Received: 18 June 2018 | Revised: 10 December 2019 | Accepted: 29 January 2020

DOI: 10.1002/cpe.5702

RESEARCH ARTICLE

WILEY

A unified model for accelerating unsupervised iterative re-ranking algorithms

Flávia Pisani[1,2] | **Lucas Pascotti Valem**[3] | **Daniel Carlos Guimarães Pedronette**[3] | **Ricardo da S. Torres**[4] | **Edson Borin**[2] | **Mauricio BreternitzJr.**[5]

[1]Department of Informatics, Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, RJ, Brazil

[2]Institute of Computing, University of Campinas, Campinas, SP, Brazil

[3]Department of Statistics, Applied Mathematics and Computing, São Paulo State University, Rio Claro, SP, Brazil

[4]Department of ICT and Natural Sciences, NTNU - Norwegian University of Science and Technology, Ålesund, Norway

[5]ISCTE-IUL Lisbon University Institute, ISTAR-IUL, Lisbon, Portugal

Correspondence
Flávia Pisani, Institute of Computing, University of Campinas, Av. Albert Einstein, 1251, Cidade Universitária, Campinas, SP, Brazil.
Email: fpisani@ic.unicamp.br

Funding information
Advanced Micro Devices; Conselho Nacional de Desenvolvimento Científico e Tecnológico, Grant/Award Numbers: #307560/2016-3, #484254/2012-0, #308194/2017-9, an; Coordenação de Aperfeiçoamento de Pessoal de Nível Superior, Grant/Award Number: #88881.145912/2017-01; FAPESP-Microsoft Virtual Institute, Grant/Award Numbers: #2013/50155-0, #2013/50169-1, #2014/50715-9; Fundação de Amparo à Pesquisa do Estado de São Paulo, Grant/Award Numbers: #2013/08645-0, #2014/12236-1, #2015/24494-8, #2016; Fundo de Apoio ao Ensino, à Pesquisa e Extensão, Universidade Estadual de Campinas

Summary

Despite the continuous advances in image retrieval technologies, performing effective and efficient content-based searches remains a challenging task. Unsupervised iterative re-ranking algorithms have emerged as a promising solution and have been widely used to improve the effectiveness of multimedia retrieval systems. Although substantially more efficient than related approaches based on diffusion processes, these re-ranking algorithms can still be computationally costly, demanding the specification and implementation of efficient big multimedia analysis approaches. Such demand associated with the significant potential for parallelization and highly effective results achieved by recently proposed re-ranking algorithms creates the need for exploiting efficiency vs effectiveness trade-offs. In this article, we introduce a class of unsupervised iterative re-ranking algorithms and present a model that can be used to guide their implementation and optimization for parallel architectures. We also analyze the impact of the parallelization on the performance of four algorithms that belong to the proposed class: Contextual Spaces, RL-Sim, Contextual Re-ranking, and Cartesian Product of Ranking References. The experiments show speedups that reach up to 6.0×, 16.1×, 3.3×, and 7.1× for each algorithm, respectively. These results demonstrate that the proposed parallel programming model can be successfully applied to various algorithms and used to improve the performance of multimedia retrieval systems.

KEYWORDS
GPGPU, image re-ranking model, multimedia retrieval, OpenCL, parallel computing

1 | INTRODUCTION

The increasing popularity of multimedia data acquisition and sharing technologies has contributed to the generation of large multimedia collections in recent years. This scenario demands the creation of effective and efficient retrieval services. In the context of supporting image searches, one approach that has become increasingly important relies on the use of Content-Based Image Retrieval (CBIR) systems. However, despite the continuous advances made in this area, in terms of either effectiveness[1-3] (quality of retrieved images) or efficiency[4-6] (time spent to obtain the results), various research challenges are still open.

Intending to address some of these challenges, several studies[7-10] have demonstrated that unsupervised techniques can significantly improve the effectiveness of CBIR systems without requiring any user intervention. As the effectiveness of these systems is very dependent on the distance metrics adopted, substantial work[11-17] has also been conducted with the aim of improving the effectiveness of these metrics. Diffusion processes[18-20] and graph-based learning[21] approaches have been applied with this purposed. More recently, unsupervised iterative *re-ranking* algorithms[9,14,22-24] were proposed to improve the effectiveness of retrieval tasks by exploiting contextual information. The goal of these approaches is to mimic the behavior of humans considering specific *contexts* when judging the similarity of objects.

Usually, these methods replace pairwise similarities by more global affinity metrics that consider the relationships among collection images, which are encoded in ranked lists. In an iterative process, pairwise distances between images are recomputed and ranked lists are updated to reflect the contextual information incorporated in these new distances. Although much more efficient than related methods based on diffusion process, one drawback of re-ranking solutions, however, is the computational effort required to execute them on large-scale datasets. In short, this means that while these re-ranking algorithms are effective, they lack in efficiency, making them inappropriate for handling large multimedia datasets. That being said, we note that the unsupervised iterative re-ranking algorithms mentioned have good potential for parallelization, and thus we can increase their efficiency by taking advantage of parallel architectures.

One example of a parallel architecture that can be employed to accelerate this type of algorithm involves graphics processing units (GPUs), as advances in GPU hardware and programming models have enabled general-purpose computation on these devices. Among other accelerators, general-purpose graphic processing units (GPGPUs), as they are known, are present on several of the top-500 high-performance computing systems list.[25] The increasing popularity of GPGPUs has led to the development of heterogeneous computing architectures[26] and the use of highly parallel heterogeneous devices, such as a single chip that integrates a GPU and a central processing unit (CPU), also named accelerated processing unit (APU). These devices have execution and programming models that are different from traditional general-purpose processors (GPPs), meaning that simple recompilation techniques do not apply when porting parallel applications to these heterogeneous systems and the responsibility of choosing the correct tools and reimplementing the algorithms is left to the programmer.

In this article, we propose a unified model that allows the parallelization and efficient execution of a class of unsupervised iterative re-ranking algorithms on diverse processing devices. The main contributions and novelties with regard to previous work[27-30] can be summarized in terms of four aspects: *(i)* it presents a detailed description of a class of unsupervised iterative re-ranking algorithms; *(ii)* it introduces a unified model that generalizes the implementation of these algorithms for parallel architectures; *(iii)* it shows how this model can be used to simplify the parallelization of algorithms that belong to the proposed class; and *(iv)* it validates our method on many parallel devices, including APUs and different GPUs.

Figure 1 provides an overview of the parallel execution, including the steps that may be computed sequentially and the ones that should be performed in parallel. As *initialize data structures* and *normalization* are very simple procedures, which may not benefit from parallel execution due to overhead, they are executed sequentially. On the other hand, *update distances/similarities* and *ranked list resorting* should be parallelized to improve performance. The sequence of all steps is executed iteratively.

Our solution is validated through the implementation of four algorithms that belong to this class, namely, contextual spaces re-ranking,[14] RL-Sim re-ranking,[9] contextual re-ranking,[22] and Cartesian product of ranking references.[30] However, it is worth mentioning that the discussed model can also be applied to other ranking algorithms in the literature.[16,31]

Using OpenCL,[26] we were able to test our implementations in different environments, showing that the model properly handles many design challenges, such as synchronization and concurrency issues. By choosing an open standard for programming heterogeneous devices, which is supported by several hardware accelerator vendors instead of other popular yet proprietary solutions such as CUDA, we increase the portability of the code, thus making our tests closer to real-world applications, where the same parallel algorithm must be executed in multiple settings. Other studies that report efficiency results[23,24] do not evaluate parallelization strategies and therefore will not be included in our analysis.

This article is organized as follows: Section 2 discusses related work. Section 3 presents a class of unsupervised iterative re-ranking algorithms, whereas Section 4 describes methods that belong to this class. Section 5 briefly outlines parallel solutions for these re-ranking algorithms using OpenCL. Section 6 reports the results of the performance experiments of our parallelization approach, and Section 7 shows the results for our experiments with larger datasets. Finally, Section 8 states our conclusions and presents possible research venues to be considered in future work.

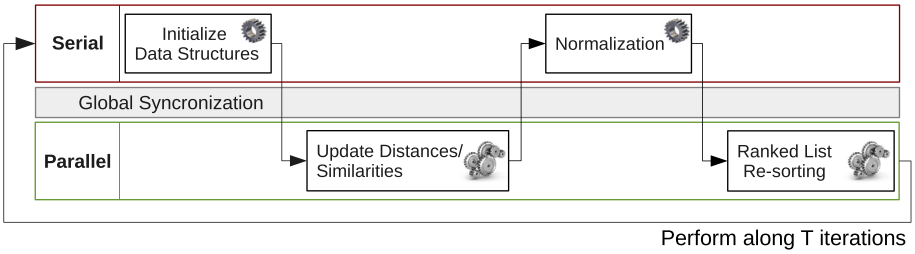


FIGURE 1 Overview of the parallel execution of the unified iterative re-ranking model

2 | RELATED WORK

This section describes other studies that are related to the topic of this article. Section 2.1 introduces image retrieval and re-ranking techniques, whereas Section 2.2 describes the use of GPUs for general-purpose computations. Section 2.3 discusses parallelization of CBIR methods, focusing on GPGPU approaches. Finally, Section 2.4 introduces other efforts to model image retrieval algorithms.

2.1 | Image retrieval and re-ranking in CBIR tasks

Content-based image retrieval (CBIR) systems aim at retrieving the elements in a collection that are most similar to a given query image.[2] In order to do so, it is necessary to have a metric to make comparisons, which is generally obtained by computing a predefined distance measure between the query image and each one of the collection images. Traditional distance metrics, such as the Euclidean distance, are often adopted in these cases and are able to express the pairwise similarity between any two images.

However, these approaches fail to return satisfactory results in many situations, mainly due to the well-known semantic gap problem.[32-34] This has motivated research attempts to improve distance metrics in CBIR systems in the past few years,[8,11-16,22,31,35,36] leading to promising results considering several approaches and postprocessing techniques.[19,37-39]

The use of context can also play an important role in CBIR applications; nonetheless, traditional systems usually perform only pairwise image analysis (i.e., they compute similarity or distance measures considering only pairs of images).[10] Because of this, many CBIR approaches[11,12,14,20,23,34-36] were recently proposed to improve the effectiveness of retrieval tasks by replacing the use of pairwise similarities with more global affinity metrics that consider the relationship among collection objects without needing labeled training data.

Figure 2 shows an example of how the use of contextual information can improve the result of CBIR systems. Let $\rho(i_x, i_y)$ be the distance between images x and y and suppose that image 5 (i_5, with continuous green border) is the query image. We see that image 9 (i_9) is close to the top of the query's ranked list, so we analyze i_9's ranked list looking for images that are similar to it (this ranked list is indicated by the red arrow pointing to i_9 with dashed green border). We find out that image 20 (i_{20}, with dotted orange border) is close to the top of i_9's ranked list. Therefore, since i_9 is similar to i_5, we can improve i_5's rank by moving i_{20} closer to the top.

Our focus in this article is on *unsupervised learning* approaches, meaning that the methods analyzed consider only the domain of object instances and no labeled training data are needed. Since labeling is often a laborious and time-consuming task, whereas it is far easier to obtain unlabeled data, these techniques often represent a very attractive solution. In addition, we adopted an iterative strategy to process contextual information with the goal of re-ranking the images returned at top positions of ranked lists.[9,14,15,35,40]

2.2 | General-purpose computing on GPUs (GPGPUs)

Graphics processing units (GPUs) are power-efficient, massively-parallel computing devices and their use as parallel processors is fast emerging due to the fact that they combine high computation power and low price.

