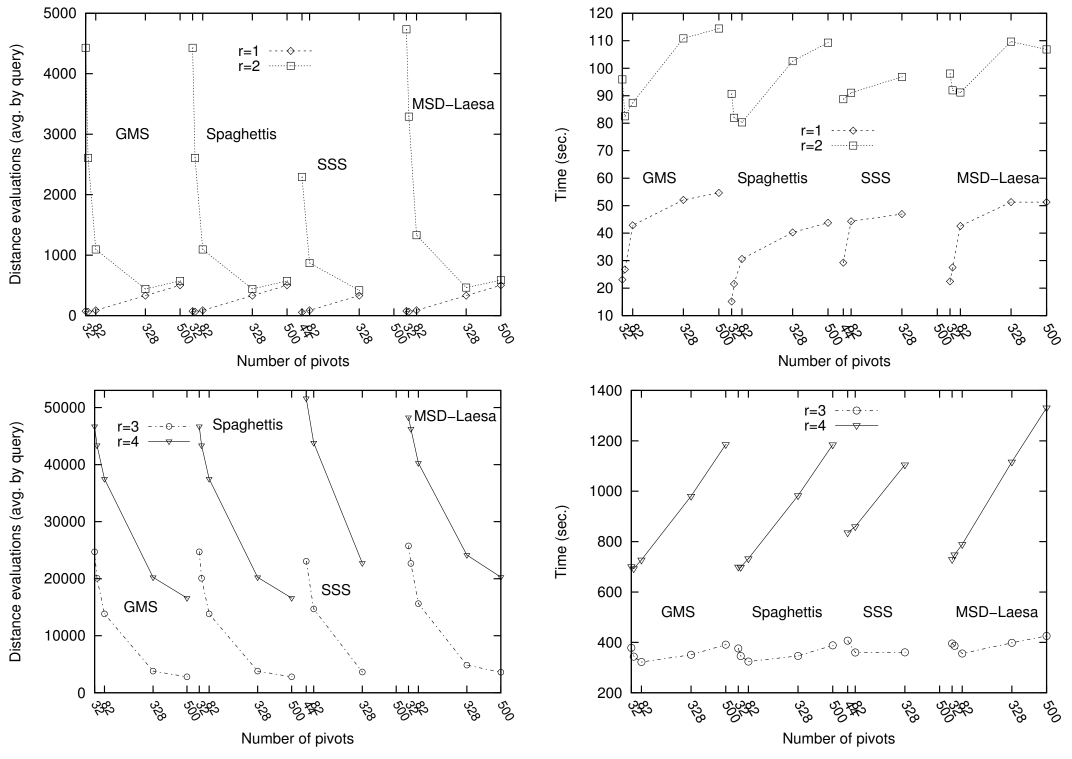
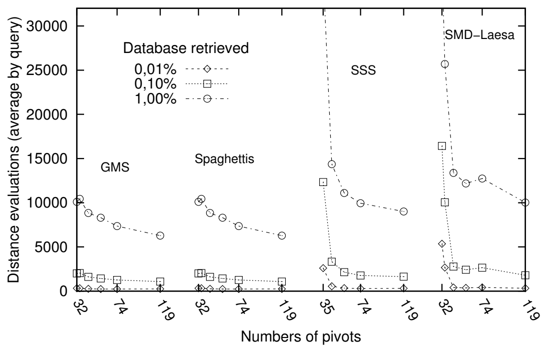


Figure 3: Calculations of average distance per query (left column) and total execution time (right column) for the Spanish dictionary case study considering the following data structures: GMS, SSS-Index, Spaghettis, MSD-Laesa.

considering low range. In this situation, the binary search becomes a problem because it increases the execution time. For instance, in the histogram case study the percentage of discarded objects is just 50%. However, in the dictionary case study the percentage of discarded objects decreases from 38.5% for a low range till 3% for upper ranges, considering just a pivot.

Tables 1 and 2 show, in detail, the execution time (in seconds) of the best cases depending on the range or on the data retrieved percentage, respectively. In these tables several modifications of the generic structure are considered. In these modifications the pivots were not selected randomly but following the pivots selection methods used by the other structures. Thus, first we randomly or using SSS get a subset of pivots from the database and then MSD is applied to get the number of pivots for the best performance case (32 or 44 depending on the range). Only modifications of the structure with a good performance are considered in the tables (e.g. “MSD *x* on SSS *y*” cases are not included



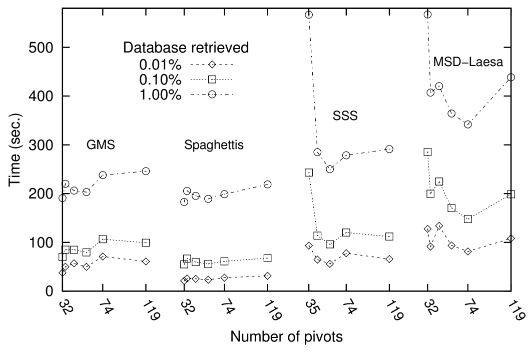


Figure 4: Calculation of average distance per query (left column) and total execution time (rigth column) for the histogram of colors case study considering the following data structures: GMS, SSS-Index, Spaghettis, MSD-Laesa.

Index	1	2	3	4
Spaghettis 32	15.45	90.66	375.80	699.58
MSD 32 on GMS 665	22.51	89.86	382.35	703.83
MSD-Laesa 32	22.46	98.05	395.62	729.82
MSD 32 on GMS 1362	23.05	90.15	380.10	710.81
GMS 32	23.08	95.92	378.41	700.94
MSD 32 on SSS 1362	24.83	96.33	398.35	738.19
MSD 32 on SSS 665	25.07	92.33	394.45	747.89
MSD 44 on GMS 1362	25.78	76.64	344.74	705.13
SSS-Index 44 ($\alpha = 0.55$)	29.25	88.75	406.80	835.76

Table 1: Execution time for the best methods for the Spanish dictionary case study (column: range; row: data structure)

in color histograms because they have a poor performance).

According to the experimental results, it is not possible to select a metric structure as the best one, because it depends on the space distribution of the database. In fact, in this context we expect that SSS-Index provides better results than GMS. Thus, two structures are candidates to be eligible as the best: Spaghettis and GMS. However from the point of view of a future GPU implementation the best one is GMS due to:

1. By using a generic structure it is not necessary to apply a binary search like Spaghettis does. Binary search operation is very expensive in a GPU-based platform in comparison with the evaluation of distances.

2. Thanks to the combination of a generic structure and MSD pivot selection, it is

Index	0.01	0.1	1.0
Spaghettis 32	21.00	54.90	182.87
GMS 32	37.77	69.89	190.74
MSD 32 on GMS 119	39.28	71.85	190.26
MSD 32 on GMS 1014	46.55	96.10	246.91
SSS-Index 57 ($\alpha = 0.6$)	55.77	95.91	249.85
MSD-Laesa 35	91.33	199.75	406.99

Table 2: Execution time for the best methods for the color histograms case study (column: data retrieved percentage; row: data structure)

possible to reduce the number of pivots till satisfying the memory constraints inherent to the GPU-based platforms, obtaining at the same time a slight improvement in execution time. However, this benefit depends on the space distribution. For instance, good results were obtained in words space by applying random selection of pivots and after that applying MSD, but no benefits were obtained in color histograms (see Tables 1 and 2).

To sum up, using the generic structure we will take benefits in terms of execution time and, in addition, the code is more simple.

Having that in mind, next sections present different Heterogeneous GPU-based versions of the similarity search algorithm.

3 Range Search on Heterogeneous Platforms

In this section we describe the proposed algorithm and discuss specific details related to the different considered implementations on the selected hardware platform. Differences between implementations are given by the type of computing elements that are used in each of the implementations, i.e., CPU cores and/or GPUs. We have considered three cases: one CPU core and one GPU, several CPU cores, and several CPU cores and several GPUs.

There are some features in the implementations that are common to all:

- Data structures. The main data structures are used in all implementations, i.e., dataset, queries, GMS, and pivots. Maybe, each implementation can use different auxiliary structures, and it is also possible that some of these structures are handled in a different way and from different levels in the system memory hierarchy.

- Processing. Obviously, all the essential computing operations are performed in all the implementations. However, to fully solve all the queries some extra actions must be considered in some implementations, in particular those using GPU devices. For instance, in these cases data structures must be transferred from host/CPU main memory to device/GPU global memory.
- Pre-processing. Generic Metric Structure is built in a preprocessing stage and is loaded during the program execution. That is, in this stage the number of pivots are defined, the pivots are randomly selected and the structure is generated.

Particular details of each implementation are included below.

3.1 Multicore implementation

In order to use the capability of our CPU with 8 cores, to carry out a performance evaluation on this platform and to compare the results between platforms (GPU and multi-core CPU), we have implemented a multicore-based version of the same algorithm.

We have implemented the three parts described in Section 2.2 which are the most expensive computationally as we have explained above. Mainly, this implementation consists in distributing the queries to all available cores using OpenMP pragmas. In particular, we have used the `#pragma omp parallel for` directive in *OpenMP*.

3.2 GPU implementation

Unlike previous works [22], where 2 and 3 kernels were considered, in this case all the operations solving a given query are included into one kernel. The queries are distributed to all available GPUs, and each GPU solves the queries one by one, i.e. there exists a loop such that each iteration completely solves a query by means a kernel call.

Before calling the kernel to be executed in the GPU, main data structures are transferred from host main memory to GPU global memory: dataset, GMS, and pivots. Some auxiliary structures are also located into global memory, and will be used for storing temporal and final results.

After that, the kernel code corresponds to the following main actions:

- The query q to be solved is directly transferred to the GPU shared memory by using the CUDA parameter-passing mechanism.
- Then, in order to solve the query, as many threads as the number of objects in the dataset are launched to execution. A few of them compute the distance between q and the pivots p_i, and the resulting distances $d(q, p_i)$ are maintained into the shared memory in each multiprocessor because they will be used by all the threads (step 1 of basic algorithm).
- In the next step, all the threads determine whether the elements of the database are or not candidates for the query q. Each thread checks out only one element (Steps 2 and 3 of basic algorithm).
- Finally, for each candidate element the corresponding thread determines whether that element is a valid solution for the query q.

Finally, results are transferred from GPU global memory to CPU main memory.

Figure 5 shows the algorithm operation inside the GPU, in particular the steps 2 and 3 of the basic algorithm described in Section 2.2. The data distribution into shared memory and the accesses carried out by the threads to the data and structures can be observed.

3.2.1 Memory access latency

When hundreds or thousands of threads are simultaneously running, there is enormous pressure on the memory system, which can increase in a significant amount the average memory latency, and as a consequence decrease the performance. In order to reduce that latency, some actions can be taken into consideration, e.g. to exploit the lower shared

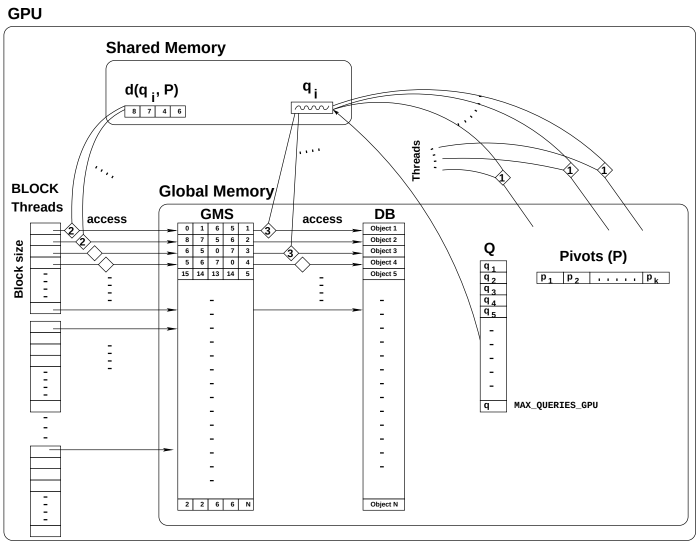


Figure 5: Inside of a GPU. Steps 2 and 3 from the basic algorithm.

memory latency or promote coalesced access to global memory[1].

