J Real-Time Image Proc (2015) 10:533–550
DOI 10.1007/s11554-012-0281-6

ORIGINAL RESEARCH PAPER

Speeding up the log-polar transform with inexpensive parallel hardware: graphics units and multi-core architectures

Marco Antonelli · Francisco D. Igual · Francisco Ramos · V. Javier Traver

Received: 13 February 2012 / Accepted: 22 September 2012 / Published online: 21 October 2012

Abstract Log-polar imaging is a kind of foveal, biologically inspired visual representation with advantageous properties in practical applications in computer vision, robotics, and other fields. While the cheapest, most flexible, and most common approach to get log-polar images is to use software-based mappers, this solution entails a cost which prevents certain experiments or applications from being feasible. This may be the case in some real-time (robotic) applications and, in general, when the conversion cost is not affordable for the task at hand. To overcome this drawback and make log-polar imaging more generally available, parallel solutions with affordable modern multi-core architectures have been devised, implemented, and tested in this work. Experimental results reveal that speed-up factors as high as or higher than 10 or 20, depending on the configuration, are possible to get log-polar images from large gray-level or color cartesian images using commodity graphics processors. Remarkable speedups are also reported for current multi-core processors. This noteworthy performance allows visual tasks that would otherwise be unthinkable with sequential implementations to become feasible. Additionally, since three different approaches have been explored and compared in terms of several criteria, different cost-effective choices are advisable depending on different visual task requirements or hardware availability.

Keywords Log-polar mapping · Real-time computer vision · Graphics processors · Multi-core CPUs · CUDA · Shaders

Authors listed in alphabetical order due to similar contributions.

M. Antonelli
Computer Science and Engineering Department, Jaume I University, Castellón, Spain

M. Antonelli
Robotics Intelligence Laboratory, Jaume I University, Castellón, Spain

F. D. Igual (✉)
Dept. Arquitectura de Computadores y Automática, Universidad Complutense de Madrid, Madrid, Spain
e-mail: figual@fdi.ucm.es

F. Ramos · V. J. Traver
Computer Languages and Systems Department, Jaume I University, Castellón, Spain

F. Ramos
Computer Graphics Group, Jaume I University, Castellón, Spain

F. Ramos · V. J. Traver
Institute of New Imaging Technologies (iNIT), Jaume I University, Castellón, Spain

V. J. Traver
Computer Vision Group, Jaume I University, Castellón, Spain

1 Introduction

Unlike conventional uniformly sampled cartesian images, foveal imaging techniques offer advantages in some applications due to their particular geometric and sampling properties. One popular and useful technique is the log-polar model where, by mimicking the vision of mammals including humans, a large field of view is represented in a very compact way by a smart selective data reduction strategy where visual acuity is preserved at the center of fixation (the fovea), while resolution decreases exponentially towards the periphery.

This elegant trade-off solution to the conflicting requirements of having a large field of view, using limited computational resources and having enough visual detail,

has attracted scientists and engineers alike, and the log-polar imaging has been proven useful in several fields such as robotics, computer graphics, computer vision, image processing, and pattern recognition.

A recent survey [44] provides an updated coverage of the properties of the log-polar imaging model and its applications to robotics in tasks such as visual attention, target tracking, egomotion estimation, vergence control, and depth-cues estimation. In practice, most of the experimental work conducted on the log-polar model in robotics use software-based mapping algorithms to get log-polar images from cartesian images. After the mapping, processing is usually limited to the log-polar domain, which saves significant computational resources while benefiting from the properties of this transformed domain. In computer graphics, the log-polar mapping can be a very effective solution for the tricky problem of providing a good view of *both* fine and coarse details *at the same time*, with potential benefits for interactive exploration of complex visual data [13]. For image registration, the log-polar transform brings the possibility of aligning images having undergone large deformations [56]. Pattern recognition benefits mainly from the edge invariance of the log-polar transform [29, 50] as well as from its implicit mechanism for including the visual context (peripheral information) without emphasizing it too much [8], which is an implicit focus-of-attention mechanism useful in other tasks such as vergence control and visual tracking [9, 15, 45].

Obviously, a software-based solution is very convenient because of the flexibility it offers. For instance, different geometric arrangements can be tested cheaply, easily and quickly, and their effects on the particular visual or robotic task at hand can readily be explored. While this setting is perfectly acceptable as a proof-of-concept for a wide range of experiments, it may suffer from serious computational problems in other interesting conditions. For instance, to really enjoy from a wide field of view and the high resolution at the fovea, the mapping should be performed from large cartesian images. Unfortunately, performing the mapping from large/huge images by conventional software routines may be costly enough to prevent fast real-time operations from being feasible which, in turn, may jeopardize the feasibility of some experimental work.