Once specially designed for computer graphics, today's GPUs have support for accessible programming frameworks such as OpenCL (described in Section 5.1) and are attracting researchers who employ them for general-purpose computing due to their extensive data processing capability.[41]

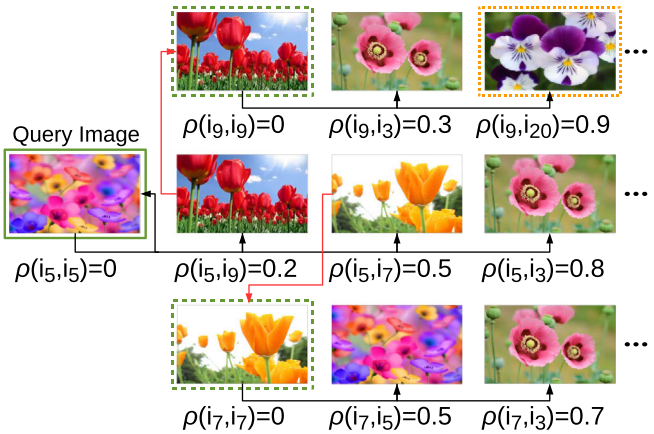


FIGURE 2 Example of the use of contextual information. The image with dotted orange border (i_{20}) is close to the top of i_9's ranked list. Given that i_9 is close to the top of i_5's ranked list, i_{20} should be as well

As these devices are particularly suitable for highly data parallel problems, using them is an interesting approach for multimedia applications that need a large amount of computing resources.[42]

Gunarathne et al.[43] designed a GPU-based solution for iterative statistical applications and implemented three iterative statistical algorithms (K-means, multidimensional scaling, and PageRank) using OpenCL. We note that iterative algorithms in general are at the core of several scientific applications and have traditionally been parallelized and optimized for large multiprocessors, based on either shared memory or clusters of interconnected nodes.

OpenCL and GPUs were also used by Strong and Gong,[44] who accelerated their self-sorting map (SSM) algorithm for organizing and visualizing multimedia, making it possible to arrange millions of items in a structured layout with no overlap within seconds.

Wu et al.[45] presented a study on efficient execution of the PageRank algorithm on GPUs. They analyzed the characteristics of the sparse matrices used in PageRank and implemented a fast sparse matrix-vector multiplication (SpMV) using a modified compressed sparse row (CSR) format.

A GPGPU approach with a large number of processing units for on-line machine learning was introduced by Xiao et al.[46] Their work considers the stochastic gradient descent algorithm, discussing a parallel solution, its performance gain, and variations in accuracy.

A GPU-based approach for computing large-scale distance matrices was proposed by Arefin et al..[47] Distance matrices contain pairwise distances and have a wide range of usage in several fields of scientific research, for example, machine learning, image analysis, and information retrieval. Their work splits these matrices into submatrices and uses GPUs to divide computational tasks and data.

Machine learning tasks on GPUs were also addressed by other studies, such as the one by Cano et al.,[48] which evaluated the Pittsburgh rule-based classifiers on GPUs by considering a model that parallelizes the fitness computation. Their experimental study supports the conclusions about the efficiency and high performance of GPUs for this type of task.

2.3 | Parallel image retrieval and re-ranking

In addition to the applications described in Section 2.2, GPGPU approaches can also benefit image retrieval tasks.

Strong and Gong[49] discussed how to efficiently organize a collection of images based on their similarities with the objective of facilitating photo browsing and searching. Their method generates a feature vector for each image in the collection and then uses these vectors to train a self-organizing map (SOM) with the help of GPUs.

Another image retrieval technique exploiting GPUs was developed by Pham et al.[50] who presented two algorithms implemented for GPUs that retrieve images using factorial correspondence analysis (FCA), which reduces dimensions and limits the number of elements considered during the search. They adapted the FCA method, normally applied to textual data analysis, to handle images using scale-invariant feature transformation (SIFT) local descriptors.

Zhu et al[51] introduced a GPU-based, high-throughput image retrieval algorithm. Their work analyzed the parallelism in the implementation of the local descriptor SURF and mapped tasks on a GPU through the use of block-level parallelism. In another research venue, image searches based on local descriptors were accelerated by using indexing schemes that exploit distributed CPU-GPU platforms.[52] Parallel computing on GPUs is also pointed to as a promising, efficient solution for other image retrieval tasks.[53]

2.4 | Modeling unsupervised algorithms for image retrieval

Most methods in the field of unsupervised learning algorithms for image retrieval follow the same principle: first, the manifold, defined by the provided affinity matrix, is interpreted as a weighted graph; then, the pairwise affinities are reevaluated in the context of all other elements by diffusing the similarity values through this graph. Donoser and Bischof[19] revisited algorithms that follow this pattern and derived a generic framework for this type of technique.

Unlike their study, we address the design of a parallelization model for a class of algorithms that is based on iterative re-ranking methods. Also, we evaluate the proposed approach by using the model to parallelize four algorithms that belong to this class.

Given that both image re-ranking and GPGPUs are relatively recent approaches, to the best of our knowledge, there are no other studies about parallel models for efficient image re-ranking computation with the help of GPUs.

3 | INTRODUCING A CLASS OF UNSUPERVISED ITERATIVE RE-RANKING ALGORITHMS

This section presents the class of unsupervised iterative re-ranking algorithms considered in our study. The execution of these algorithms relies on using contextual information about the image collection, such as the distances for all pairs of images in the collection or the lists that rank all images in increasing order of their distances to queries (considering each collection image as a query).

FIGURE 3 Typical steps in the considered class of re-ranking algorithms

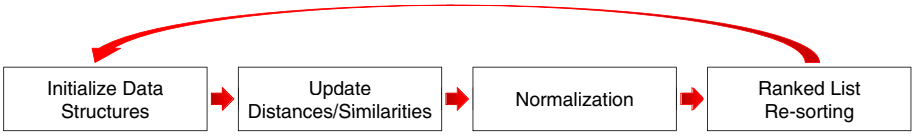
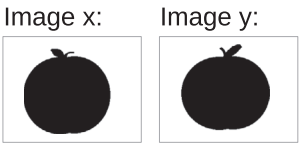


FIGURE 4 Similar reference images



The methods start using traditional pairwise distance measures (e.g., measures obtained using the Euclidean distance) to express the distance between each pair of images in the collection. These distances are then used to analyze the relationships among images and to create new measures that regard global affinity, improving the results of the CBIR system. Nonetheless, whenever the method recomputes the distances, it also needs to update the ranked lists in order to reflect the newly incorporated contextual information, thus creating two separate tasks that must be performed.

Taking that into consideration, we can categorize unsupervised iterative re-ranking algorithms as a class that is represented by two main steps:

1. *Update distances/similarities:* the re-ranking algorithm analyzes the contextual information defined in terms of the relationships among images, which is encoded in ranked lists and in the distances/similarities among images. Based on this analysis, it is possible to compute new distances/similarities for each pair of images. Different algorithms may use distinct approaches for this recomputation: similarity between ranked lists,[9] image processing techniques,[22] clustering methods,[35] among others.
2. *Resort images to generate new ranked lists (or simply, ranked list resorting):* the ranked lists should present the retrieved images in an increasing order of distance (i.e., decreasing order of similarity). To ensure the consistency between the ranked lists and new distances/similarities, the algorithm sorts collection images according to their new distances/similarities to the query.

In addition, it is necessary to execute auxiliary operations, such as initializing data structures before the first step and normalizing the new distances before the second. This normalization guarantees that the distance from an image x to an image y is the same as from y to x. Due to the iterative nature of the methods, all tasks must be repeated a certain number of times (T).

Figure 3 summarizes the main steps of the class of rerank algorithms considered in this work.

4 | ITERATIVE RE-RANKING METHODS

This section presents a brief description of four iterative re-ranking approaches that belong to the class of algorithms described in Section 3: *contextual spaces re-ranking*,[14] *RL-Sim re-ranking*,[9] *contextual re-ranking*[22], and *Cartesian product of ranking references*[30]. For each method, we describe how each step of Figure 3 is implemented.

In this section, we use the following convention: $\mathcal{C}$ is an image collection of size N and $\mathcal{D}$ is an image descriptor. The distance function ρ defined by $\mathcal{D}$ can be used to compute the distance $\rho(img_x, img_y)$ for all $img_x, img_y \in \mathcal{C}$ in order to obtain an $N \times N$ distance matrix, A. The examples given in this section are based on data from the MPEG-7 image collection[54] with the CFD shape descriptor.[55]

4.1 | The contextual spaces re-ranking algorithm

The *contextual spaces re-ranking* algorithm[14] relies on contextual spaces to exploit image relationships in the context of the query instead of just using pairwise distances. *Contextual Spaces* are bidimensional representations of an image collection regarding two reference images, $img_x, img_y \in \mathcal{C}$. They are constructed considering the nearest neighbors of a query image, which are the elements that are most similar to the query according to a descriptor. Similar reference images, like the ones in Figure 4, produce a contextual space that looks like the one depicted in Figure 5. Dissimilar reference images, such as the ones in Figure 6, produce a contextual space analogous to Figure 7.

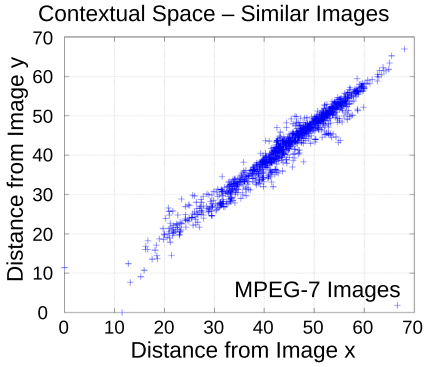


FIGURE 5 Contextual Space for two similar reference images

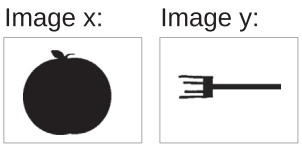


FIGURE 6 Dissimilar reference images

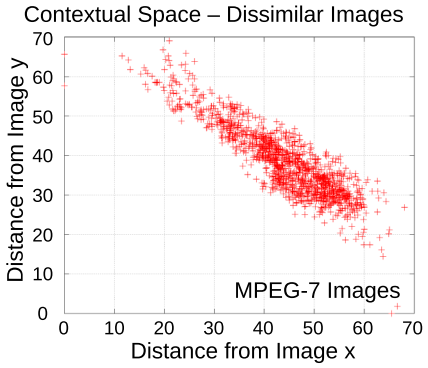


FIGURE 7 Contextual space for two dissimilar reference images

This method implements the steps from Figure 3 as follows:

1 *Initialize data structures:* The distance matrix A is taken as input.
2 *Update distances:* The re-ranking algorithm takes into account the distances between each nearest neighbor and the other collection images and uses this information to calculate new distances. This information is encoded by k contextual spaces defined in terms of the k-nearest neighbors of a query.
3 *Normalization:* Let t be the current iteration. For all images $x, y \in \mathcal{C}$, set distances $A_{t+1}[x, y] = A_{t+1}[y, x] = min(A_{t+1}[x, y], A_{t+1}[y, x])$.
4 *Ranked list resorting:* The image collection is then reranked based on these new distances.

This process is repeated iteratively, further improving the effectiveness of the results each time it is performed.

4.2 | The RL-Sim re-ranking algorithm

The *RL-Sim re-ranking* algorithm[9] characterizes contextual information by computing the similarity among ranked lists. This is possible due to the intuitive premise that, in general, if two images are similar, their ranked lists should be similar as well.

This method implements the steps from Figure 3 as follows:

1. *Initialize data structures:* The distance matrix A is taken as input.
2. *Update distances:* This algorithm uses the contextual distance measure, which is defined regarding the similarity/dissimilarity of ranked lists, that is, the distance between two images is updated by taking into account the similarity of their ranked lists. While the distance value $\rho(img_x, img_y)$

FIGURE 8 Updating the distance matrix based on the similarity of a ranked list

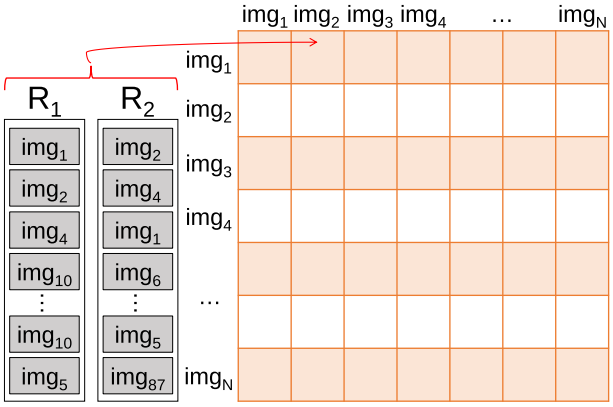


FIGURE 9 Similar reference images

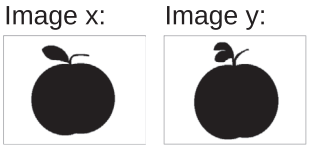


FIGURE 10 Context image for similar reference images

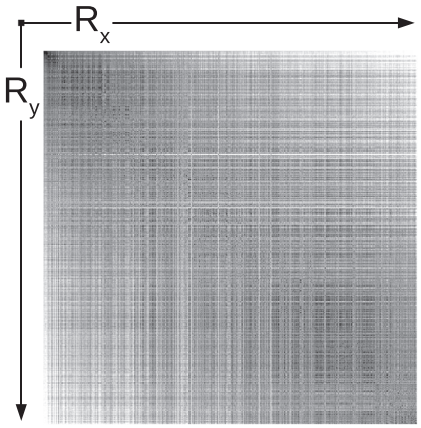
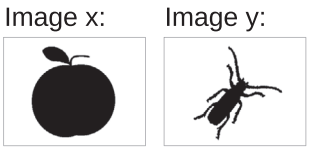


FIGURE 11 Dissimilar reference images



between two images $img_x, img_y \in \mathcal{C}$ considers only the relationship between them, their respective ranked lists, R_x, R_y, also include the distances from these images to all other collection images. Figure 8 illustrates the distance computation process.