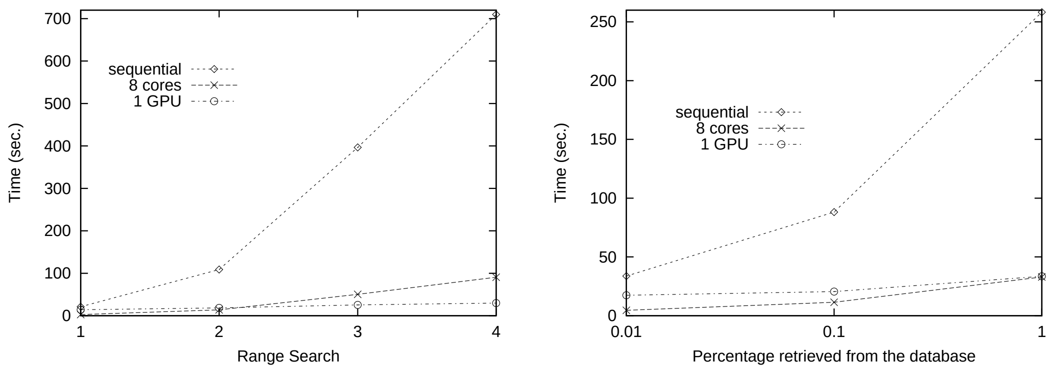
Our kernel places the most used data into shared memory, and transforms the input data structures for achieving coalesced memory access. Since each thread needs to access an entire row of the GMS structure, and all the threads attempt to simultaneously access their respective rows, the GMS structure is transformed applying the transposed operation previously to be transferred from host main memory to GPU global memory. In this way, when threads in a given warp access to GMS structure, they will find the data in consecutive positions, allowing coalesced accesses to global memory.

3.3 Results and Discussion

In order to get a broader view on the use of a multicore and a GPU-based platform, Figure 6 shows absolute execution times for the following implementations: sequential, multicore (with 8 cores), and GPU using shared memory.

As expected, graphics in Figure 6 show a considerable decrease in the execution time when comparing the sequential version with both the multicore and the GPU versions. For the color histograms space, the behavior of the different versions shows the same

[1]Global memory accesses by threads of a warp (of a half warp for devices of compute capability 1.x) are coalesced by the device into as few as one transaction when certain access requirements are met.



(a) Execution time for the Spanish dictionary case study. (b) Execution time for the color histograms case study.

Figure 6: Execution time for sequential, multicore and GPU implementation.

differences when the search radius increases. In this case, the version with 8 cores is always better than the one with one GPU. For the space of words, such regularity in the behavior of the structures is lost. For ranges greater than 2 the GPU platform versions have a better performance than those with 8 cores. This difference is basically due to the characteristics of the space of words.

The number of elements we can discard may depend on the search range. For example, with 16 pivots and a range $r = 1$ in the space of words, more than 99.9% of elements of the database can be discarded; however, with $r = 4$, this percentage decreases to 39.7%. Nevertheless, in the color histograms space these values range from 97% to 90.1% for the minor and major radius respectively, i. e., the structure has a more stable behaviour in this space. The reason of this different behaviour is the nature of the distances: in the first case we use a discrete distance while in the second case the distance is continuous. We can conclude that, in order to discard the maximum amount of objects in the space of words, it is important to use all the pivots for high ranges. Finally, we can say that for the space of words the number of threads available to resolve a query is very important, therefore, for high ranges the performance of the GPU implementation is better than the 8 cores implementation.

4 Heterogeneous implementations of the Similarity Search Algorithm

Nowadays, GPU/MultiGPU devices are widely used to get reasonable execution time at a low price. The GPU/MultiGPU are managed by the CPU/core/multicore. This is why these systems are called heterogeneous systems. However, in most cases the CPU/core/multicore are idle when the GPU/MultiGPU devices are processing. As a consequence, the theoretical peak performance of the underlying architecture decreases, and of course, the term heterogeneous platform becomes more a conventional term than a real term.

In this paper the authors have developed two different heterogeneous implementations considering two different scenarios. The first one takes into account that the queries are completely processed in a unique kernel. Under this scenario more objects of the database can be allocated to the global memory of the GPU. Evidently, the consequence is that the time spent in data transfer is penalized and more calls to the unique kernel are needed. The second scenario corresponds to a block-oriented implementation transferring a chunk of queries at a time and processing it in the unique kernel, reducing the time spent in data transfer and calls to the kernel. However, less objects of the database can be allocated to the global memory of the device. Moreover, a bigger data structure is required to store the results of the queries.

In general, a query is entirely resolved by a single device. The scheme used distributes queries on all the cores, including the GPUs core administrators.

The cores share the structures and data from the host's RAM. This information is also replicated to every GPU devices. The iterations are gotten out using the `#pragma omp parallel for` directive in OpenMP. The number of iterations assigned to each core and how queries are distributed over the set of cores depend on the type of scheduling defined in the `#pragma` directive. Four scheduling schemes can be considered for this `#pragma`.

1. Dynamic: Some of the iterations are allocated to a smaller number of threads.

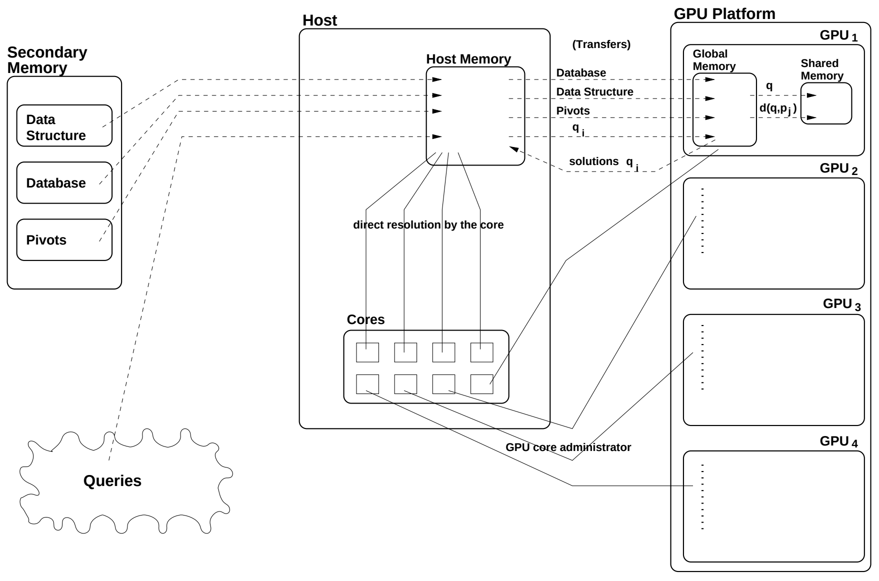


Figure 7: Heterogeneous System Scheme.

Once a particular thread finishes its allocated iteration, it gets another one from the iterations that are left. The parameter *chunk* defines the number of contiguous iterations that are allocated to a thread at a time.

2. Guided: Large chunks of contiguous iterations are allocated to each thread dynamically. But the chunk is not constant, it decreases exponentially with each successive allocation to a minimum size specified in the parameter chunk.

3. Runtime: Scheduling decision is deferred until runtime by variable `OMP_SCHEDULE`.

4. Static: In this case, all the threads are allocated iterations before they execute the loop. The iterations are divided among threads equally by default. However, it is possible to allocate a fixed number of contiguous iterations to a particular thread by using the parameter chunk.

In general, when the different iterations in the loop may take different time to execute it is more convenient to use `dynamic` scheduling. Thus, this is the scheduling scheme used in this work.

Figure 7 shows the operation diagram of the heterogeneous system, including GPU core administrator. In the scheme it is possible to see the data transfer. In this diagram there are 8 cores and 4 GPUs.

4.1 Heterogeneous implementation: one query - one kernel

In this section we introduce an implementation that considers the comments in previous section, related to the scheduling scheme, but where the kernel processes one query at a time. The reason to consider this scenario is that under these circumstances more objects of the database can be allocated to the global memory of the GPU. However, the time spent in data transfer increases and more calls to the unique kernel are performed. On the other hand, this implementation allows us to obtain a first approach to the behavior of the GMS structure for this heterogeneous scheme applying different chunk size.

4.1.1 Execution time

Figures 8 and 9 show the results obtained when using a multicore implementation against an implementation following the heterogeneous scheme using `dynamic` scheduling. In the graphs, the X axis represents the percentage of iterations assigned to each core (equivalent to the number of queries), except for 1q that means 1 query. For a better visualization of the results, search ranges have been separated into individual graphics. On the other hand, in the legend YZ, Y means number of cores and Z number of GPUs. Then, 80 means 8 cores and 0 GPUs, meanwhile 81 means 8 cores, but one of them manages 1 GPU.

In these figures, it can be observed that the implementation following the heterogeneous scheme has better performance in terms of execution time than the multicore version, except for $12,5\%$ where all resources carried out the same number of iterations. In this case, the GPU is underused.

The advantage of using all the available resources in 81 is shown in Figure 10 where we can see that the heterogeneous version gets the best performance.

In particular, we can observe that the performance obtained by 81 evolves in parallel to the performance obtained by using just 1 GPU. The gap between them is given by the constant speed-up of using 8 cores.

Figures 11 and 12 show the results for the heterogeneous system using from 1 to 4 GPUs. In general, the behavior of 84 is the best for all cases, especially for low number

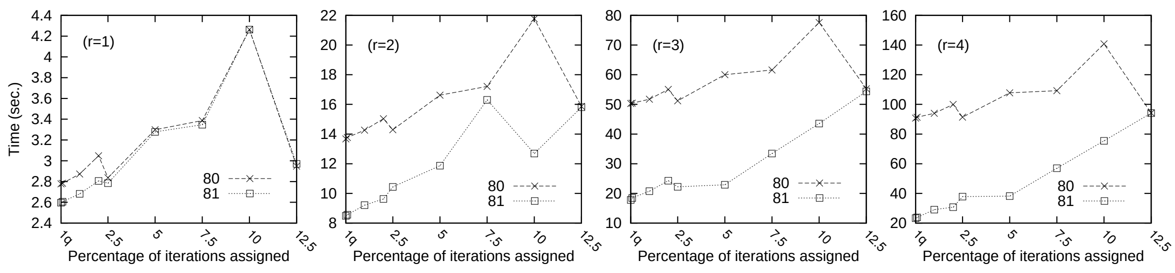


Figure 8: Execution time depending on the chunk size when using dynamic scheduling (space of words).