3. *Normalization:* Let t be the current iteration. For all images $x, y \in \mathcal{C}$, set distances $A_{t+1}[x, y] = A_{t+1}[y, x] = \min(A_{t+1}[x, y], A_{t+1}[y, x])$.
4. *Ranked list resorting:* The image collection is reranked based on the new distances defined in terms of the similarity of ranked lists.

4.3 | The contextual re-ranking algorithm

The *contextual re-ranking* algorithm[22] is an approach that obtains contextual information based on context images. A *context image* is a grayscale image representation of distance matrices computed by CBIR descriptors. It contains information about all distances among images and also the spatial relationship between each query image and its ranked list. Figure 9 is an example of two similar images and Figure 10 shows its respective grayscale image representation. Figure 11 depicts the case where the images are not similar, resulting in the context image in Figure 12.

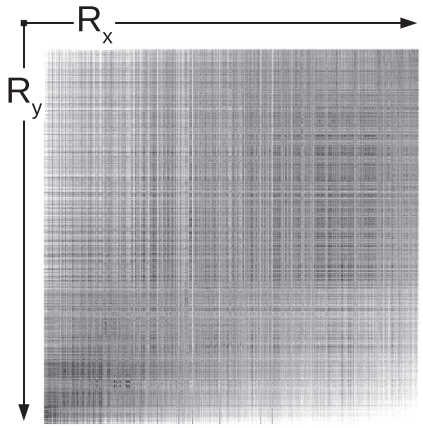


FIGURE 12 Context image for dissimilar reference images

This method implements the steps from Figure 3 as follows:

1. *Initialize data structures:* The distance matrix A is taken as input.
2. *Update distances:* The main idea of this re-ranking algorithm consists in processing the contextual information of a query image $img_x \in \mathcal{C}$ by constructing context images for this image and each one of its k-nearest neighbors. The results of the processed contextual information are stored in an affinity matrix W, which is an $N \times N$ matrix, where $W[a, b]$ represents the similarity between images img_a and img_b. Image processing techniques are then used to process the context images created. In particular, a median filter is applied to improve the quality of distance scores. The affinity matrix W is then updated and the same process is performed for all images in the collection.
3. *Normalization:* Let t be the current iteration. A new distance matrix A_{t+1} is computed as the inverse of the affinity matrix W. Afterward, for all images $x, y \in \mathcal{C}$, set distances $A_{t+1}[x, y] = A_{t+1}[y, x] = \min(A_{t+1}[x, y], A_{t+1}[y, x])$.
4. *Ranked list resorting:* The image collection is reranked based on the new distances $A_{t+1}[x, y]$.

4.4 | The Cartesian product of ranking references algorithm

The *Cartesian product of ranking references* (CPRR) algorithm[30] is a less-costly iterative re-ranking method, as it only considers the subset of images contained at the top-L positions of the ranked list. The main idea of this approach consists in maximizing the similarity information encoded in rankings through Cartesian product operations. The method considers the Cartesian product of both kNN and reverse kNN sets.

This method implements the steps from Figure 3 as follows:

1 *Initialize data structures:* The set of ranked lists with top-L positions are taken as input. A sparse similarity matrix W is initialized at each iteration. In addition, before the first iteration, the ranked lists are normalized by considering the reciprocal rank positions.

2 *Update distances:* equivalently to distances, the CPPR algorithm updates the similarities among images. First, a rank-similarity measure $r_k(q_i)$ is computed based on the position of image img_i in the ranked list of image img_q. Next, the Cartesian product is computed for the kNN and reverse kNN sets of each dataset image. The Cartesian product returns a set of pairs of images, where each pair (img_x, img_y) in a kNN set receives a similarity update defined as: $wc(x, y) = wc(x, y) + r_k(q, x) \times r_k(q, y)$.

3 *Normalization:* as Cartesian product operations are symmetric, no normalization is required.

4 *ranked list resorting:* The image collection is reranked based on the similarity scores updated by the Cartesian product operations.

5 | PARALLELIZATION OF RE-RANKING ALGORITHMS USING OPENCL

In this section, we propose a parallelization model to simplify the acceleration of the algorithms that belong to the class presented in Section 3. We also show how we used this model to implement the algorithms described in Section 4.

We start by briefly introducing the OpenCL standard and then, later, we describe its use in the parallelization of the considered re-ranking algorithms.

5.1 | OpenCL overview

OpenCL is an open industry standard introduced in 2009 for general purpose task-parallel and data-parallel programming of CPUs, GPUs, and other accelerators. It is a framework comprised of a language, an API, libraries, and a runtime system.

In OpenCL, a program is executed on a *device*, which can be a multicore CPU, a GPU, or another processor (e.g., DSPs and cell/B.E.). These devices typically contain one or more *compute units* (CUs), which in turn are composed of one or more virtual scalar processors, called *processing elements* (PEs), and *local memory*. PEs within a CU execute a single stream of instructions as either single instruction/multiple data (SIMD) units, executed in lock-step, or single program/multiple data (SPMD) units, allowing each PE to maintain its own program counter.

A *kernel* is a function declared in an OpenCL *program* and is executed on an OpenCL device. Kernels are dynamically compiled and scheduled for execution by a *command* sent to a *command-queue*. When a kernel is invoked on a device, an instance of its execution is called a *work-item*. Work-items are executed by one or more PEs as part of a *work-group* executing on a CU[56] ND-ranges are N-dimensional index spaces (where N is one, two, or three) to which OpenCL maps all work-items to be launched. It is possible to specify how these mapped work-items are divided into work-groups.[56,57]

Besides these terms defined by the Khronos Group, AMD also introduced the concept of *wavefronts* as groups of work-items executed in lock-step on a CU. They form work-groups and having the size of a work-group be a multiple of the size of its wavefronts benefits the performance of the program.[57]

5.2 | Parallel implementation of re-ranking algorithms

Given the importance of efficiency in real-world scenarios, we propose a model that facilitates the parallelization of the class of re-ranking algorithms considered in our study. Even though the presented parallelization strategy requires the use of synchronization mechanisms, it is still a favorable compromise between the performance and the simplicity of the implementation, as it shows efficiency gains (discussed in Sections 6 and 7) and provides a straightforward way to divide the main steps of iterative re-ranking algorithms into different independently parallelizable kernels.

The model is illustrated in Figure 1 and in the upper part of Figure 13, named "general parallelization model."

The two more computationally intensive steps, "update distances/similarities" and "ranked list resorting," are a good fit for parallelization techniques, since they consist in operations that can be performed on independent data. For example, in the first step, the calculation of the distance between a pair of images does not affect other calculations within the same iteration, and in the second, each ranked list can be independently resorted.

The lower part of Figure 13, named "re-ranking algorithms fitting the model," illustrates how each re-ranking algorithm considered in our study is parallelized according to the model. Sections 5.2.2 and 5.2.3 give further details about how the main steps were parallelized.

5.2.1 | Modeling the OpenCL kernels

The first relevant choice required for designing parallel solutions using OpenCL consists in modeling the kernels. The simplest solution would be designing the re-ranking algorithm in a single OpenCL kernel, repeatedly executed at each iteration. However, some steps must be completed in a predefined order. For example, ranked lists can only be resorted after the distances/similarities among images have been recomputed. Consequently, barriers must be enforced to ensure the correctness of data dependencies between different steps of the algorithm.

Although barriers are available in OpenCL, they only provide synchronization among commands in command-queues and work-items in the same work-group. In order to obtain synchronization among work-groups (which we refer to as "global synchronization"), it is necessary to either use atomic operations in global memory or separate operations into different kernels.[56] We chose the latter option and designed the parallel re-ranking algorithms using two different kernels, one for each step.

Between the execution of the two kernels, a straightforward normalization process is required. Since this also depends on all distances being updated beforehand, computing this minor step inside the first kernel would require a barrier for global synchronization. Considering that this process presents low computational cost, creating a new kernel for this operation may not be profitable due to the introduction of overhead. Therefore, we can simply compute this normalization step using a serial implementation that runs on the CPU host device.

The same occurs with the initialization of the data structures used in the algorithms, such as the matrix that stores the distances/similarities at the beginning of each iteration. We leave the evaluation of different approaches for implementing these serial steps as future work.

5.2.2 | Implementation of the "update distances/similarities" step

This section presents how the first step, which is related to the task of updating distances/similarities, was parallelized for each re-ranking algorithm discussed in this article. In each iteration of the algorithms, the "update distances/similarities" step calculates new distances/similarities among the

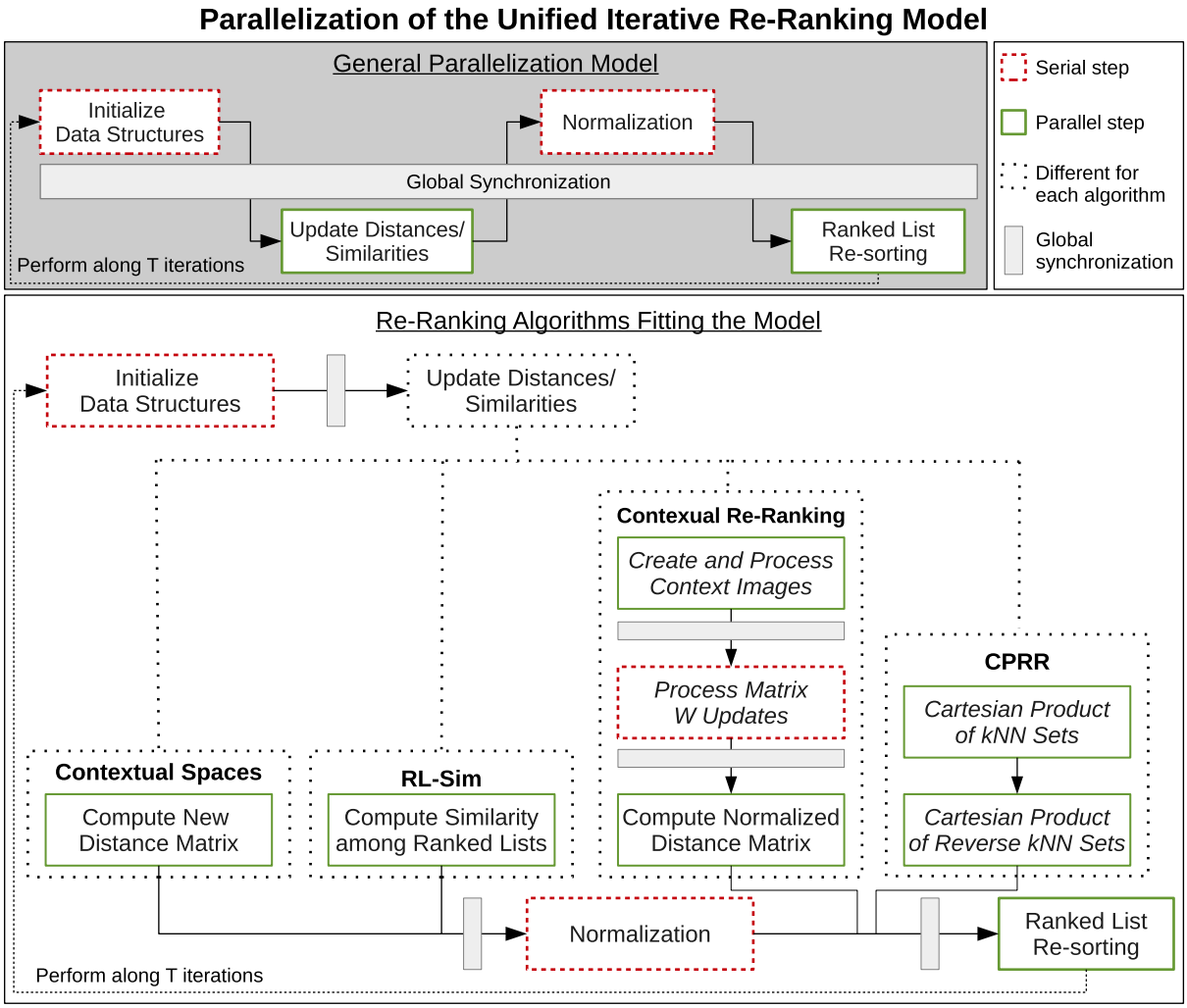


FIGURE 13 Overview of the division of steps for parallelizing the unified iterative re-ranking model and how re-ranking algorithms fit the parallelization structure

collection images, generating new values that are used as input for the next iteration. We refer to the number of images in the collection being used as N.