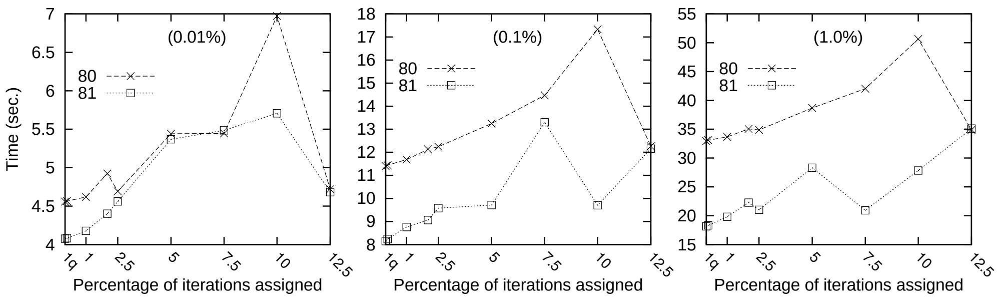
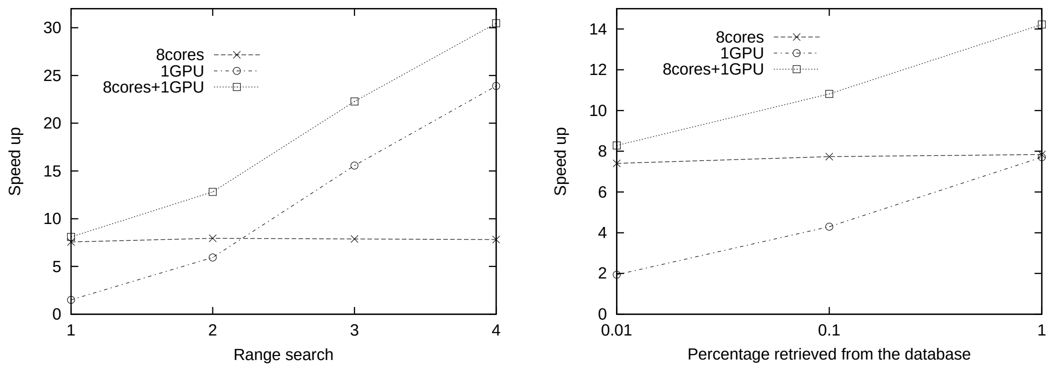


Figure 9: Execution time depending on the chunk size when using dynamic scheduling (color histrograms).



(a) Spanish dictionary. (b) Color histograms.

Figure 10: Speed-ups for heterogeneous system.

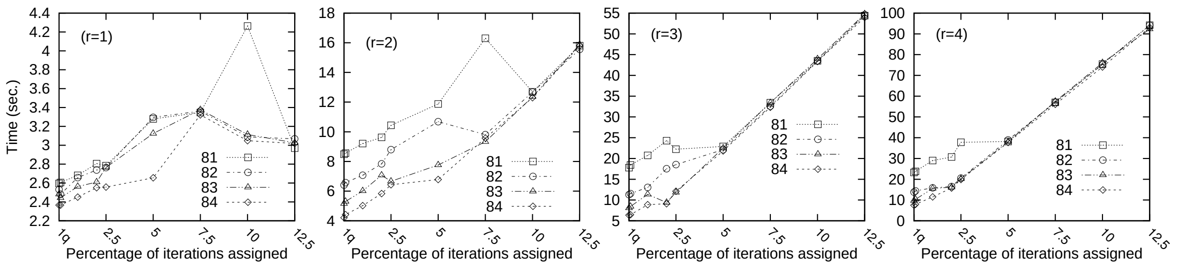


Figure 11: Adding GPUs to the heterogeneous scheme changing cores by GPUs. Spanish dictionary case study.

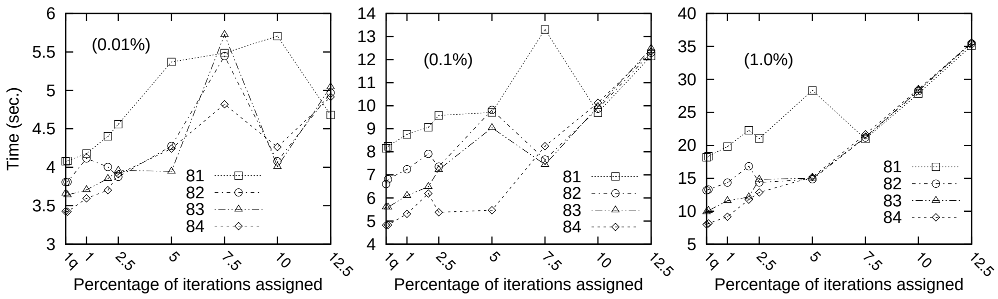


Figure 12: Adding GPUs to the heterogeneous scheme changing cores by GPUs. Color histogram case study.

of queries. When the number of queries increases the cores receive more queries and, due to the fact that they are slower than GPUs, the global behavior is subordinated to them.

Another aspect to consider is the chunk size. Unexpectedly, one could think that having a chunk size that balances the computational load will be the best option. However, according to our results the best chunk size is just one query. This can be explained considering that, in spite the chunk size used by the dynamic scheduling through the OpenMP `#pragma#` directive, actually the system resolves just one query at a time, i.e., OpenMP internally makes a for loop of chunk size and sends to the devices only one query at a time. Moreover, the time spent on transferring queries is independent of the chunk size.

In the next section we are going to analyze in detail the execution time distribution, the number of queries assigned to every device and the relationship between execution time and number of queries assigned.

4.1.2 Distribution of execution time and queries

Firstly, we show the behaviour of the heterogeneous system using 8 cores and one of them managing one GPU (81).

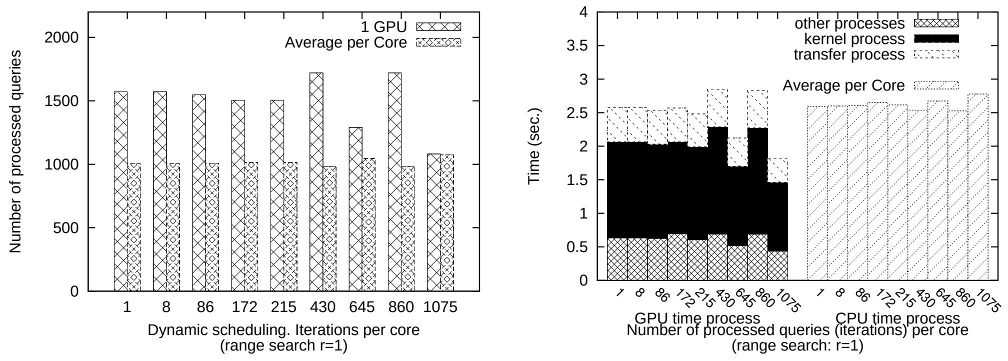
Left column in Figures 13 and 14 shows the amount of queries resolved by the GPU or the cores. In the case of cores, due to the fact that there are 7 free cores, we have considered the average number of queries processed. The right column shows the execution time used by the different devices (GPU or cores) splitted in three parts:

1. Transfer process: it represents the time spent in transferring the queries to the GPU and retrieving the solutions of the query.
2. Kernel process: it represents the calculations of the solutions for each query by the GPU.
3. Other processes: it represents other needed operations before transferring or calculating. For instance, some operations carried out for the core that manages the GPU.

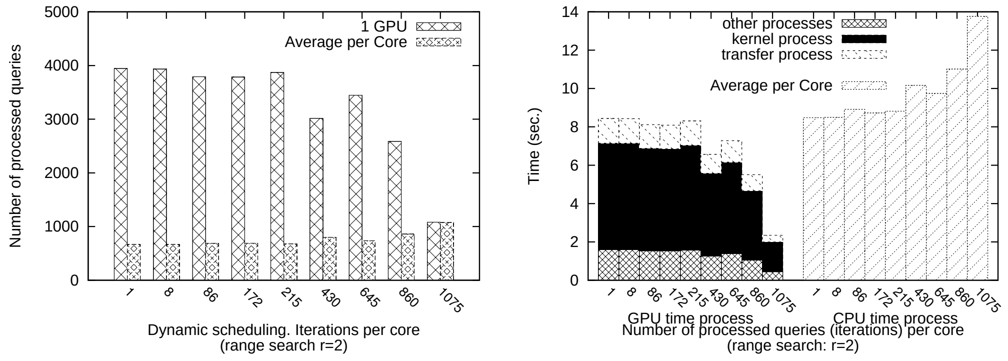
In the case of cores, there is not transfer process or additional processes, then all the time represents calculation of solutions for the queries.

In both cases, as the computational load increases (increasing the range for the dictionary case study or increasing the retrieval percentage for the color histogram) the heterogeneous system takes benefits of the power of the GPU. In those cases, it is necessary to evaluate more objects in the database, and then the GPU is able to complete more queries than the cores. For instance, in the last graph in Figure 14, each core processes almost 1000 queries meanwhile the GPU processes almost 6000.

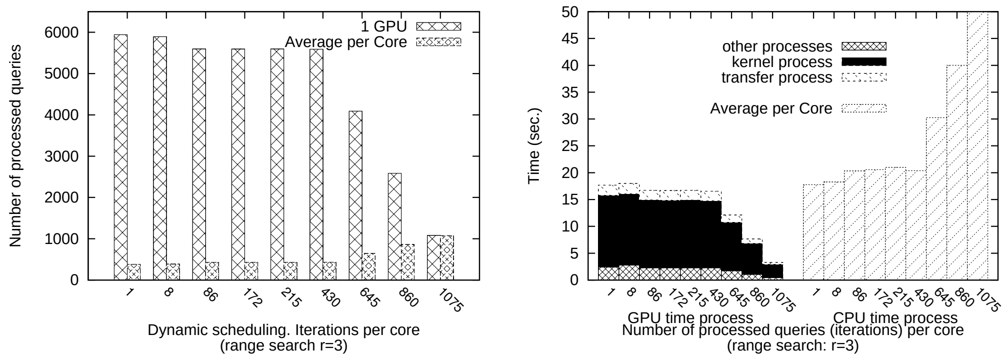
With respect to the execution time, there are two parts almost constant. One of them is related to other processes. This time is difficult to avoid because the core has to manage the GPU. The other one is the transfer process. As we explained before, the transfer time is independent of the chunk size, due to the way in which OpenMP sends the queries to the GPU (in the case of cores there are not transferences). However, this time could be reduced if we deceive OpenMP and really we send a complete chunk to