Contextual spaces re-ranking

Let k be the number of images used to create the contextual space of a query image in a given iteration of the contextual spaces re-ranking algorithm.

Considering the distance between the query and each of its k-nearest neighbors, the "update distances/similarities" kernel of this method calculates the new distance between a pair of collection images. Since each of these computations can be performed independently, we have a two-dimensional *ND-range* with $N \times N$ work-items executing this kernel. This division fits well the capabilities of devices that have many computational cores, such as GPUs.

Furthermore, a few optimizations were implemented to improve the performance of the execution on GPUs, for example, changing the way the kernel accesses the ranked lists matrix to obtain the k-nearest neighbors.

By setting each ranked list in a column (instead of the usual approach of having one list per row), a work-item x that attempts to access the kth element of its ranked list can benefit from the fact that a work-item y may have recently accessed the same position of a nearby list, as shown in Figure 14. This happens because the value that x needs is in the same cache line as a value that was recently accessed and thus will already be cached when x tries to read it. Exploiting this type of coalesced access greatly increases the cache hit ratio, leading to better execution times.

RL-Sim re-ranking

Let ψ be the function that represents the similarity of ranked lists, which is used to calculate new distances, and λ be the number of images in each ranked list that is considered when the distances are redefined.[9]

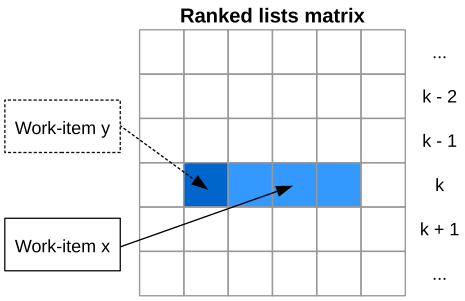


FIGURE 14 Work-item *y* accesses the dark blue position of row *k*, bringing the values in the light blue positions to the cache memory as well. Work-item *x* subsequently accesses a value that is already cached

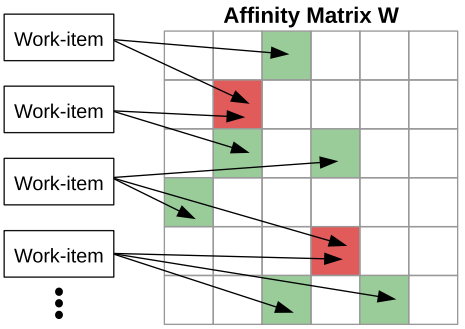


FIGURE 15 Increments being directly made on matrix *W*. Some updates made to positions altered by more than one work-item are lost

The "update distances/similarities" kernel of the RL-Sim re-ranking algorithm is responsible for computing the distances between one image and all other collection images. This kernel is executed by $N \times \lambda$ work-items divided into a two-dimensional OpenCL *ND-range*. Every work-item computes the function ψ between the current image and one of the images at the top λ positions of each of the N ranked lists. This kernel division aims at avoiding divergent control flows among work-items, as this can cause GPUs to be underutilized and result in a performance decrease of the execution in these devices.

A total of $N^2 \times \lambda$ comparisons are made per iteration. This number becomes considerably large when dealing with extensive image collections and, since the $N \times \lambda$ operations done for an image do not depend on the calculations done for the others, this parallelization approach is applicable.

Contextual re-ranking

In order to calculate new distances, the contextual re-ranking algorithm must first create and process context images for each collection image being used as the query. Consider that *k* is the number of context images created for each query image in a given iteration and that *W* is the affinity matrix that stores the result of the contextual information.

Since *W* needs to be completely computed before starting the calculation of the new distances, we decided to model the "update distances/similarities" step of this algorithm as two OpenCL kernels with a simple serial operation in between. As explained in Section 5.2.1, this division allows global synchronization between the different parts of the computation.

For each of the query's *k*-nearest neighbors, the first kernel constructs a context image, applies image processing techniques (thresholding and filtering), and obtains the resulting black pixels. Based on these pixels, it computes the increment values that are later used to update *W*. This kernel is executed by a one-dimensional *ND-range* with *N* work-items.

The performance of this kernel was optimized through the use of direct updates to the matrix *W* (as opposed to the use of synchronized calculations stored on a temporary matrix). We note that concurrently accessing the same elements of *W*, as illustrated in Figure 15, may lead to the loss of some of the increments. Global synchronization mechanisms could be used to prevent these losses; however, experiments show that, due to the small number of conflicts, this does not significantly affect the effectiveness of the re-ranking algorithm and the lack of synchronization greatly improves the overall performance.[27] Nevertheless, this procedure creates an element of nondeterminism, leading to the possibility of different ranked lists being created in each execution.

The serial operation executed after the first kernel consists in simply determining the maximum distance value between two images in the distance matrix. This value is then used for normalization purposes when computing the new distances.

The task that remains for the second kernel is the actual computation of the new distance matrix. In the same way as the previously described algorithms, each value of the new matrix can be independently computed. However, unlike the other methods, the contextual re-ranking algorithm computes the distance matrix based on the values in *W*, requiring an additional kernel. Still, the computation is quite straightforward, allowing us to perform a few more optimizations.

Let img_x and img_y be images in the collection and $\rho(img_x, img_y)$ be the distance from img_x to img_y. The kernel is responsible for calculating both $\rho(img_x, img_y)$ and $\rho(img_y, img_x)$ for all $y > x$ and then normalizing these distances. In this scenario, instead of two dimensions and $N \times N$ work-items (one for each position of the matrix), we use a single dimension with N work-items. To accommodate this implementation, the proposed model is adapted to not consider the serial normalization step afterward.

Even though the kernels presented in this subsection are parallelized through the execution of N work-items each, which is arguably less than some of the previously discussed methods, the optimizations made in combination with the fact that N is a large number for big image collections favor this parallelization approach.

Cartesian product of ranking references

The "update distances/similarities" step of the CPRR algorithm updates similarity scores instead of distances among images. This particularly improves the performance of this algorithm on larger datasets, as only nonzero similarity values are considered by other steps of the method. All the similarity updates are derived from Cartesian product operations and stored in a sparse matrix.

The Cartesian product operations are performed over kNN and reverse kNN sets. The sets are obtained for each collection image, resulting in a list of pairs of images, which should have their similarity updated. Given that the computation of Cartesian product operations to obtain the pairs is independent among different sets, the CPRR algorithm can be widely parallelized. To this end, two kernels are used: one to process the kNN sets and another for the reverse kNN sets. In both cases, an one-dimensional *ND-range* is used with N work-items.

Although the Cartesian product operations are independently processed in parallel, the updated similarity measure is stored in the global memory. Therefore, concurrent accesses can cause loss of updates. As the overhead associated with global synchronization mechanisms and atomic operations in OpenCL is high, we employed a relaxed concurrence model similar to what was described for the contextual re-ranking algorithm. Direct updates of the similarity score are allowed, even when causing the loss of updates. Experimentally, we assessed that the impact of such losses on the effectiveness is very low, while significant improvements are observed in terms of performance.[30]

5.2.3 | Implementation of the "ranked list resorting" step

Since all algorithms described in Section 4 perform the same operations in the second step (resorting ranked lists), it is possible to use the same approach for all of them when designing the corresponding kernel.

Let N be the number of images in a collection. The "ranked list resorting" step resorts N ranked lists (one for each collection image being used as the query). Considering that there is no dependency between the computations done for each ranked list, we can execute each sorting operation in parallel. A kernel that sorts one ranked list is used for this and the execution uses an *ND-range* with a single dimension containing N work-items.

Due to the fact that the impact of the sorting step on ranked lists is usually small (i.e., most of the changes occur only in the beginning of the ranked lists), we use the insertion sort algorithm, which performs well when the input is almost sorted. In this situation, insertion sort can overcome other efficient sorting algorithms.[58]

Although this approach has good results for CPUs, an algorithm specifically designed to run on GPUs must be chosen to improve the performance when using these devices. Nevertheless, this study is out of the scope of this article and the evaluation of this step in Section 6 focuses only on the comparison between serial and parallel times for the implementations running on CPUs.

6 | PERFORMANCE

This section presents the results of the performance experiments conducted to assess the impact of the parallel strategies proposed in Section 5. We compare segments corresponding to each step of the re-ranking algorithms considering executions in C/C++ and OpenCL.

In several occasions throughout this section, we abbreviate the kernel names in order to refer to them more easily. The "ranked list resorting" kernel is simply named "sort," the "process context images" kernel is named "image," and the "update distances/similarities" kernel is referred to as "update."

6.1 | Experimental setup

We executed tests on four different machines. For simplicity, these systems are named APU, FirePro, HD7950, and R9-290x. All test machines have the AMD APP SDK 2.9.1 and OpenCL 1.2. The remaining software environment installed in each of them is described in Table 1.

Table 2 presents the hardware specifications that we consider relevant to compare the machines in the scope of our experiments. We note that the machines FirePro, HD7950, and R9-290x have discrete GPUs, while the machine APU contains an integrated GPU.

TABLE 1 Software environment for each of the test machines

Name	Operating system
APU	Linux 3.16.0-33-generic Ubuntu 14.04.2
FirePro	Linux 3.16.0-31-generic Ubuntu 14.04.2
HD7950	Linux 3.11.0-15-generic Ubuntu 12.04.4
R9-290x	Linux 3.13.0-24-generic Ubuntu 14.04

TABLE 2 Hardware environment for each of the test machines

Name	CPU	CPU cores	RAM	GPU	Shaders	GPU memory
APU	AMD A8-3850 APU @ 2.90 GHz	Four physical	32 GB	Radeon HD 6550D	400	512 MB DDR3
FirePro	Intel Core i7-3770 @ 3.40 GHz	Four physical (eight logical)	32 GB	ATI FirePro V7800	1440	2 GB GDDR5
HD7950	Intel Xeon E3-1240 v3 @ 3.40 GHz	Four physical (eight logical)	24 GB	AMD Radeon HD 7950	1792	3 GB GDDR5
R9-290x	Intel Xeon E5-2630 v2 @ 2.60 GHz	Six physical (12 logical)	32 GB	AMD Radeon R9 290x	2816	4 GB GDDR5

TABLE 3 Notation used in Section 6.2 to represent kernel implementations

Notation	Meaning
s-CPU	OpenCL implementation of the "sort" kernel running on the CPU
s-GPU	OpenCL implementation of the "sort" kernel running on the GPU
s-Serial	C/C++ implementation of the "sort" kernel
i-CPU	OpenCL implementation of the "image" kernel running on the CPU
i-GPU	OpenCL implementation of the "image" kernel running on the GPU
i-Serial	C/C++ implementation of the "image" kernel
u-CPU	OpenCL implementation of the "update" kernel running on the CPU
u-GPU	OpenCL implementation of the "update" kernel running on the GPU
u-Serial	C/C++ implementation of the "update" kernel

For our main experimental analysis, we used a well-known shape database called MPEG-7,[54] which is commonly used in the evaluation of CBIR re-ranking algorithms and has 1400 shapes divided into 70 classes. The CFD shape descriptor,[55] which presents significant effectiveness gains for various re-ranking methods, was used to compare the collection images.