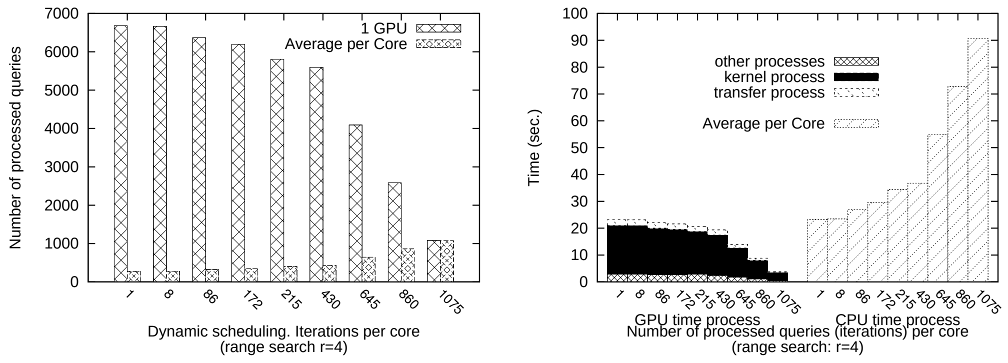
(a) $r = 1$



(b) $r = 2$

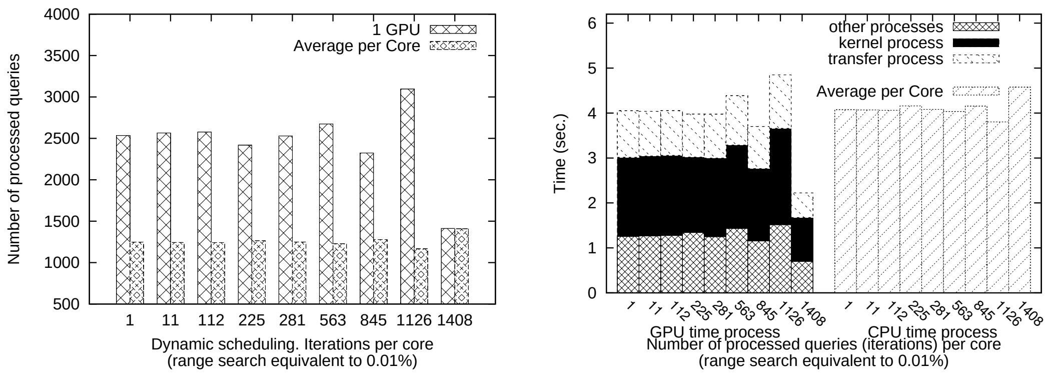


(c) $r = 3$

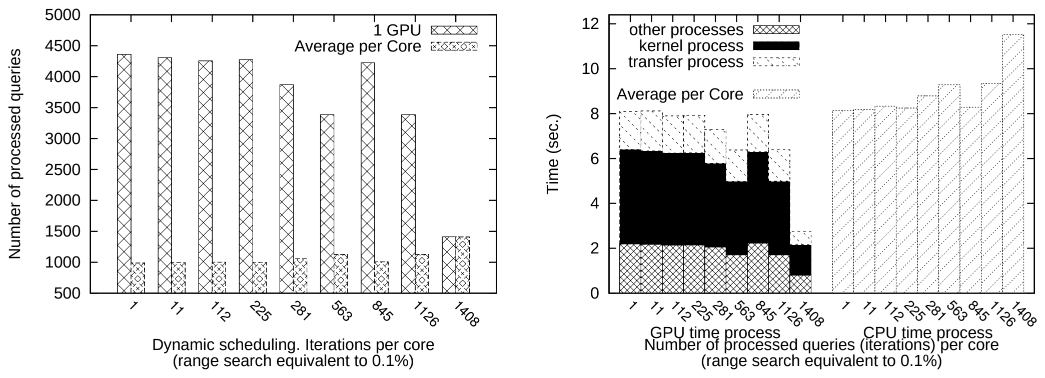


(d) $r = 4$

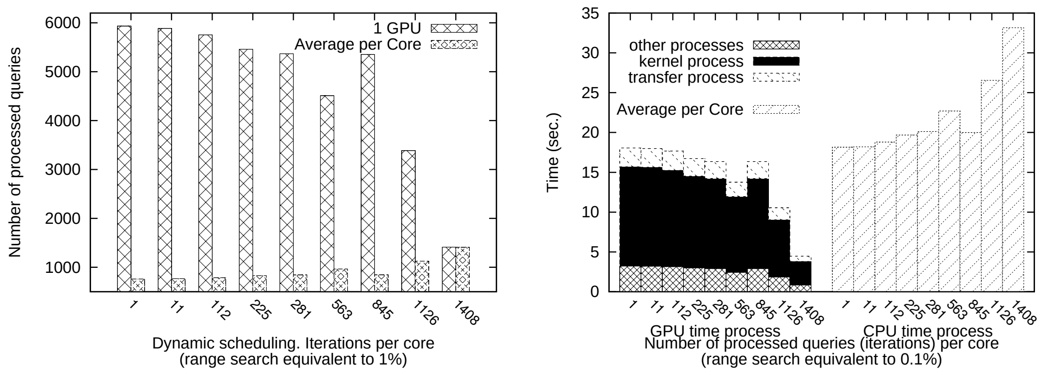
Figure 13: Distribution of queries and execution time obtained by the heterogeneous system for the Spanish dictionary case study, considering 8 cores and 1 GPU. Left column represents query distribution and right column represents execution time distribution.



(a) 0,01%



(b) 0,1%



(c) 1,0%

Figure 14: Distribution of queries and execution time obtained by the heterogeneous system for the color histogram case study, considering 8 cores and 1 GPU. Left column represents query distribution and right column represents execution time distribution.

the GPU.

Regarding the execution time of the cores, as the number of queries assigned to each core increases, the system forces the cores to spend more processing time and reduces the global system performance because it can not take benefits from the use of the GPU.

If we consider the complete system, the behavior is exactly the same we have previously commented, although the queries are better gotten out among the GPUs. This can be seen in Figures 15 and 16.

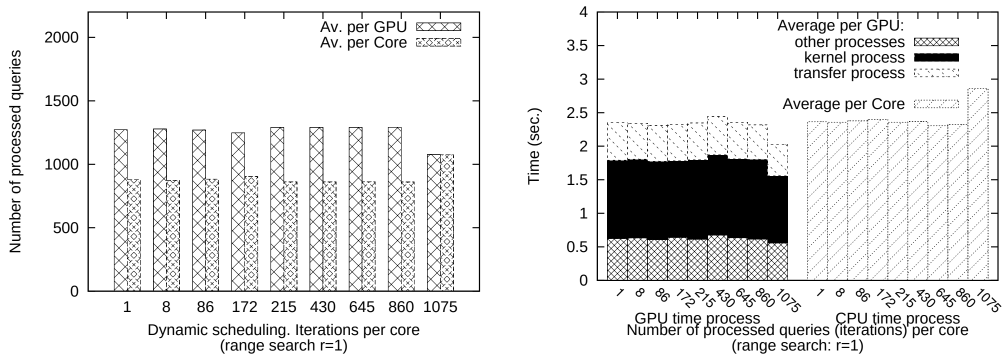
4.1.3 Scaling the number of queries

In order to study the robustness of the system, that is, if it behaves as we expect, we are going to scale the number of queries for both databases. To words space, we processed 40 thousand new queries selected from a sample of the Chilean Web which was taken from the TODOCL search engine. As vectors space we using a database with the same features as the previous sections. This is a synthetic database with 1 million vectors. This database represents a set of color histograms with 112 dimensions. We build the data structure with 100 thousand objects and the rest 900 thousand as queries.

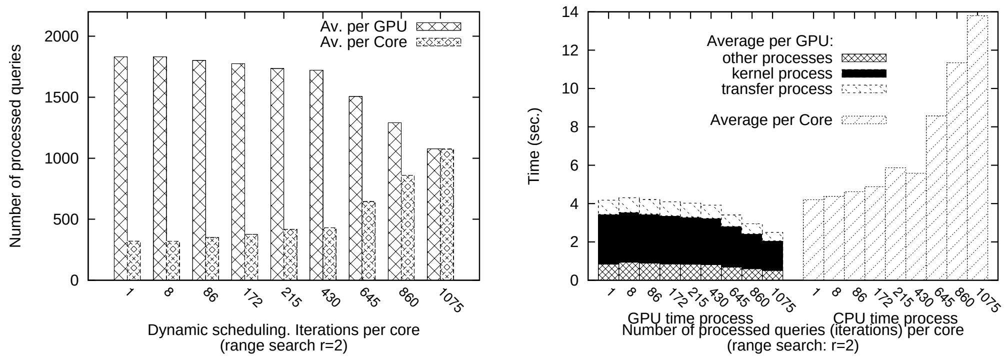
A priori, if the database remains constant, the execution time increases linearly as the number of queries also increases linearly, that is, the global system is scalable.

Then, Figure 17 shows the results for the words space when increasing the queries from 10 thousand to 40 thousand considering the whole system (8 cores and 4 GPUs) and the smaller and bigger range (1 and 4 respectively). The legend 1, 10, 100 and 1000 represents the chunk size. Figure 18 shows the results for the color histogram database when increasing the queries from 100 thousand to 900 thousand considering the whole system and the smaller and bigger range (equivalent to 0.01 and 1 percentage of retrieved information respectively). The legend 1, 10, 100, 1000 and 10000 represents the chunk size.

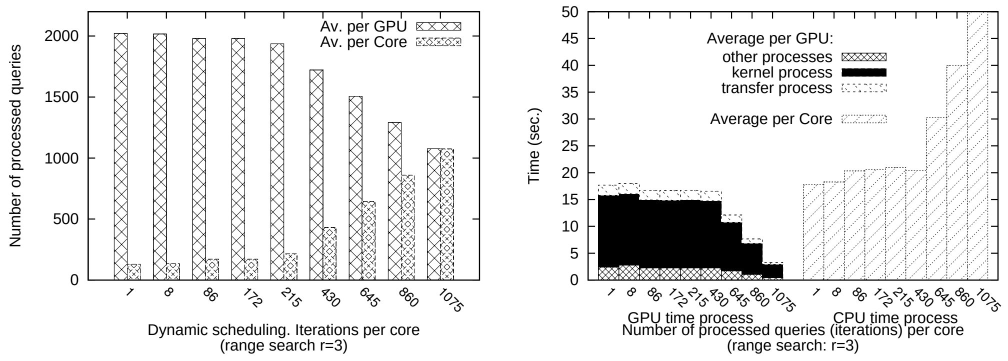
For the color histogram database, we can observe the linear behavior of the system meaning that the implementation is scalable. Notice that there is a slight improvement by using a chunk of size 1 query, as we commented in previous sections due to the opera-



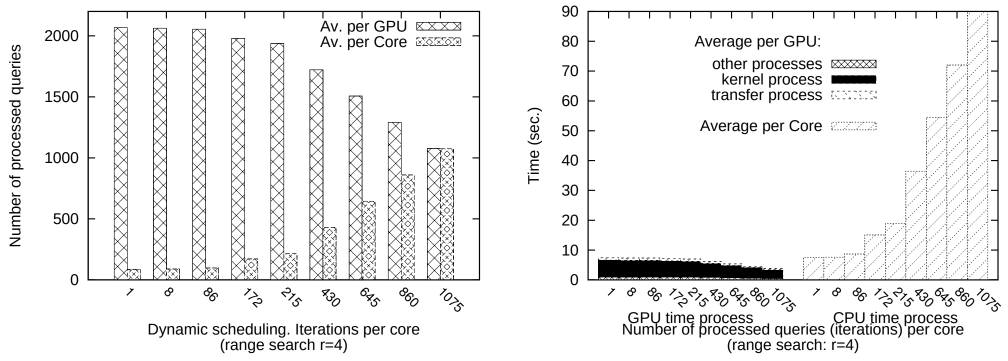
(a) $r = 1$



(b) $r = 2$

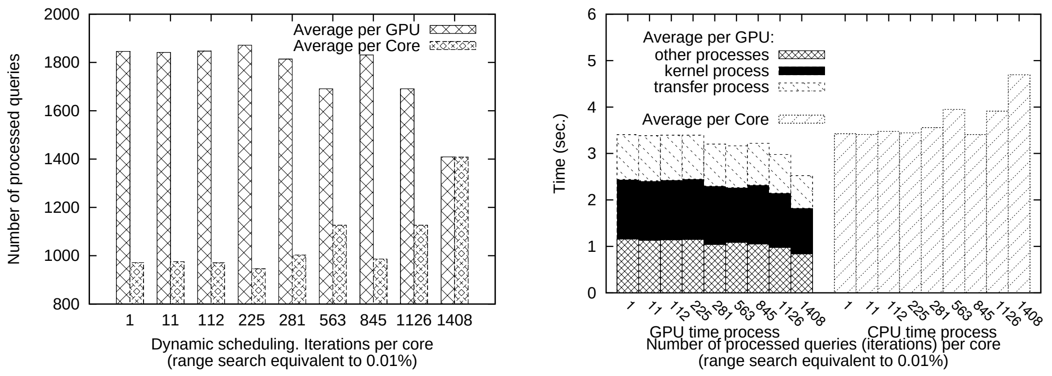


(c) $r = 3$

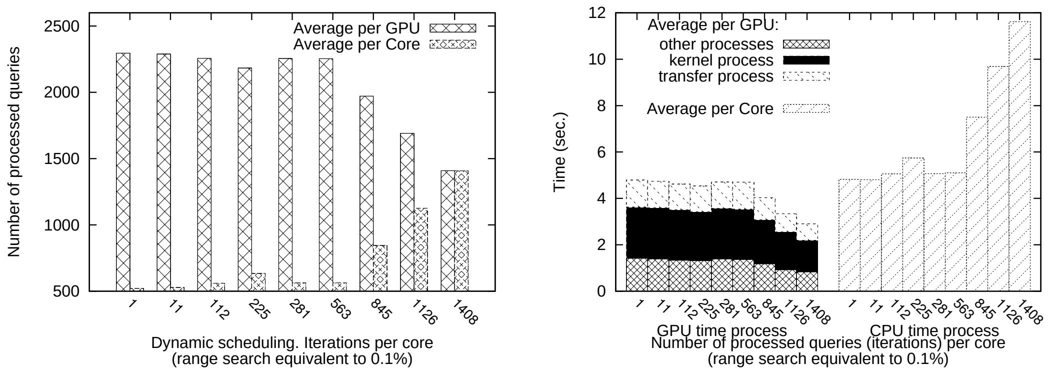


(d) $r = 4$

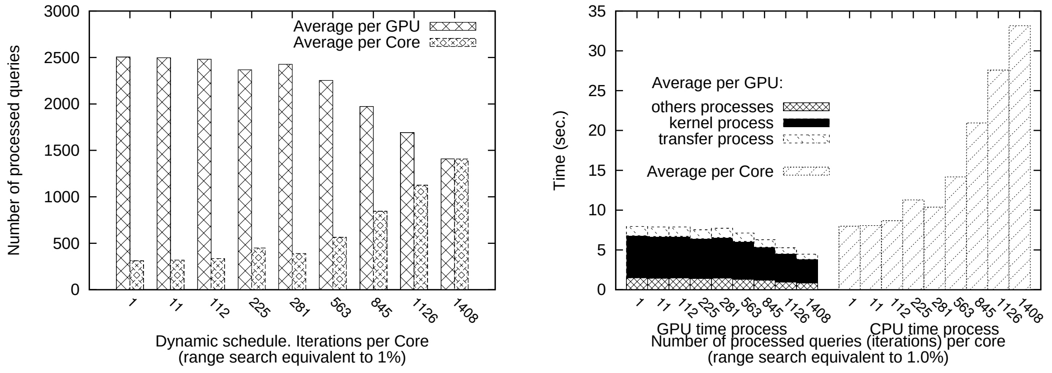
Figure 15: Distribution of queries and execution time obtained by the heterogeneous system for the Spanish Dictionary case study, considering 8 cores and 4 GPU. Left column represents query distribution and right column represents execution time distribution.



(a) 0,01%



(b) 0,1%



(c) 1,0%

Figure 16: Distribution of queries and execution time obtained by the heterogeneous system for the color histogram case study, considering 8 cores and 4 GPU. Left column represents query distribution and right column represents execution time distribution.

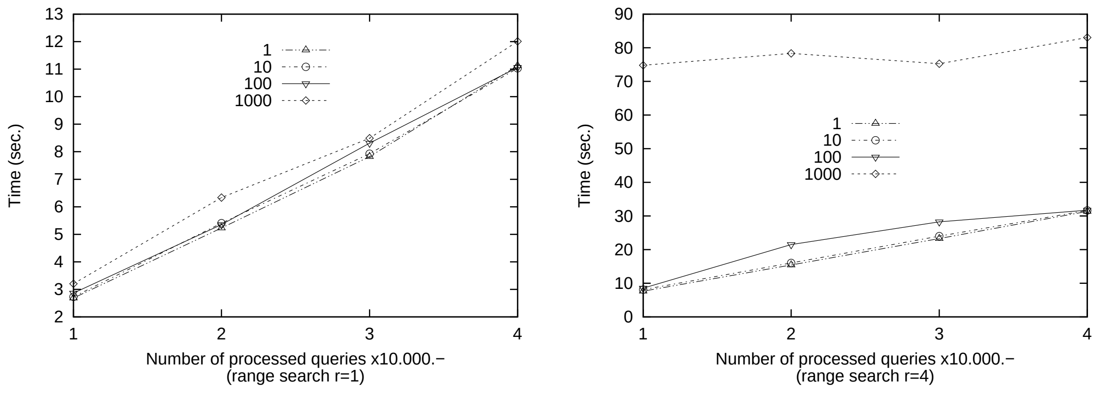


Figure 17: System scalability. Execution time vs. chunk size when the number of queries is varied from 10 thousand to 40 thousand considering 1 and 4 as range search in a system with 8 cores and 4 GPUs.

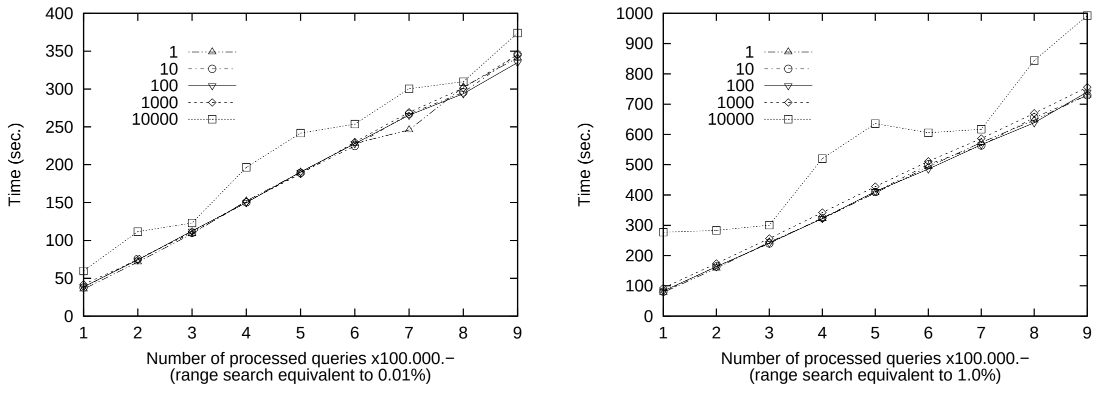


Figure 18: System scalability. Execution time vs. chunk size when the number of queries is varied from 100 thousand to 900 thousand considering 0.01% and 1% of percentage of retrieved information in a system with 8 cores and 4 GPUs.

tion way of OpenMP. However, considering such a huge amount of data, the chunk sizes 10, 100 and also 1000 provide good results, being between almost 0.01% of the number of queries (chunk size 10 over 900.000 queries) and 1% (chunk size 1000 over 100.000 queries), i.e, relative small chunks. However, as this percentage increases (chunk size 10.000 over 100.000 queries) the behavior is worst.

For the words space the behaviour is similar. Nevertheless, we can see that for a chunk size of 1000 and range search 4 the performance is very poor. In this case, independently of the number of queries (from 10 to 40 thousand), each core resolves one chunk (1000 queries) while the GPU resolves the remaining queries during the same time (6 to 36 thousand). The reason of this behaviour is the use of a relative big chunk size (10% to 2.5%) that gives more work to the cores and leaves the GPUs idle (see Figure 15(d)).

4.1.4 Discussion

Along this section, a heterogeneous implementation has been introduced where the queries are simultaneously processed in the multicore and the GPUs.

The first thing to remark is that the heterogeneous implementation, which exploits all the underlying resources in the system, improves the execution time with respect using only multicores or GPUs. The heterogeneous implementation tested using a basic configuration (8 cores and 1 GPU) is up to 31 times faster than the sequential implementation, up to 4 times faster than the multicore implementation and up to 1.3 times faster than the GPU implementation for the Spanish Dictionary case study.

This computational power has been increased using up to 8 cores and 4 GPUs, reaching from 9 to 95 times faster than the sequential implementation for the Spanish Dictionary case study, and from 10 to 32 times faster for the color histograms.

Regarding the queries distribution, the more computational complexity the more benefits from the use of the GPUs because they can process more quantity of queries and then the cores do not act as a bottleneck.

With respect to the execution time distribution, notice that the time spent in auxiliary processes and in transferring data remain almost constant, meanwhile the kernel process increases as the complexity of the problem also increases. This situation is quite good because it indicates that most time is spent in processing. However, a deeper analysis leads to think that it is possible to improve the performance if the time spent in transfer process is reduced (other processes can not be avoided because they represent needed operations carried out by the core to manage the GPU). The key point is answering why the best performance is obtained by processing one query at a time. As we commented, this is due to the fact that despite we indicate to OpenMP to process a chunk of queries, internally OpenMP interprets to process a set of queries one by one in a loop. Then, OpenMP sends one query at a time to the GPU which results in increasing the time spent transferring the queries. Thus, this algorithm could be improved if we deceive the OpenMP internal operation way and we send to the GPU a real chunk of queries.

Finally, as we expected, the algorithm is scalable.

4.2 Heterogeneous implementation considering a batch of queries - one kernel

According to the final comment of the last section, in this section we introduce a block-oriented implementation in order to reduce the number of calls to the kernel (saving time) and, of course, reduce the number of transferences (saving more time).

To carry out this block-oriented implementation, different changes have to be introduced in the code.

Firstly, we have to change the code which calls the kernel because, as we said in Section 4.1.2, it is necessary to deceive OpenMP in order to send a complete chunk of queries to the kernel instead of one query at a time.

The key point is that, the loop variable does not vary for all the queries. In this case, it varies till the number of blocks ($NUM\_BLOCKS = \frac{NUM\_QUERIES}{MQ}$).

Notice, that before calling the kernel, a complete block of queries has been allocated to the global memory of the GPU.

Regarding the changes carried out in the proposed kernel described in Section 3.2, we have the following:

1. To allocate to global memory of the GPU the needed structures containing the solutions of processing a block of MQ queries, being MQ the size of the block.
2. After that, the MQ queries are transferred to the global memory of the GPU during the resolution stage.
3. Really, the kernel is the same that was presented in Section 3.2, but in this case instead of processing just a query, it iterates to solve the complete block.
4. Finally, solutions of the MQ queries are transferred to the host memory.

A special case is when the last block has less than MQ queries. In this case this block is resolved query by query using the whole system (as version 1).

4.2.1 Execution time

In Figures 19, 20 and 21 the results in terms of execution time for the Spanish Dictionary are presented. Each graph corresponds to a range search ($r = 1, 2, 3, 4$), and the same notation as in Figures 8 and 9 is used for indicating the number of cores and GPUs. Axis X corresponds to the percentage of queries (except for $1q$ that means 1 query). In Y axis, the execution time in seconds is represented.

In Figure 19 we can observe that when the complexity of the problem increases, the size of the block has a negative influence because no improvements are obtained considering more GPUs in the system. This occurs, especially for higher block sizes, because the cores become the bottleneck of the system. When the block size is small, the GPUs have the capacity of solving more queries and then the behavior of the system is better (see Figure 25).

Figure 20 shows the result of the implementation presented in Section 4.1 (v1 in legend) with respect to the one here presented (v2 in legend). Obviously, the system improves the execution time, especially when considering a small block size. Indeed, if we make a zoom (see Figure 21) we can observe that the improvement reached with

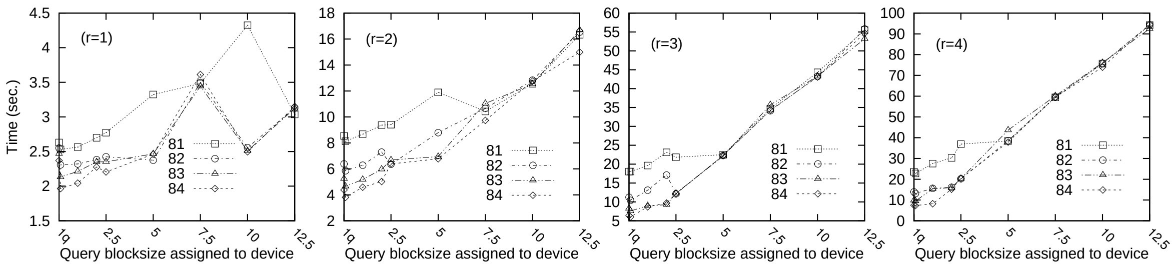


Figure 19: Adding GPUs to the second version for the heterogeneous scheme, changing cores by GPUs. Spanish dictionary case study.