6.2 | Experimental results

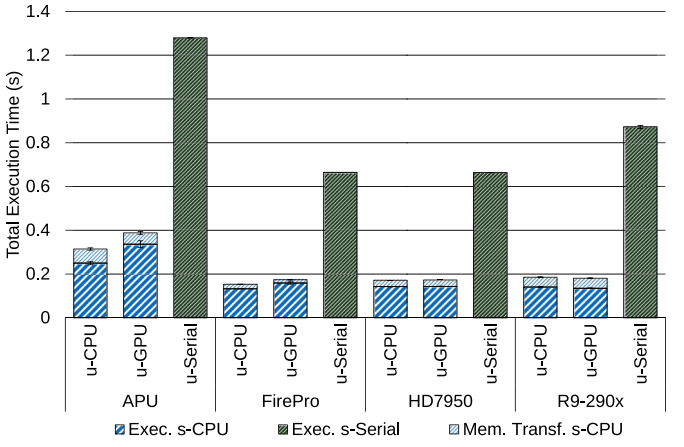
We executed a set of experiments to evaluate the performance of the re-ranking algorithms implemented in OpenCL in comparison with the serial implementations in C/C++. The C/C++ code was compiled using g++ with the -O3 flag and we measured the run times by calculating the average of 10 executions with corresponding 95% confidence intervals. The notation shown in Table 3 is used in this subsection to represent which implementation of a kernel was used in a certain test case.

As noted in Section 5.2.3, the graphs included in this section do not display the information relative to the execution of the "sort" kernels on GPUs. Since memory transfer times for serial executions are always zero, this information is also not included in the graphs.

Tables containing the speedups obtained on the parallel test cases are presented below each graph. Each table line describes a different test case, and the underlined kernels in the first column represent the kernel or kernel combination for which the speedup is being analyzed.

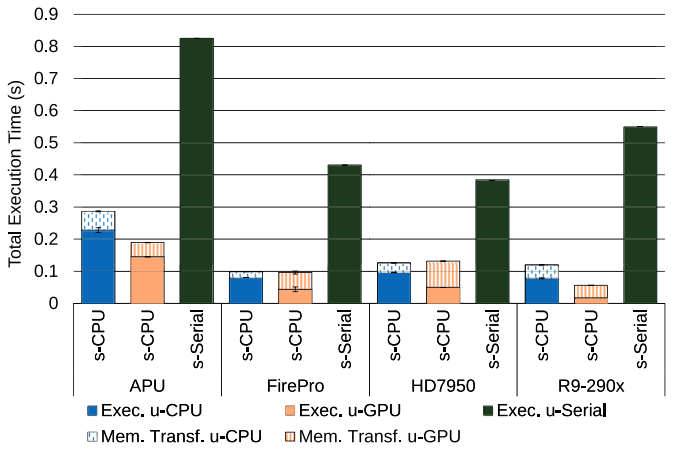
6.2.1 | Contextual spaces re-ranking algorithm

Figure 16 shows the results for the "sort" kernel of the contextual spaces re-ranking algorithm. For each test machine, three total execution times are presented: two execution and memory transfer times for the OpenCL version of the "sort" kernel running on the CPU (Exec. s-CPU + Mem.



Test Case	Speedup APU	Speedup FirePro	Speedup HD7950	Speedup R9-290x
s-CPU/u-CPU	4.1 ×	4.3 ×	3.9 ×	4.7 ×
s-CPU/u-GPU	3.3 ×	3.8 ×	3.8 ×	4.8 ×

FIGURE 16 Comparison between total execution times for the "ranked list resorting" kernel of the contextual spaces re-ranking algorithm



Test Case	Speedup APU	Speedup FirePro	Speedup HD7950	Speedup R9-290x
s-CPU/u-CPU	2.9 ×	4.4 ×	3.1 ×	4.6 ×
s-CPU/u-GPU	4.4 ×	4.5 ×	2.9 ×	9.8 ×

FIGURE 17 Comparison between total execution times for the "update distances" kernel of the contextual spaces re-ranking algorithm

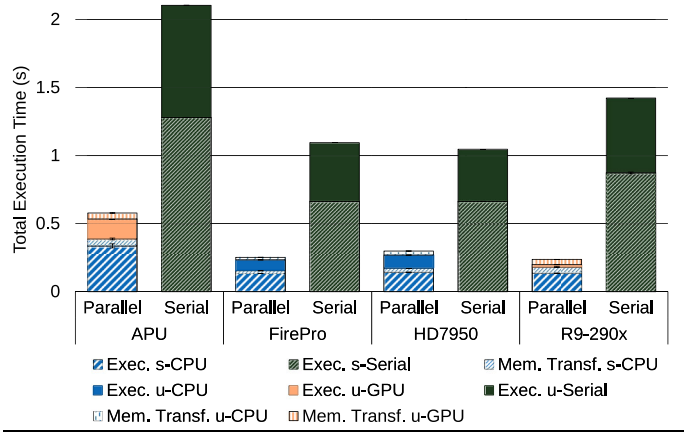
Transf. s-CPU) and one execution time for the C/C++ version of the "sort" kernel (Exec. s-Serial). One of the Exec. s-CPU + Mem. Transf. s-CPU bars is labeled u-CPU and corresponds to the s-CPU/u-CPU test case (both OpenCL "sort" and "update" kernels executed on the CPU), while the other is labeled u-GPU and refers to the s-CPU/u-GPU test case (OpenCL "sort" kernel executed on the CPU and OpenCL "update" kernel executed on the GPU).

Taking into account both the execution and memory transfer times of the parallel implementations, we see that our approach leads to good speedups on all machines. Several factors, such as caching, may interfere with the execution time of the kernels, possibly explaining the differences between the test cases. We intend to further investigate the causes of this execution time variation in future studies.

The execution time comparison for the "update" kernel is shown in Figure 17. Again, all parallelizations resulted in improved kernel performance. Since the operations of this kernel are not costly and can be easily parallelized, considerably better results are expected from the GPU executions. However, we note that this is not the case for two of the machines. Although the execution time for HD7950 was better, this machine presented a high memory transfer time, leading to an overall smaller speedup. In the case of FirePro, the speedup is better only by a slight margin. This could be explained by the fact that the this machine's CPU is a more recent model than the GPU.

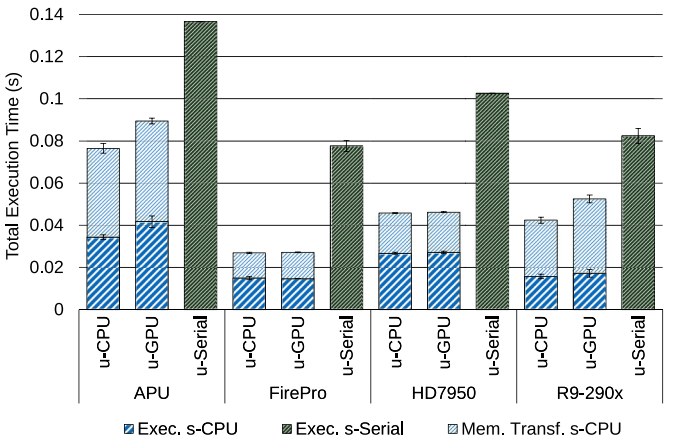
The best kernel combinations are presented in Figure 18. For the APU and R9-290x machines, the best combination of OpenCL kernels is when "sort" is running on the CPU and "update" is on the GPU. On the other hand, for the FirePro and HD7950 machines, the best combination of OpenCL kernels is when both "sort" and "update" are executed on the CPU. As we can see, the total execution time of the parallelized version is faster than a single serial kernel, even when memory transfer times are considered.

FIGURE 18 Comparison between total execution times for the best kernel combinations of the contextual spaces re-ranking algorithm



Test Case	Speedup APU	Speedup FirePro	Speedup HD7950	Speedup R9-290x
s-CPU/u-CPU	-	4.3 ×	3.5 ×	-
s-CPU/u-GPU	3.6 ×	-	-	6.0 ×

FIGURE 19 Comparison between total execution times for the "ranked list resorting" kernel of the RL-Sim re-ranking algorithm



Test Case	Speedup APU	Speedup FirePro	Speedup HD7950	Speedup R9-290x
s-CPU/u-CPU	1.8 ×	2.9 ×	2.2 ×	1.9 ×
s-CPU/u-GPU	1.5 ×	2.9 ×	2.2 ×	1.6 ×

6.2.2 | RL-Sim re-ranking algorithm

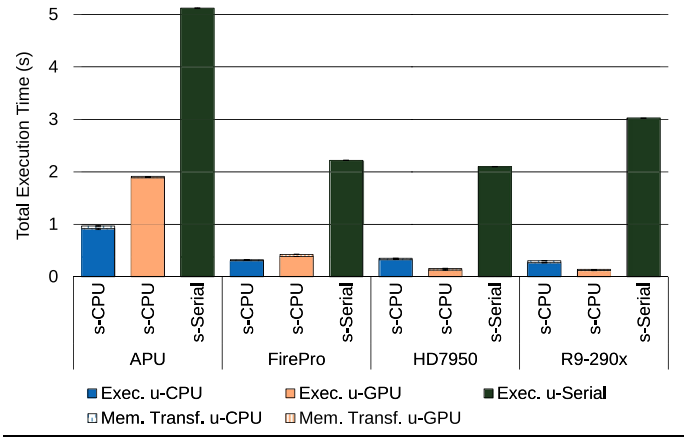
Figure 19 illustrates the results for the "sort" kernel of the RL-Sim re-ranking algorithm. Although its implementation is similar to the kernel from contextual spaces re-ranking, we see that the results present some differences in comparison with Figure 16. This is due to the fact that fewer changes in the ranked lists are required in this step in comparison with the previous algorithm, leading to the kernel running up to 10 times faster in RL-Sim. In this scenario, memory transfer times are much more prominent, since less computation is being performed. Still, the parallelization of this kernel results in positive speedups.

Figure 20 shows the results for the "update" kernel. This kernel does a lot more computation than the previous one and this becomes evident by comparing their serial running times. As mentioned before, a possible explanation for the poor GPU speedup for FirePro in comparison with the CPU is the difference in hardware generation. Further studies are needed to explain the performance of the GPU on the APU machine, but we speculate that the work-item division used might not favor this hardware, as it has less shader processing units and less GPU memory than the other machines.

Since we obtained good speedups for the second kernel and it dominates the execution time, the best kernel combinations yield good speedups as well, as displayed in Figure 21. We see that the best OpenCL kernel combinations for the APU and FirePro machines happen when both "sort" and "update" are being executed on the CPU. As for HD7950 and R9-290x, the best OpenCL kernel combination is the "sort" kernel running on the CPU and the "update" kernel on the GPU.

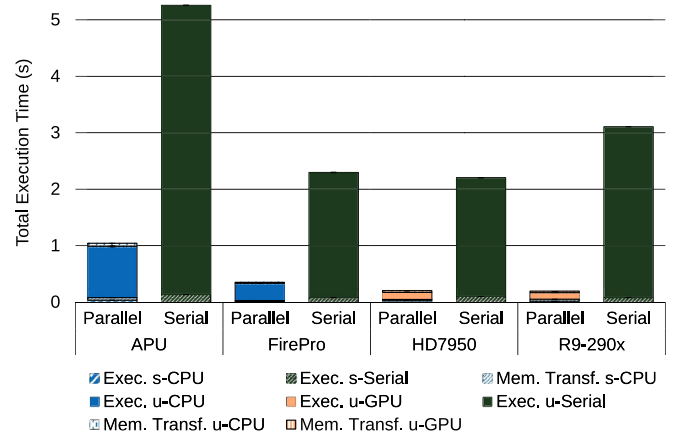
6.2.3 | Contextual re-ranking algorithm

By analyzing Figure 22, we see that the case of the "sort" kernel of the contextual re-ranking algorithm is analogous to what was observed for the same kernel in the RL-Sim re-ranking algorithm. This time, the kernels run up to almost six times faster than they did for contextual spaces re-ranking,



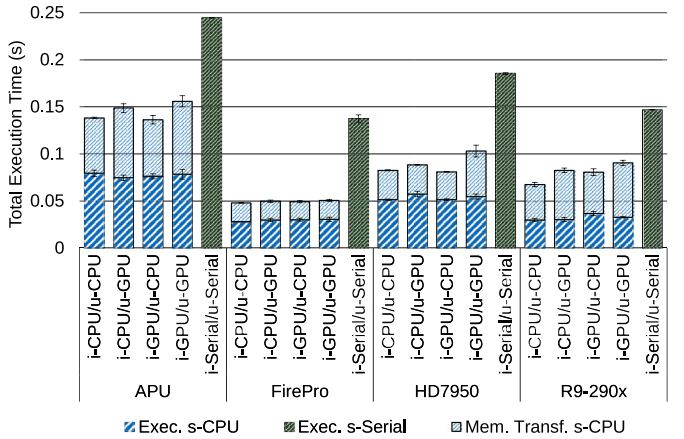
Test Case	Speedup APU	Speedup FirePro	Speedup HD7950	Speedup R9-290x
s-CPU/u-CPU	5.3 ×	6.9 ×	5.9 ×	10.0 ×
s-CPU/u-GPU	2.7 ×	5.2 ×	13.5 ×	21.5 ×