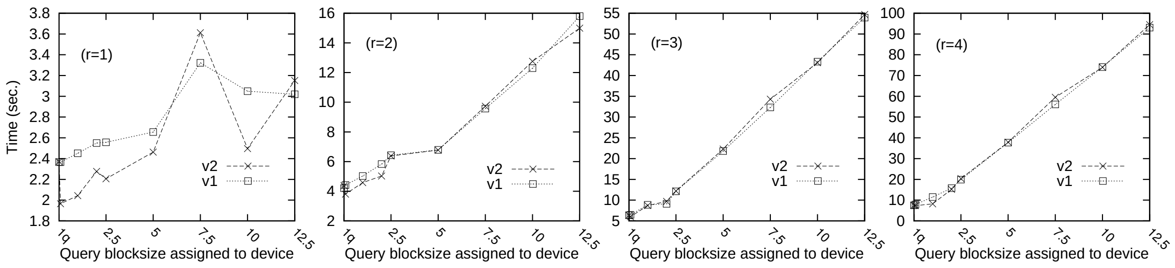


Figure 20: Comparing results for version v1 and v2. Spanish dictionary case study.

the new implementation is up to 17% in the best case, decreasing as the range increases until 1.5% (block size in X axis, not percentage). Another important result can be seen in Figure 21 is that the best performance is not obtained for 1 query, but for a small block size (8), that is around 0.1% of the total queries.

In Figures 22, 23 and 24 the results in terms of execution time for the color histogram are presented. Each graph corresponds to a percentage of retrieved information (0.01%, 0.1%, 1%). Axis X corresponds to percentage of queries (except for 1*q* that means 1 query). In Y axis, the execution time in seconds is represented. These fig-

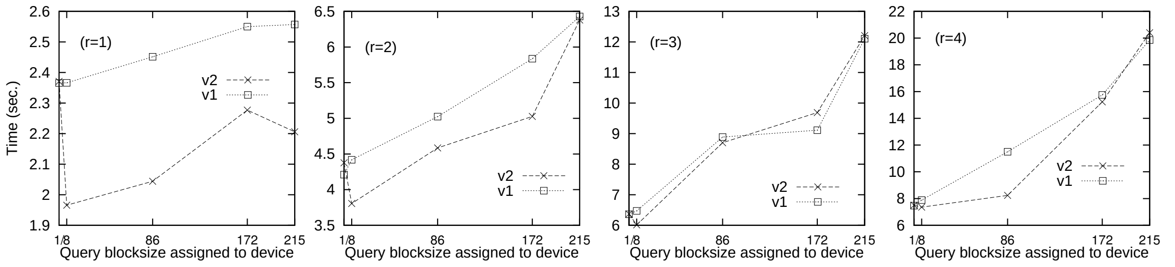


Figure 21: Details for small blocksizes, using real blocksize: 1 - 1q, 8 - 0.1%, 86 - 1%, 172 - 2%, 215 - 2.5%. Spanish dictionary case study.

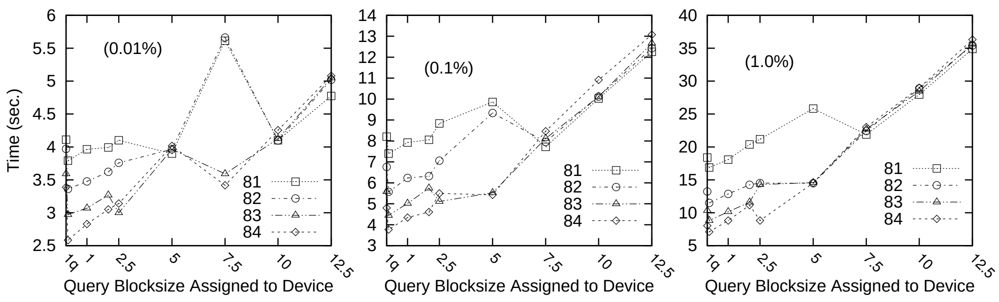


Figure 22: Adding GPUs to the second version for the heterogeneous scheme, changing cores by GPUs. Color histogram case study.

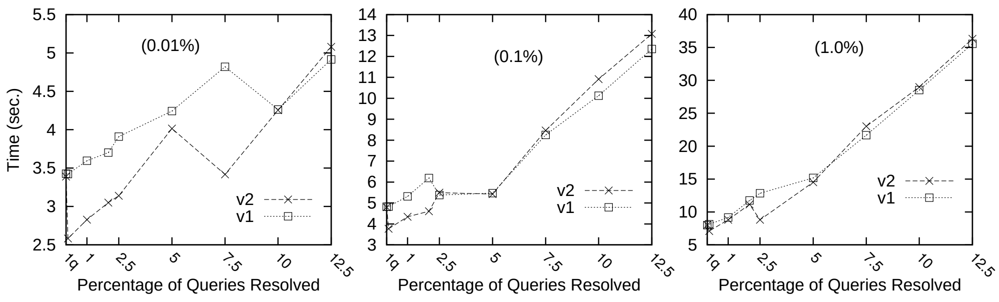


Figure 23: Comparing results for version v1 and v2. Color histogram case study.

ures endorse the results previously obtained for the Spanish Dictionary case study. The improvement reached with the new implementation is up to 25% in the best case, decreasing as the percentage of retrieved information increases until 11%. In this case the best performance is obtained for a block size of 11, that is around 0.1% of the total queries.

In both cases when the range increases the improvement of the execution time decreases. This behaviour was expected because the benefits obtained in version 2 are due to less time spent in transferences and kernel calls, and this time remains more or less constant, leading to less influence on the overall result.

4.2.2 Distribution of execution time and queries

Figures 25 and 26 show the queries distribution among the devices (left column) and the execution time distribution (right column) for a system of 8 cores and 4 GPUs. Notice that for a small block size, the GPUs carried out an important percentage of the queries

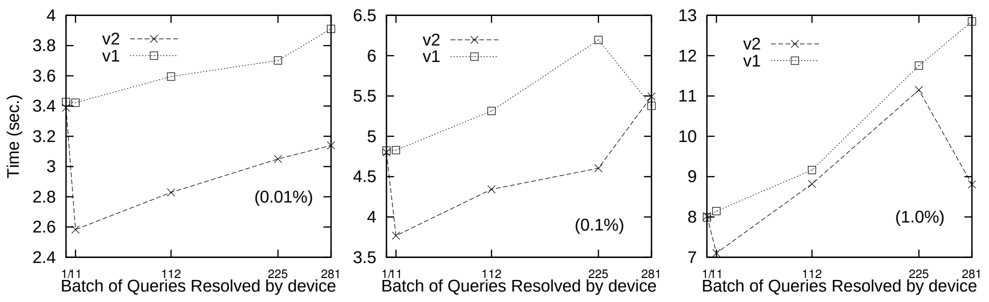


Figure 24: Details for small blocksizes, using real blocksize: 1 - 1q, 11 - 0.1%, 112 - 1%, 225 - 2%, 281 - 2.5%. case Color histogram study.

(see bottom of both figures).

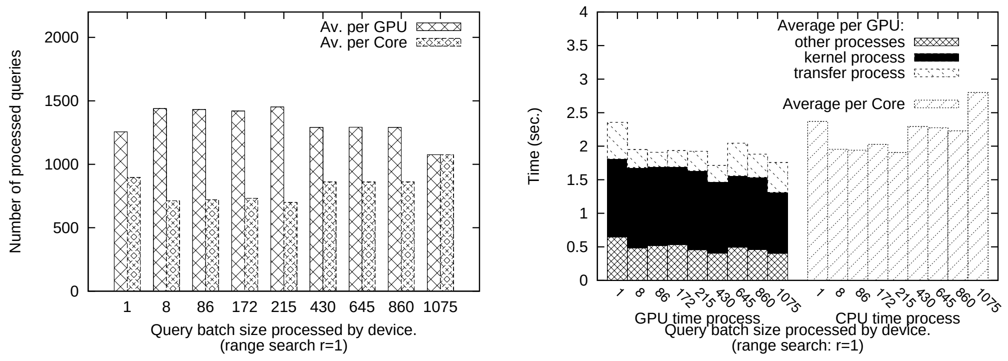
Regarding the execution time distribution, we observe that the time spent in transferring the queries has been dramatically reduced up to 50% respect to the implementation in Section 4.1.

4.2.3 Scaling the number of queries

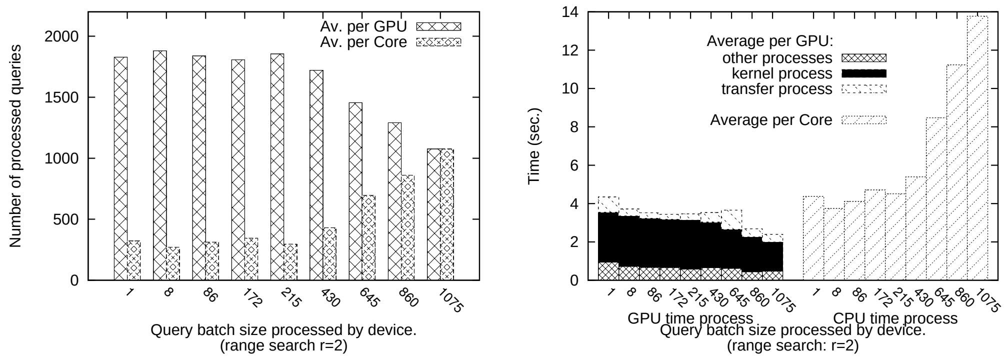
When we increase the number of queries in order to observe the behavior of the system in a stress operation mode, the best behavior does not correspond to blocks of size 1 query as in the version 1, but also for block size between 10 and 100 for the words space (see Figure 27) and between 100 and 1000 to the vector space (see Figure 28). For color histograms, a block size of 10000 cannnot be used because the needed structures cannot be allocated to the global memory.

Figures 29 and 30 show the comparative results for words space and vector space using the best choice in version 1, presented in Section 4.1, and version 2 (here presented). We appreciate that the block-oriented implementation improves the execution time. This improvement ranges from 20% (100 thousand queries) to 28% (900 thousand queries) for the color histograms.