FIGURE 20 Comparison between total execution times for the "update distances" kernel of the RL-Sim re-ranking algorithm



Test Case	Speedup APU	Speedup FirePro	Speedup HD7950	Speedup R9-290x
s-CPU/u-CPU	5.0 ×	6.6 ×	-	-
s-CPU/u-GPU	-	-	10.9 ×	16.1 ×

FIGURE 21 Comparison between total execution times for the best kernel combinations of the RL-Sim re-ranking algorithm



Test Case	Speedup APU	Speedup FirePro	Speedup HD7950	Speedup R9-290x
s-CPU/i-CPU/u-CPU	1.8 ×	2.9 ×	2.2 ×	2.2 ×
s-CPU/i-CPU/u-GPU	1.6 ×	2.8 ×	2.1 ×	1.8 ×
s-CPU/i-GPU/u-CPU	1.8 ×	2.8 ×	2.3 ×	1.8 ×
s-CPU/i-GPU/u-GPU	1.6 ×	2.7 ×	1.8 ×	1.6 ×

FIGURE 22 Comparison between total execution times for the "ranked list resorting" kernel of the contextual re-ranking algorithm

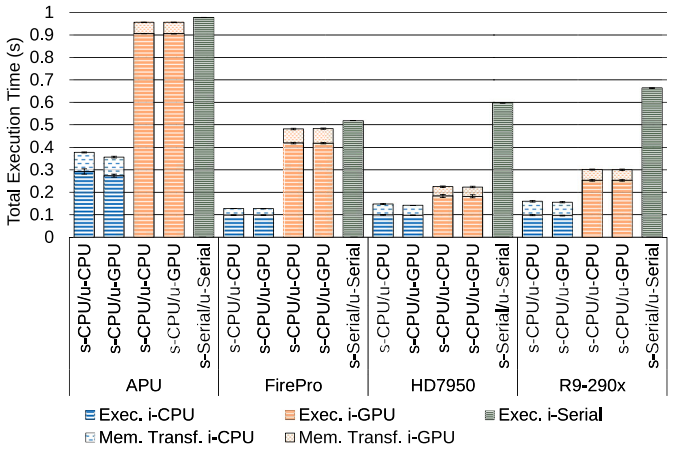
and once more the faster kernel execution time increases the impact of memory transfers on the total execution time of the operation. Nevertheless, the parallelization presents positive speedups when compared with the serial execution.

Figure 23 displays the results for the "image" kernel. The lack of performance of the GPU executions in comparison with the CPU could be explained by the fact that this kernel contains several control flow statements. In the OpenCL GPU model of parallelism, a single instruction is executed over all work-items in a wavefront in parallel. If work-items within a wavefront diverge, all paths are executed serially and the total time to execute the branch is the sum of each path time.[57]

The results for the "update" kernel are presented in Figure 24. Similar to the case of the "sort" kernel that is depicted in Figure 22, the amount of computation performed is small, which leads to memory transfer playing a more prominent role in the kernel's total execution time. We see that for almost all cases, this causes the total execution time for the parallelized versions to be larger than the serial time.

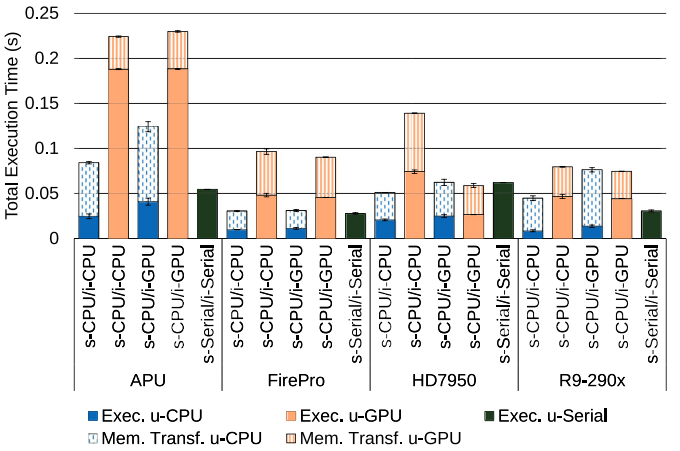
We leave the study of techniques that improve memory transfer time as future work. However, it is worth mentioning that by looking only at the execution times, it is possible to draw a few conclusions about the parallelization of this kernel. We explore this in Table 4, which shows the speedups considering only the execution times. The results indicate that the chosen approach, although leading to reasonable speedups on the CPU, does not favor GPU utilization. One possible reason for this is the fact that by giving different loads to each work-item, we balance the work distribution better on the CPU, but also create divergent paths, which cause the GPU's resources to be underutilized.

Overall, the "update" kernel represents a minor part of the total execution time of the contextual re-ranking algorithm, so it is possible to see in Figure 25 that the best kernel combination gives us performance gains. We observe that this is mainly due to the results obtained for the "image"



Test Case	Speedup APU	Speedup FirePro	Speedup HD7950	Speedup R9-290x
s-CPU/i-CPU/u-CPU	2.6 ×	4.1 ×	4.1 ×	4.2 ×
s-CPU/i-CPU/u-GPU	2.7 ×	4.1 ×	4.2 ×	4.3 ×
s-CPU/i-GPU/u-CPU	1.0 ×	1.1 ×	2.7 ×	2.2 ×
s-CPU/i-GPU/u-GPU	1.0 ×	1.1 ×	2.7 ×	2.2 ×

FIGURE 23 Comparison between total execution times for the "process context images" kernel of the contextual re-ranking algorithm



Test Case	Speedup APU	Speedup FirePro	Speedup HD7950	Speedup R9-290x
s-CPU/i-CPU/u-CPU	0.6 ×	0.9 ×	1.2 ×	0.7 ×
s-CPU/i-CPU/u-GPU	0.2 ×	0.3 ×	0.4 ×	0.4 ×
s-CPU/i-GPU/u-CPU	0.4 ×	0.9 ×	1.0 ×	0.4 ×
s-CPU/i-GPU/u-GPU	0.2 ×	0.3 ×	1.1 ×	0.4 ×

FIGURE 24 Comparison between total execution times for the "update distances" kernel of the contextual re-ranking algorithm

TABLE 4 Speedups considering only the execution times from the "update distances" kernel of the contextual re-ranking algorithm

Test Case	Speedup APU	Speedup FirePro	Speedup HD7950	Speedup R9-290x
s-CPU/i-CPU/u-CPU	2.2×	2.8×	3.0×	3.6×
s-CPU/i-CPU/u-GPU	0.3×	0.6×	0.8×	0.7×
s-CPU/i-GPU/u-CPU	1.3×	2.5×	2.5×	2.3×
s-CPU/i-GPU/u-GPU	0.3×	0.6×	2.3×	0.7×

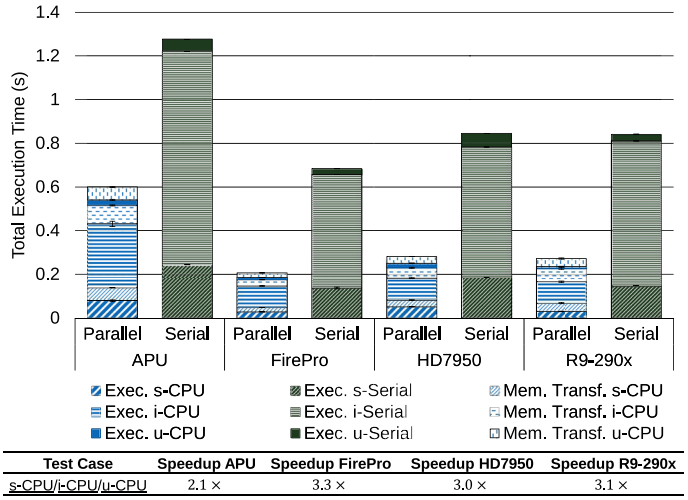


FIGURE 25 Comparison between total execution times for the best kernel combinations of the contextual re-ranking algorithm

TABLE 5 Hardware and software environments for the test machines used in the scalability evaluation

Name	Operating system	CPU	CPU cores	RAM	GPU	Shaders	GPU memory
CPU-Machine	Linux 4.4.0-59-generic Ubuntu 14.04.5 LTS	Intel Xeon E5-2620 v2 @ 2.10 GHz	6 physical (24 logical)	24 GB	—	—	—
GPU-Machine	Linux 4.4.0-1098-aws Ubuntu 16.04.6 LTS	Intel Xeon E5-2686 v4 @ 2.30 GHz	2 physical (4 logical)	60 GB	Tesla K80	2496	12 GB GDDR5

kernel, which occupies most of the execution time. On all test machines, the best OpenCL kernel combination is that of the case where the "sort," "image," and "update" kernels are all running on the CPU.

7 | LARGER DATASETS

The use of re-ranking approaches on image retrieval scenarios relies not only on effectiveness and efficiency but also on their ability to handle larger datasets. Therefore, this section evaluates the performance of the proposed parallelization model on a collection of datasets with increasingly growing sizes. Section 7.1 describes the environment and datasets considered and Section 7.2 presents the experimental results.

In several occasions throughout this section, we abbreviate the kernel names in order to refer to them more easily. The "ranked list resorting" kernel is simply named "sort," the "Cartesian product of reverse *k*NN sets" kernel is named "rev," the "Cartesian product of *k*NN sets" kernel is named "prod," and the "update distances/similarities" kernel is referred to as "update."

7.1 | Experimental setup

For the experiments presented in this section, we considered two different machines. One of them was used for the serial and CPU executions and the other only for the GPU executions. Table 5 presents the hardware and software specifications for both of them. The GPU-machine corresponds to an Amazon web services deep learning AMI instance.

TABLE 6 Datasets considered in the experimental evaluation

Dataset	Size	Type	Descriptor	General description
MPEG-7[54]	1400	Shape	Contour features descriptor (CFD)[55]	Composed of 1400 shapes divided in 70 classes. Commonly used for evaluation of postprocessing methods
Corel-5K[60]	5000	Objects/scenes	50-layers residual neural network (CNN-ResNet)[61]	Composed of 50 categories with 100 images for each class, including diverse scene content such as fireworks, bark, microscopy images, tiles, trees, etc
Corel-10K[60]	10,000	Objects/scenes	50-layers residual neural network (CNN-ResNet)[61]	Composed of 100 categories with 100 images for each class, including diverse scene content such as fireworks, bark, microscopy images, tiles, trees, etc
ALOI[62]	72,000	Objects	Local color histograms (LCH)[63]	Images from 1000 classes of objects, with different viewpoint, occlusion, and illumination conditions

TABLE 7 Notation used in Section 7.2 to represent kernel implementations

Notation	Meaning
s-CPU	OpenCL implementation of the "sort" kernel running on the CPU
s-GPU	OpenCL implementation of the "sort" kernel running on the GPU
s-Serial	C/C++ implementation of the "sort" kernel
r-CPU	OpenCL implementation of the "rev" kernel running on the CPU
r-GPU	OpenCL implementation of the "rev" kernel running on the GPU
r-Serial	C/C++ implementation of the "rev" kernel
p-CPU	OpenCL implementation of the "prod" kernel running on the CPU
p-GPU	OpenCL implementation of the "prod" kernel running on the GPU
p-Serial	C/C++ implementation of the "prod" kernel
u-CPU	OpenCL implementation of the "update" kernel running on the CPU
u-GPU	OpenCL implementation of the "update" kernel running on the GPU
u-Serial	C/C++ implementation of the "update" kernel

Table 6 presents the datasets used in the experiments. We evaluated our approach considering datasets of up to 72,000 images, which encompass a wide variety of pictures and different scenarios. For each dataset, different descriptors were considered with their default parameters. The convolutional neural network (CNN) descriptors were trained on the ImageNet[59] dataset.

7.2 | Experimental results

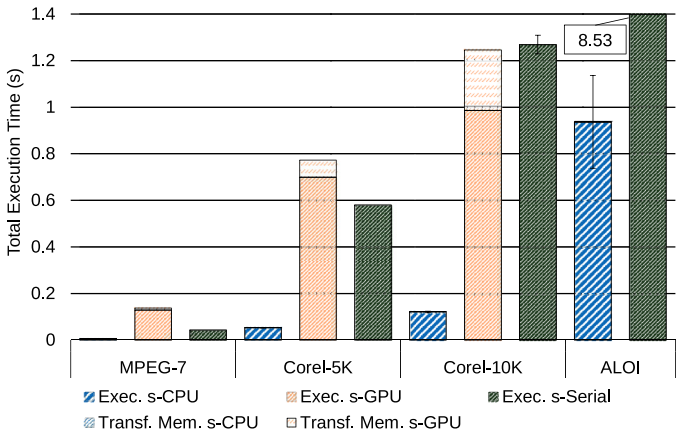
We executed a set of experiments to evaluate the performance of the CPRR algorithm implemented in OpenCL in comparison with its serial implementations in C/C++. The C/C++ code was compiled using g++ with the -O3 flag and we measured the run times by calculating the average of 10 executions with corresponding 95% confidence intervals. The ALOI dataset was too large for the available GPU memory, therefore only CPU and serial total execution times are shown for these tests. The notation displayed in Table 7 is used in this subsection to represent which implementation of a kernel was used in a certain test case.

Figure 26 shows the results for the "sort" kernel of the Cartesian product of ranking references algorithm. For each dataset, three total execution times are presented: two execution and memory transfer times for the OpenCL version of the "sort" kernel running on the CPU (Exec. s-CPU + Mem. Transf. s-CPU) and one execution time for the C/C++ version of the "sort" kernel (Exec. s-Serial). In all cases, all kernels were executed on the same device, that is, all kernels were executed on the CPU for OpenCL CPU tests and all kernels were executed on the GPU for OpenCL GPU tests.

As discussed in Section 5.2.3, the implementation of this kernel is not optimized for GPUs, explaining the GPU performance results. Nonetheless, we see that the CPU parallelization achieved good speedups when compared with the serial implementation.

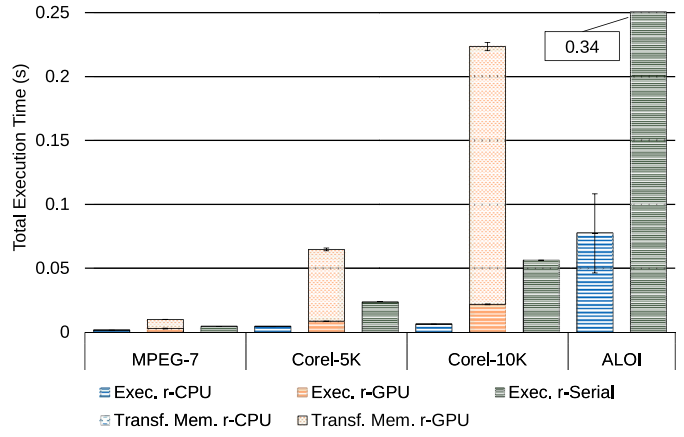
We can also compare the total execution time of the parallel CPU implementation running on each dataset. For the Corel-5K dataset, it increased by 10.2× when compared with that of MPEG-7, the total execution for Corel-10K took 2.3× as long as the one for Corel-5K, and the total execution time for ALOI was 7.8× that of Corel-10K. Still, even considering these increases, we see that our parallelization approach allowed us to execute this kernel in under 1.14 seconds in the worst case for the largest tested dataset.

Figure 27 has the results for the "rev" kernel. We observe that, although GPU execution times were the lowest ones among our results, high memory transfer times lead to this device not being the most suitable for the parallelization of this kernel. On the other hand, CPU parallelization again presented good speedups, with the total execution time of this kernel being less than 0.11 seconds in the worst case for the largest tested dataset.



Test Case	Speedup MPEG-7	Speedup Corel-5K	Speedup Corel-10K	Speedup ALOI
s-CPU/r-CPU/p-CPU/u-CPU	8.3 ×	11.0 ×	10.5 ×	9.1 ×
s-GPU/r-GPU/p-GPU/u-GPU	0.3 ×	0.8 ×	1.0 ×	–

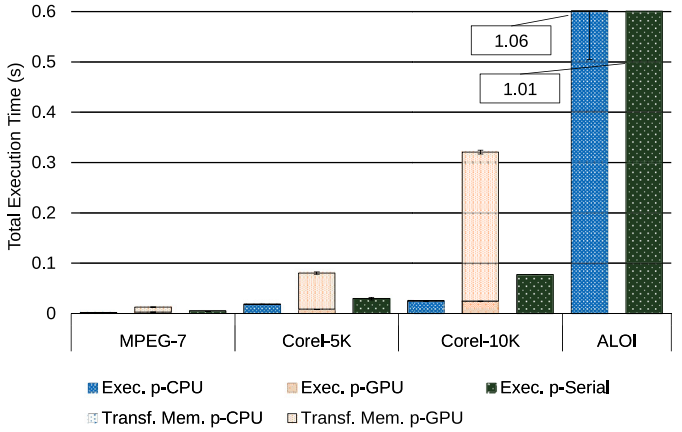
FIGURE 26 Comparison between total execution times for the "ranked list resorting" kernel of the Cartesian product of ranking references algorithm



Test Case	Speedup MPEG-7	Speedup Corel-5K	Speedup Corel-10K	Speedup ALOI
s-CPU/r-CPU/p-CPU/u-CPU	3.3 ×	5.4 ×	9.1 ×	4.4 ×
s-GPU/r-GPU/p-GPU/u-GPU	0.5 ×	0.4 ×	0.3 ×	–

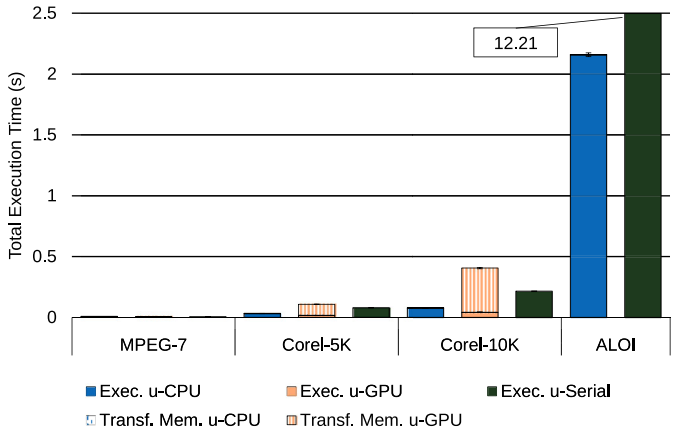
FIGURE 27 Comparison between total execution times for the "Cartesian product of reverse *k*NN sets" kernel of the Cartesian product of ranking references algorithm

FIGURE 28 Comparison between total execution times for the "Cartesian product of *k*NN sets" kernel of the Cartesian product of ranking references algorithm



Test Case	Speedup MPEG-7	Speedup Corel-5K	Speedup Corel-10K	Speedup ALOI
s-CPU/r-CPU/p-CPU/u-CPU	2.8 ×	1.6 ×	3.1 ×	1.0 ×
s-GPU/r-GPU/p-GPU/u-GPU	0.4 ×	0.4 ×	0.2 ×	–

FIGURE 29 Comparison between total execution times for the "update distances/similarities" kernel of the Cartesian product of ranking references algorithm



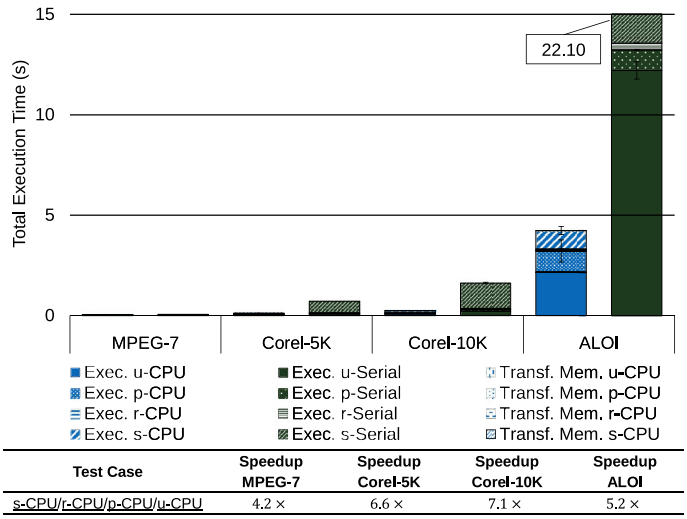
Test Case	Speedup MPEG-7	Speedup Corel-5K	Speedup Corel-10K	Speedup ALOI
s-CPU/r-CPU/p-CPU/u-CPU	0.9 ×	2.4 ×	2.8 ×	5.7 ×
s-GPU/r-GPU/p-GPU/u-GPU	0.8 ×	0.7 ×	0.5 ×	–

When we analyze the total execution time for the Corel-5K dataset in comparison with MPEG-7, we see that there is an increase of 3.1×, with another increase in total execution time by 1.4× from Corel-5K to Corel-10K and 12.5× from Corel-10K to ALOI.

Similar to Figure 27, Figure 28 presents low execution times and high memory transfer times for the "prod" kernel tests. However, unlike other kernels, the CPU parallelization for "prod" did not yield any speedups for the ALOI dataset, and we see that the total execution time for this test was 42.6× that of Corel-10K, which was in turn 1.4× slower than Corel-5K, a test with a time that was 10.3× larger than the one for MPEG-7.

Figure 29 illustrates the results for the "update" kernel. Even though this is the most costly kernel of this method, we see that our CPU parallelization approach allowed us to executed it in under 2.17 seconds in the worst case for the 72,000 images of the ALOI dataset. Moreover, we see a factor of 28.0× between the total execution time on the ALOI dataset when compared with Corel-10K and factors of 2.4× and 6.1× between Corel-10K and Corel-5K and Corel-5K and MPEG-7, respectively.

Finally, we can analyze the best kernel combination in Figure 30. As previously highlighted, executing our parallelization approach on the CPU gave us the best results in all the tested kernels. Moreover, we note that this gave us good speedups for the complete method in all tested datasets. We see that the ALOI total execution takes 18.47× as much time as Corel-10K, which is 2.13× slower than Corel-10K, which in turn takes 7.88× longer than MPEG-7. Nevertheless, in the worst case, the accelerated implementation of CPRR runs in under 5.02 seconds for the largest tested dataset.



Test Case	Speedup MPEG-7	Speedup Corel-5K	Speedup Corel-10K	Speedup ALOI
s-CPU/r-CPU/p-CPU/u-CPU	4.2 ×	6.6 ×	7.1 ×	5.2 ×

FIGURE 30 Comparison between total execution times for the best kernel combinations of the Cartesian product ranking references algorithm

8 | CONCLUSION

In this article, we described a class of unsupervised iterative re-ranking algorithms used for CBIR applications and proposed a unified model to simplify their acceleration on parallel computing systems.

We also designed and implemented parallel versions of four algorithms that belong to this class using the OpenCL framework. This allowed us to validate the results through experiments on different CPUs and GPUs by making only a few modifications to the code. The total execution times we obtained demonstrate that significant speedups can be reached with this technique.

By using this unified model, we were able to create a straightforward approach to splitting each algorithm into independently parallelizable kernels. To improve the performance of the tests made on GPUs, various important questions were addressed during the design of the parallel algorithms as well, such as the need for global synchronization in OpenCL, the concurrent access of data structures, and the impact of memory access patterns. Experimental results have demonstrated that the proposed parallel programming model can be successfully applied to accelerate the processing of multimedia retrieval services.

Future work involves exploring tuning approaches, simultaneous execution of kernels on CPUs and GPUs, and the use of our parallel implementations in web-scale image datasets.

ACKNOWLEDGMENTS

The authors thank AMD, FAEPEX, CAPES (grant #88881.145912/2017-01), FAPESP (grants #2018/15597-6, 2017/25908-6, #2014/12236-1, #2015/24494-8, #2016/50250-1, #2017/20945-0, and #2019/19312-9), the FAPESP-Microsoft Virtual Institute (grants #2013/50155-0, #2013/50169-1, and #2014/50715-9), and CNPq (grants #307560/2016-3, #484254/2012-0, #308194/2017-9, and #140653/2017-1) for the financial support.