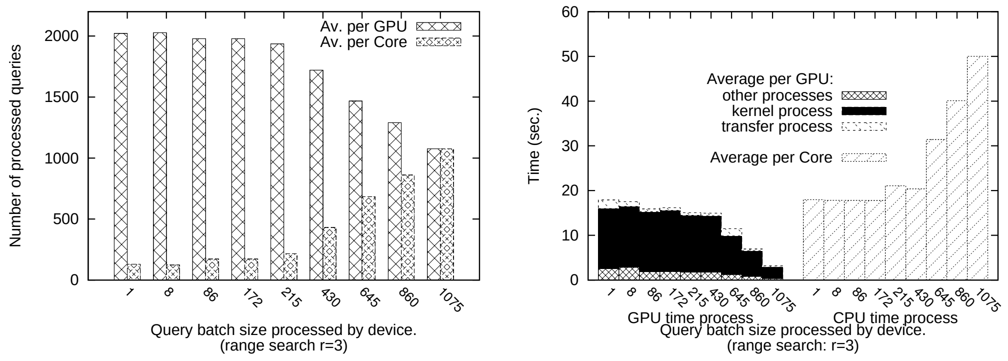
For the words space we need to make a deeper analysis. For a blocksize of 100 and range search 1, the improvement ranges from 19% (10 thousand queries) to 21% (40 thousand queries). Nevertheless, a blocksize of 100 was not apropriate for range 4, getting better results for a blocksize of 10 with an improvement of 9% for 40 thousand



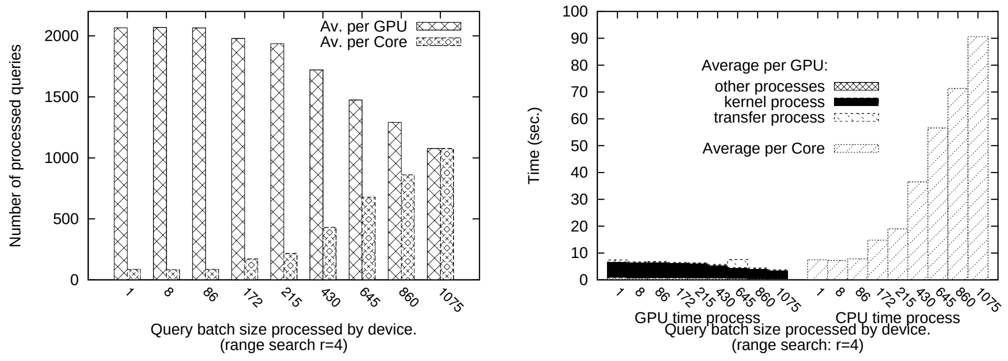
(a) $r = 1$



(b) $r = 2$

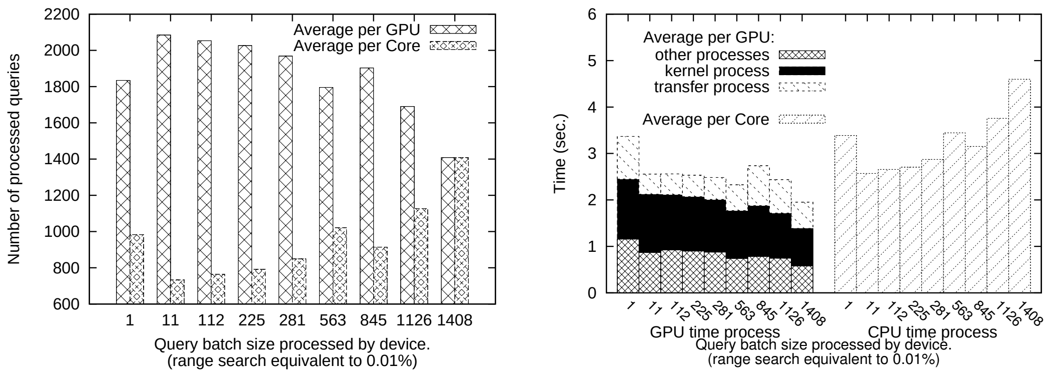


(c) $r = 3$

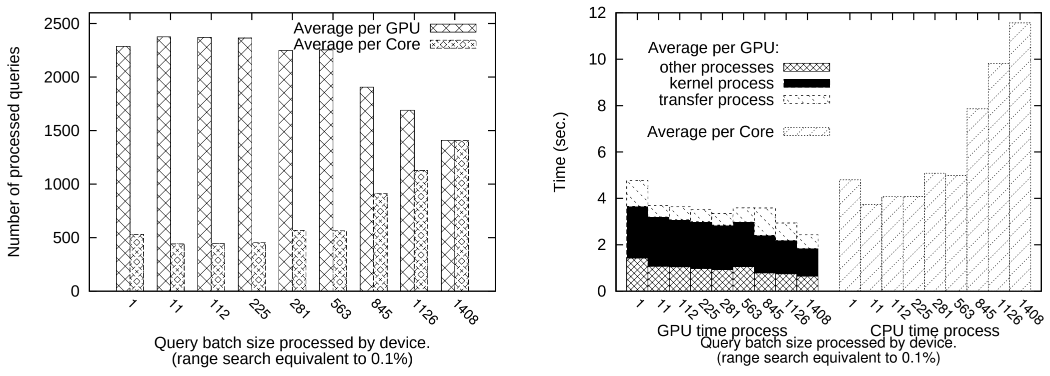


(d) $r = 4$

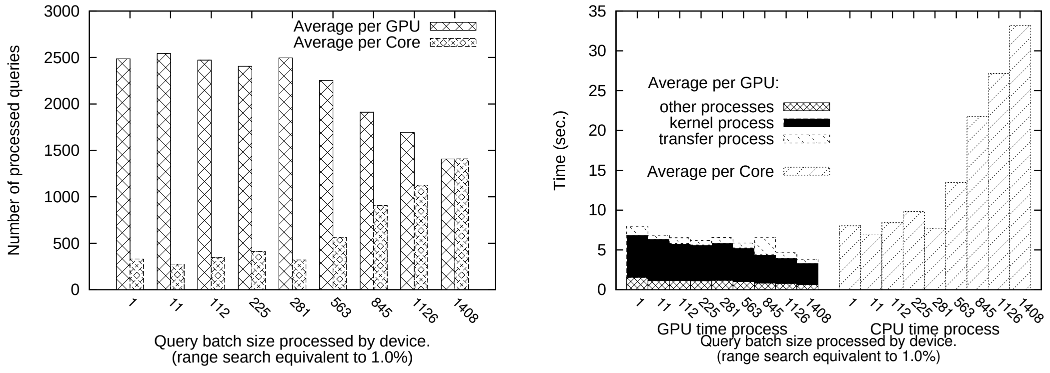
Figure 25: Distribution of queries and execution time obtained by the heterogeneous system for the Spanish Dictionary case study, considering 8 cores and 4 GPU. Left column represents query distribution and right column represents execution time distribution.



(a) 0,01%



(b) 0,1%



(c) 1,0%

Figure 26: Distribution of queries and execution time obtained by the heterogeneous system for the case color histogram study, considering 8 cores and 4 GPU. Left column represents query distribution and right column represents execution time distribution.

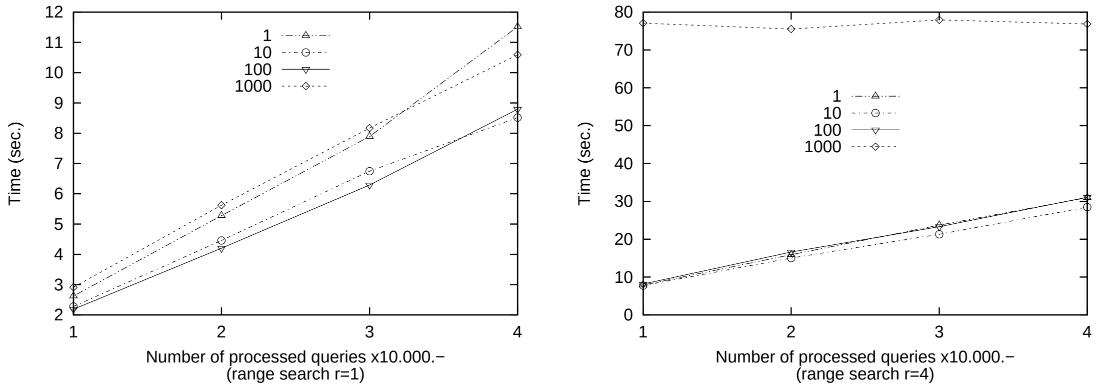


Figure 27: Version 2: System scalability. Execution time vs. chunk size when the number of queries is varied from 10 thousand to 40 thousand considering range search 1 and 4 in a system with 8 cores and 4 GPUs.

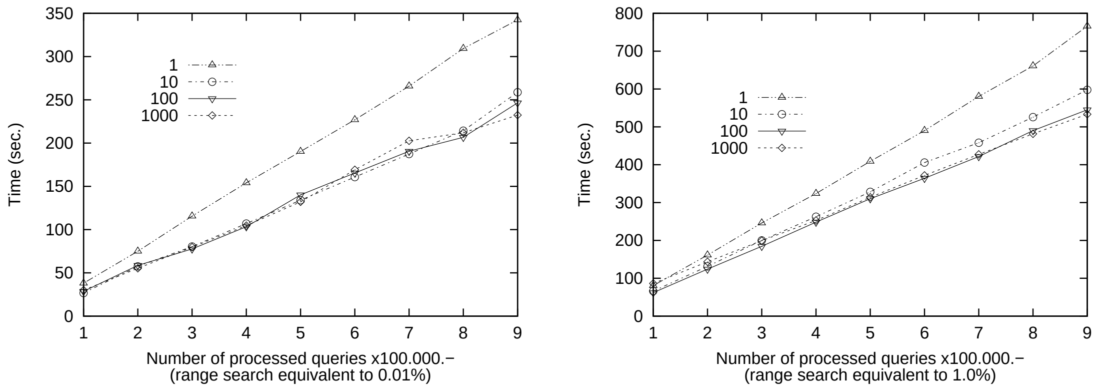


Figure 28: Version 2: System scalability. Execution time vs. chunk size when the number of queries is varied from 100 thousand to 900 thousand considering 0.01% and 1% of percentage of retrieved information in a system with 8 cores and 4 GPUs.

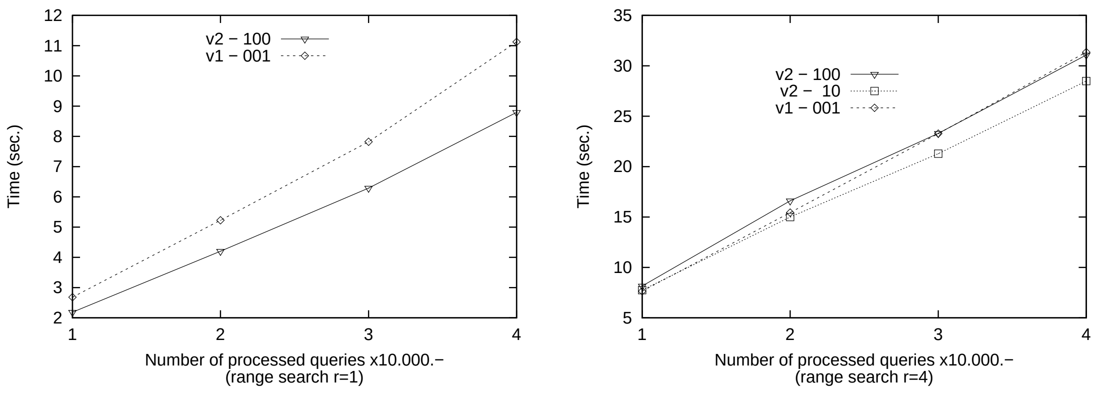


Figure 29: Version 2: Scaling the number of queries from 10 thousand to 40 thousand considering range search 1 and 4 in a system with 8 cores and 4 GPUs. Comparing results for version 1, best option with number of queries equal to 1, versus the second version with block size equal to 100 ($MQ = 100$) and also equal to 10 for range 4.