ORCID

Flávia Pisani https://orcid.org/0000-0003-2670-7197
Lucas Pascotti Valem https://orcid.org/0000-0002-3833-9072
Daniel Carlos Guimarães Pedronette https://orcid.org/0000-0002-2867-4838
Ricardo da S. Torres https://orcid.org/0000-0001-9772-263X
Edson Borin https://orcid.org/0000-0003-1783-4231
Mauricio BreternitzJr. https://orcid.org/0000-0003-1752-6255

REFERENCES

1. Zheng L, Yang Y, Tian Q. SIFT meets CNN: a decade survey of instance retrieval. *IEEE Trans Pattern Anal Mach Intell*. 2018;40(5):1224-1244.
2. Zhou W, Li H, Tian Q. Recent advance in content-based image retrieval: a literature survey. *CoRR*. 2017; abs/1706.06064.
3. Paulin M, Mairal J, Douze M, Harchaoui Z, Perronnin F, Schmid C. Convolutional patch representations for image retrieval: an unsupervised approach. *Int J Comput Vis*. 2017;121(1):149-168.
4. Gao Y, Shi M, Tao D, Xu C. Database saliency for fast image retrieval. *IEEE Trans Multimedia*. 2015;17(3):359-369.
5. Zheng L, Wang S, Liu Z, Tian Q. Fast image retrieval: query pruning and early termination. *IEEE Trans Multimedia*. 2015;17(5):648-659.
6. Valem LP, Pedronette DCG, de Torres RS, Borin E, Almeida J. Effective, efficient, and scalable unsupervised distance learning in image retrieval tasks. Paper presented at: Proceedings of the 5th ACM on International Conference on Multimedia Retrieval; 2015:51-58.

7. Yang X, Bai X, Latecki LJ, Tu Z. Improving shape retrieval by learning graph transduction. Paper presented at: Proceedings of the European Conference on Computer Vision ; 2008:788-801; Springer, Berlin, Heidelberg.
8. Yang X, Latecki LJ. Affinity learning on a tensor product graph with applications to shape and image retrieval. Proceedings of the CVPR; 2011:2369-2376; IEEE.
9. Pedronette DCG, Torres RDS. Image re-ranking and rank aggregation based on similarity of ranked lists. *Pattern Recog*. 2013;46:2350-2360.
10. Pedronette DCG, Torres RDS. Exploiting pairwise recommendation and clustering strategies for image re-ranking. *Inf Sci*. 2012;207:19-34.
11. Kontschieder P, Donoser M, Bischof H. Beyond pairwise shape similarity analysis. Paper presented at: Proceedings of the Asian Conference on Computer Vision; 2009:655-666; Springer, Berlin, Heidelberg.
12. Yang X, Prasad L, Latecki LJ. Affinity learning with diffusion on tensor product graph. *IEEE Trans Pattern Anal Mach Intell*. 2013;35(1):28-38.
13. Pedronette DCG, Almeida J, de Torres RS. A scalable re-ranking method for content-based image retrieval. *Inf Sci*. 2014;265(1):91-104.
14. Pedronette DCG, de Torres RS, Calumby RT. Using contextual spaces for image re-ranking and rank aggregation. *Multimed Tools Appl*. 2014;69:689-716.
15. Luo L, Shen C, Zhang C, Hengel A. Shape similarity analysis by self-tuning locally constrained mixed-diffusion. *IEEE Trans Multimedia*. 2013;15(5):1174-1183.
16. Pedronette DCG, de Torres SR. Unsupervised rank diffusion for content-based image retrieval. *Neurocomputing*. 2017;260:478-489.
17. Pedronette DCG, Gonçalves FMF, Guilherme IR. Unsupervised manifold learning through reciprocal KNN graph and connected components for image retrieval tasks. *Pattern Recogn*. 2018;75:161-174.
18. Bai S, Bai X, Tian Q, Latecki LJ. Regularized diffusion process for visual retrieval. Paper presented at: Proceedings of the 31st AAAI Conference on Artificial Intelligence; 2017:3967-3973.
19. Donoser M, Bischof H. Diffusion processes for retrieval revisited. Paper presented at: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition; 2013:1320-1327.
20. Yang X, Koknar-Tezel S, Latecki LJ. Locally constrained diffusion process on locally densified distance spaces with applications to shape retrieval. Paper presented at: Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition; 2009:357-364; IEEE.
21. Wang J, Li Y, Bai X, Zhang Y, Wang C, Tang N. Learning context-sensitive similarity by shortest path propagation. *Pattern Recogn*. 2011;44(10–11):2367-2374.
22. Pedronette DCG, Torres SR. Exploiting contextual information for image re-ranking and rank aggregation. *Int J Multimed Inf Retr*. 2012;1:115-128.
23. Xiang B, Song B, Xinggang W. Beyond diffusion process: Neighbor set similarity for fast re-ranking. *Inf Sci*. 2015;325:342-354.
24. Bai S, Bai X. Sparse contextual activation for efficient visual re-ranking. *IEEE Trans Image Process*. 2016;25(3):1056-1069.
25. TOP500.Org. *Top500 List - June 2016 | TOP500 supercomputer sites*; 2016.
26. Stone John E, David G, Guochun S. OpenCL: a parallel programming standard for heterogeneous computing systems. *IEEE Comput Sci Eng*. 2010;12:66-73.
27. Pedronette DCG, Torres RS, Borin E, Breternitz, M. Efficient image re-ranking computation on GPUs. Paper presented at: Proceedings of the 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications; 2012:95-102; IEEE.
28. Pedronette DCG, Torres RS, Borin E, Breternitz M. Image re-ranking acceleration on GPUs. Paper presented at: Proceedings of the 2013 25th International Symposium on Computer Architecture and High Performance Computing; 2013:176-183; IEEE.
29. Pisani F, Pedronette DCG, Torres SR, Borin E. Contextual spaces re-ranking: accelerating the re-sort ranked lists step on heterogeneous systems. *Concurr Comput Pract Exp*. 2016;29(22):e3962.
30. Valem LP, Pedronette DCG, Almeida J. Unsupervised similarity learning through Cartesian product of ranking references. *Pattern Recog Lett*. 2018;114:41-52.
31. Pedronette DCG, Penatti OA, Calumby, RT, de Torres RS. Unsupervised distance learning by reciprocal KNN distance for image retrieval. Paper presented at: Proceedings of International Conference on Multimedia Retrieval; 2014:345-352.
32. Hoi SCH, Liu W, Chang S-F. Semi-supervised distance metric learning for collaborative image retrieval and clustering. *ACM Trans Multimed Comput Commun Appl*. 2010;6:18:1-18:26.
33. Wang C, Zhao J, He X, Chen C, Bu J. Image retrieval using nonlinear manifold embedding. *Neurocomputing*. 2009;72(16–18):3922-3929.
34. Pedronette DCG, de Torres SR. A correlation graph approach for unsupervised manifold learning in image retrieval tasks. *Neurocomputing*. 2016;208:66-79.
35. Pedronette DCG, Torres SR. Exploiting clustering approaches for image re-ranking. *J Vis Lang Comput*. 2011;22:453-466.
36. Mai HT, Kim MH. Utilizing similarity relationships among existing data for high accuracy processing of content-based image retrieval. *Multimed Tools Appl*. 2014;72(1):331-360.
37. Pang S, Xue J, Gao Z, Tian Q. Image re-ranking with an alternating optimization. *Neurocomputing*. 2015;165:423-432.
38. Zhang S, Yang M, Cour T, Yu K, Metaxas DN. Query specific rank fusion for image retrieval. *IEEE Trans Pattern Anal Mach Intell*. 2015;37(4):803-815.
39. Yang F, Jiang Z, Davis LS. Submodular re-ranking with multiple feature modalities for image retrieval. Paper presented at: Proceedings of the Asian Conference on Computer Vision; 2015:19-34; Springer, Cham.
40. Jegou H, Schmid C, Harzallah H, Verbeek J. Accurate image search using the contextual dissimilarity measure. *IEEE Trans Pattern Anal Mach Intell*. 2010;32:2-11.
41. Rostrup S, Srivastava S, Singhal K. Fast and memory-efficient minimum spanning tree on the GPU. Paper presented at: Proceedings of the 2nd International Workshop on GPUs and Scientific Applications, GPUScA; 2011:3-13.
42. Kalva H, Colic A, Garcia A, Furht B. Parallel programming for multimedia applications. *Multimed Tools Appl*. 2011;51(2):801-818.
43. Gunarathne T, Salpitikorala B, Chauhan A, Fox G. Optimizing OpenCL kernels for iterative statistical algorithms on GPUs. Paper presented at: Proceedings of the 2nd International Workshop on GPUs and Scientific Applications (GPUScA'11); 2011:33-44; University of Vienna.
44. Strong G, Gong M. Self-sorting map: an efficient algorithm for presenting multimedia data in structured layouts. *IEEE Trans Multimedia*. 2014;16(4):1045-1058.
45. Wu T, Wang B, Shan Y, Yan F, Wang Y, Xu N. Efficient PageRank and SpMV computation on AMD GPUs. Paper presented at: Proceedings of the 2010 39th International Conference on Parallel Processing; 2010:81-89.
46. Xiao FZ, McCreath E, Webers C. Fast on-line statistical learning on a GPGPU. Paper presented at: Proceedings of the AusPDC; 2011:35-42.
47. Arefin AS, Riveros C, Berretta R, Moscato P. Computing large-scale distance matrices on GPU. Paper presented at: Proceedings of the 2012 7th International Conference on Computer Science & Education (ICCSE); 2012:576-580; IEEE.

48. Alberto C, Amelia Z, Sebastián V. Parallel evaluation of pittsburgh rule-based classifiers on GPUs. *Neurocomputing*. 2014;126:45-57.
49. Strong G, Gong M. Browsing a large collection of community photos based on similarity on GPU. Paper presented at: Proceedings of the International Symposium on Visual Computing; 2008:390-399; Springer, Berlin, Heidelberg.
50. Pham NK, Morin A, Gros P. Accelerating image retrieval using factorial correspondence analysis on GPU. Paper presented at: Proceedings of the International Conference on Computer Analysis of Images and Patterns; 2009:565-572; Springer, Berlin, Heidelberg.
51. Zhu F, Chen P, Yang D, Zhang W, Chen H, Zang B. A GPU-based high-throughput image retrieval algorithm. Paper presented at: Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units; 2012:30-37.
52. Teodoro G, Valle E, Mariano N, Torres SR Jr, Wagner M, Saltz JH. Approximate similarity search for online multimedia services on distributed CPU-GPU platforms. *VLDB J*. 2014;23(3):427-448.
53. He Y, Kang C, Wang J, Xiang S, Pan C. Image tag-ranking via pairwise supervision based semi-supervised model. *Neurocomputing*. 2015;167:614-624.
54. Latecki LJ, Lakämper R, Eckhardt U. Shape descriptors for non-rigid shapes with a single closed contour. Paper presented at: Proceedings IEEE Conference on Computer Vision and Pattern Recognition. CVPR 2000 (Cat. No. PR00662); 2000:424-429; IEEE.
55. Pedronette DCG, de Torres RS. Shape retrieval using contour features and distance optimization. Paper presented at: Proceedings of the VISAPP; 2010:197-202.
56. Khronos OpenCL Working Group. *OpenCL specification version 1.2*; 2016.
57. AMD. AMD accelerated parallel processing OpenCL programming guide; 2013.
58. Bentley JL. *Programming Pearls*. New York, NY: ACM Press Series, Addison-Wesley; 2000.
59. Krizhevsky A, Sutskever I, Hinton GE. ImageNet classification with deep convolutional neural networks. *NIPS'12*. Red Hook, NY: Curran Associates Inc; 2012:1097-1105.
60. Liu G-H, Yang J-Y. Content-based image retrieval using color difference histogram. *Pattern Recogn*. 2013;46(1):188-198.
61. He K, Zhang X, Ren S, Sun J. Deep residual learning for image recognition. Paper presented at: Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR); 2016:770-778.
62. Geusebroek J-M, Burghouts GJ, Smeulders AWM. The Amsterdam library of object images. *Int J Comput Vis*. 2005;61:103-112.
63. Lu H, Ooi B, Tan K. Efficient image retrieval by color contents. Paper presented at: Proceedings of the International Conference on Applications of Databases (ADB); 1994, vol. 819:95-108; Springer, Berlin, Germany.

How to cite this article: Pisani F, Pascotti Valem L, Guimarães Pedronette DC, da S. Torres R, Borin E, Breternitz M. A unified model for accelerating unsupervised iterative re-ranking algorithms. *Concurrency Computat: Pract Exper*. 2020;e5702.
https://doi.org/10.1002/cpe.5702