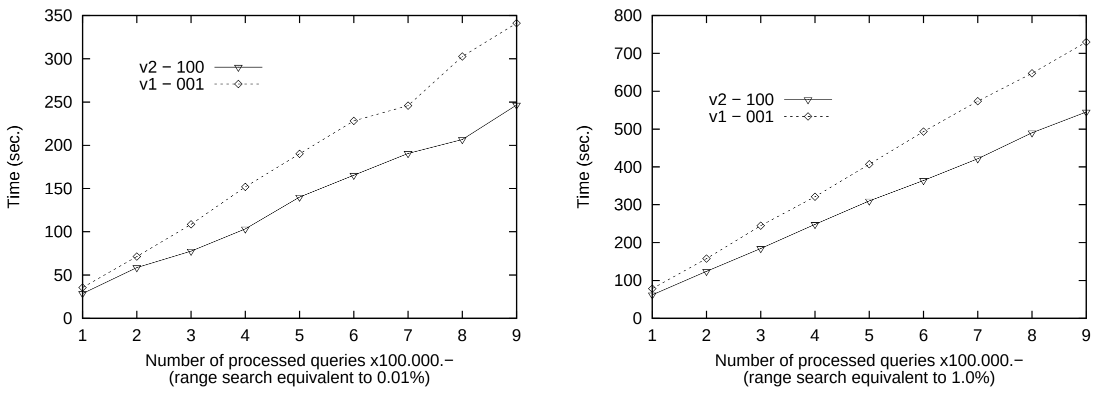


Figure 30: Version 2: Scaling the number of queries from 100 thousand to 900 thousand considering 0.01% and 1% of percentage of retrieved information in a system with 8 cores and 4 GPUs. Comparing results for version 1, best option with number of queries equal to 1, versus the second version with block size equal to 100 ($MQ = 100$).

queries but no improvement for 10 thousand queries, i.e., this blocksize is still big for the amount of queries considered.

4.2.4 Discussion

As expected, the block-oriented implementation improves the performance of the 1 query implementation. This is due to the fact that the time spent in transferring the queries has been dramatically reduced up to 50% with respect to the version 1. The execution time has been improved (with respect to v1) up to 25%, and up to 27% when scaling.

However, dealing with a block of queries has a fault. That is, dealing with a block of queries forces to allocate needed structures in the global memory to store the solutions. In particular, in the worst case we need a bidimensional array of dimension $MQ \times D$, being MQ the block size and D the number of objects in database. In version 1, we only need, as a maximum, an array of dimension D. Thus, the block size used in version 2 is limited by the available storage. However, this situation is not of concern because we have proved along the paper that small block sizes are more suitable.

With respect to the sequential version, the speed-ups obtained range from 13 to 36 for color histograms, and from 10 to 97 for words space.

5 Conclusion and Future Works

In this paper two heterogeneous implementations for similarity search in metric spaces, using a generic metric structure (GMS), were presented. The first one (version 1) takes into account that each query is completely processed in a unique kernel (one query - one kernel). The second one (version 2) corresponds to a block-oriented implementation transferring a chunk of queries (MQ queries) at a time and processing it in the unique kernel (MQ queries - one kernel).

With respect to memory usage, more queries need to be allocated to the global memory in version 2, leaving less space for the GMS and the database (comparing with version 1). Moreover, also a bigger data structure is required to store the results of the queries.

The time spent in data transfer and calls to the kernel has been reduced in version 2 with respect to version 1. Although the amount of data transferred is the same in both versions, it is faster to transfer MQ queries in one packet (one kernel) than transfer MQ "packets" of one query (MQ kernels). In general, the reduction in transfering data is up to 50%.

Another measure of the speed of the algorithm is answering the question, how many

Algorithms \ Spaces	Spanish Dictionary		Color Histograms	
	1	4	0.1	1.0
sequential (1 core)	409	12	333	44
multicore (8 cores)	3098	95	2472	342
1 GPU (1 q - 1 k)	616	290	648	336
1 GPU (MQ q - 1 k)	769	310	873	372
4 GPU (1 q - 1 k)	2131	1122	2069	1305
4 GPU (MQ q - 1 k)	2922	1216	3086	1493
Full System v 1	3638	1152	3292	1411
Full System v 2	4378	1169	4362	1586

Table 3: Number of queries per second for all versions, considering only the maximum and minimum range.

queries are processed per second?. The answer is given in Table 3.

In Table 3 the worst case is given when considering the upper range search for Spanish dictionary case study (r=4) or the upper percentage of information retrieved for the color histograms case study (1%). On the other hand, the best case is given when considering the lower range search for the Spanish dictionary case study (r=1) or the lower percentage of information retrieved for the color histograms case study (0.01%). As we expected, the number of queries processed for both case studies is similar, and evidently the version 2 processes a high number of queries, especially when the best case is considered.

Comparing the sequential version against the full system v2 we can see that for small ranges we get a speed-up between 10 and 13, while for high ranges we get a speed-up between 36 and 97. The reason of these results is related to the behaviour of the metric structure. For small ranges more objects are discarded, so less distance evaluations are performed. On the contrary, for high ranges the behaviour of the structure is worst, more distance evaluations are performed, there is more computational load, and more benefits can be obtained from the parallel platform.

If the measure to take into account is the number of queries processed by the GPUs versus the number of queries processed by the core, this ratio varies between 1.5 and 20 for the Spanish dictionary case study and between 2 and 6 for the color histograms case study. In both cases, the best results are obtained when the complexity of the problem increases. That profs that the GPUs are better exploited.

Finally, as shown, the system scales properly, obtaining an almost linear performance.

According to the experimental results, sometimes the use of the CPU cores instead of benefitting the performance represents a bottleneck. Thus, in a future work other kind of load distribution can be considered. For instance, due to the fact that GPUs are quicker than CPU cores, it is possible to assign a fix amount of queries to all the GPUs (for example, 80% of queries) and the rest to the CPU cores. Then, this 80% of queries are processed by the GPUs in blocks as in this paper. The rest of queries are processed dynamically one-by-one by the CPU cores. Evidently, in this case it is necessary to tune in this percentage according to a previous speed-up study.

During this paper, we have considered different parameters to measure the goodness of the implementation, as execution time, distance evaluation and needed storage. However, we can introduce a new parameter as energy efficiency or energy consumption. In this case, we could assess that the better implementation spends less time, needs less memory and consumes less energy.

Finally, we could extend the block-oriented implementation not only to GPUs but also to main memory and secondary memory in order to improve the performance.

Acknowledgments

This work has been partially supported by the Ministerio de Ciencia e Innovación, project SATSIM (Ref: CGL2010-20787-C02-02), Spain and Vice-Chancellory of Research at the University of Magallanes, Chile.

References

[1] *NVIDIA CUDA C Programming Guide, Version 4.0.* NVIDIA, 2011. http://developer.nvidia.com/object/ gpucomputing.html.

[2] Adil Alpkocak, Taner Danisman, and Ulker Tuba. A parallel similarity search in high dimensional metric space using m-tree. In *Advanced Environments, Tools, and Applications for Cluster Computing*, volume 2326 of *LNCS*, pages 247–252. Springer Berlin / Heidelberg, 2002.

[3] Ricardo Baeza-Yates, Walter Cunto, Udi Manber, and Sun Wu. Proximity matching using fixed-queries trees. In *5th Combinatorial Pattern Matching (CPM'94)*, volume 807 of *LNCS*, pages 198–212. Springer Berlin Heidelberg, 1994.

[4] Ricardo J. Barrientos, José I. Gómez, Christian Tenllado, Manuel Prieto, and Mauricio Marin. kNN query processing in metric spaces using GPUs. In *17th International European Conference on Parallel and Distributed Computing (Euro-Par 2011)*, volume 6852 of *LNCS*, pages 380–392, Berlin, Heidelberg, 2011. Springer.

[5] Ricardo J. Barrientos, José I. Gómez, Christian Tenllado, Manuel Prieto, and Mauricio Marin. Range query processing in a multi-GPU environment. In *10th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA 2012)*, pages 419–426, Madrid, Spain, July 2012.

[6] Sergei Brin. Near neighbor search in large metric spaces. In *the 21st VLDB Conference*, pages 574–584. Morgan Kaufmann Publishers, 1995.

[7] Edgar Chávez, José L. Marroquín, and Gonzalo Navarro. Fixed queries array: A fast and economical data structure for proximity searching. *Multimedia Tools and Applications*, 14(2):113–135, June 2001.

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

[9] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree : An efficient access method for similarity search in metric spaces. In *the 23st International Conference on VLDB*, pages 426–435, 1997.

[10] Vincent Garcia, Eric Debreuve, and Michel Barlaud. Fast k nearest neighbor search using GPU. *Computer Vision and Pattern Recognition Workshop*, 0:1–6, 2008.

[11] Veronica Gil-Costa, Ricardo Barrientos, Mauricio Marin, and Carolina Bonacic. Scheduling metric-space queries processing on multi-core processors. *Euromicro Conference on Parallel, Distributed, and Network-Based Processing*, 0:187–194, 2010.

[12] Veronica Gil-Costa, Mauricio Marin, and Nora Reyes. Parallel query processing on distributed clustering indexes. *Journal of Discrete Algorithms*, 7(1):3–17, 2009.

[13] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. *Introduction to Parallel Computing (2nd Edition)*. Addison Wesley, 2 edition, 2003.

[14] Magnus Hetland. The basic principles of metric indexing. In Carlos Coello, Satchidananda Dehuri, and Susmita Ghosh, editors, *Swarm Intelligence for Multi-objective Problems in Data Mining*, volume 242 of *Studies in Computational Intelligence*, pages 199–232. Springer Berlin / Heidelberg, 2009.

[15] Iraj Kalantari and Gerard McDonald. A data structure and an algorithm for the nearest point problem. *IEEE Transactions on Software Engineering*, 9(5):631–634, September 1983.

[16] Quansheng Kuang and Lei Zhao. A practical GPU based kNN algorithm. *International Symposium on Computer Science and Computational Technology (ISCSCT)*, pages 151–155, 2009.

[17] Luisa Micó, José Oncina, and Rafael C. Carrasco. A fast branch & bound nearest neighbour classifier in metric spaces. *Pattern Recognition Letters*, 17(7):731–739, June 1996.

[18] María Luisa Micó, José Oncina, and Enrique Vidal. A new version of the nearest-neighbour approximating and eliminating search algorithm (aesa) with linear preprocessing time and memory requirements. *Pattern Recognition Letters*, 15(1):9–17, January 1994.

[19] Gonzalo Navarro and Roberto Uribe-Paredes. Fully dynamic metric access methods based on hyperplane partitioning. *Information Systems*, 36(4):734 – 747, 2011.

[20] Oscar Pedreira and Nieves R. Brisaboa. Spatial selection of sparse pivots for similarity search in metric spaces. In *33rd Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2007)*, volume 4362 of *LNCS*, pages 434–445, Harrachov, Czech Republic, 2007. Springer.

[21] Jeffrey K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. In *Information Processing Letters*, volume 40, pages 175–179, 1991.

[22] Roberto Uribe-Paredes, Enrique Arias, José L. Sánchez, Diego Cazorla, and Pedro Valero-Lara. Improving the performance for the range search on metric spaces using a multi-GPU platform. In *Database and Expert Systems Applications (DEXA)*, volume 7447 of *Lecture Notes in Computer Science*, pages 442–449. Springer Berlin Heidelberg, 2012.

[23] Roberto Uribe-Paredes, Diego Cazorla, José L. Sánchez, and Enrique Arias. A comparative study of different metric structures: Thinking on GPU implementations. In *Lecture Notes in Engineering and Computer Science: Proceedings of The World Congress on Engineering 2012 (WCE 2012)*, pages 312–317, London, England, July 2012.

[24] Roberto Uribe-Paredes, Pedro Valero-Lara, Enrique Arias, José Luis Sanchez, and Diego Cazorla. Similarity search implementations for multi-core and many-core processors. In *International Conference on High Performance Computing and Simulation (HPCS)*, pages 656–663, 2011.

[25] URL. Green 500 list. http://www.green500.org/.

[26] URL. Top 500 list. http://www.top500.org/.

[27] Pavel Zezula, Pasquale Savino, Fausto Rabitti, Giuseppe Amato, and Paolo Ciaccia. Processing m-trees with parallel resources. In *RIDE ’98: Proceedings of the Workshop on Research Issues in Database Engineering*, pages 147–, Washington, DC, USA, 1998. IEEE CS.