Therefore, to benefit from the flexibility of software and, at the same time, avoid costly transformation routines, fast implementations are called for. In order to reduce the computational burden of the log-polar mapping, several approaches have been explored in the past, including CCD or CMOS sensors [31, 38, 41], special-purpose cards [21, 43], Digital Signal Processing (DSP) hardware [39], Field Programmable Gate Arrays (FPGAs) [52, 53], or algorithms using particular strategies, either exploiting the symmetry of the transform [47] or using scan-line procedures [12]. Nevertheless, despite these efforts, parallel solutions have rarely been investigated. As far as we know, a reported study, 15 years ago, consisted of three processors of a network of transputers which were devoted to the log-polar transform, as a part of an active vision system [14]. Recently, graphic processors are used, by using mipmapping, to implement the log-polar transform in the context of creating image mosaics [25].

While some of these solutions may alleviate or solve the computational problem, some approaches are hard to design or manufacture, or do not allow easy and flexible customization of the mapping parameters, or are economically not affordable. In contrast, the aim of this work was to explore how the problem can be tackled using modern hardware architectures, namely, Graphics Processing Units (GPUs) and multi-core processors, which have become very popular during the past few years, and are widely available in low-cost computers.

While the advantages of GPUs were initially explored in computer graphics, GPUs are now being used in many other application domains, including computer vision. and it is now a trending topic since the emergence of the *General Purpose computations on GPUs* (GPGPU) concept in early 2000s. Recent application fields include biomedical image processing [34], hyper-spectral image manipulation [40] or face recognition [33], to name but a few. Tools and libraries have been developed to make the implementation of computer vision algorithms on GPUs easier [7]. The relationship between speed-up and programming effort [28] is indeed an important factor to bear in mind.

Recent special issues of the IEEE Signal Processing Magazine [17, 18], as well as workshops in the top computer vision conferences [6, 19] are good signs of the vitality, relevance, and timeliness of the topic. On the other hand, the log-polar domain is still a topic of current research and application interest, as revealed by very recent (last two years) publications in a range of domains such as robotics [55], watermarking [32], biometrics [36], machine condition monitoring [54], etc. Therefore, given that both GPUs and log-polar imaging are of current interest, it is the problem addressed in this paper.

Besides GPUs, the interest in the acceleration of computer vision applications using multi-core architectures dramatically increased with the popularization of multi-core processors in the past decade [16, 20, 27]. Recent works aim at combining both technologies, reporting comparative results and the benefits of each architecture for specific problems [24].

To the best of our knowledge, no previous attempt (besides the limited study in [25]) exists to accelerate the log-polar mapping with these architectures and study the

design, implementation, and performance issues involved. However, the interest of such a study to find out how much speed-up can be expected from which architectures is evident. Many applications can potentially benefit from the foreseeable increase in performance, mostly in the fields of computer vision, robotics, and computer graphics. The benefits from faster mappings are one or several of these: the possibility to process image sequences at higher frame rate, the ability to deal with larger cartesian and log-polar images, making affordable more complex log-polar mapping models, etc.

GPUs and multi-core processors are parallel hardware which are an inexpensive option in comparison with specific high-performance shared-memory multi-processors or distributed-memory clusters. Besides this economic advantage, studies in the recent trend in energy-aware computing show that, in comparison with CPUs, power savings are possible with GPUs, provided that the performance increase is beyond some bound [35]. Since energy consumption is of a particular concern for battery-supplied devices, alleviating this problem in this case is especially important. For instance, the use of GPUs for a face recognition application on mobile devices is interestingly shown to provide a 4.25× speed-up in performance and almost a fourfold energy saving with respect to the CPU [48].

In this work, we provide a set of implementations for the log-polar transform targeting different parallel architectures (namely modern graphics processors and multi-core architectures). For the GPU versions of the transform, we propose a comparison between a CUDA-based implementation and a Shaders-based one. While CUDA has emerged as a new standard for the development of general-purpose algorithms on the GPU, it is still limited to Nvidia hardware. Thus, a Shaders-based implementation is an option to obtain portable solutions and compare the abilities of graphics processors from different manufacturers. In addition, such a comparison is useful to gain insights regarding the efficiency of each type of programming models for GPGPU algorithms. In this paper, the advantage of portability of Shaders is not considered, and only Nvidia GPUs have been used in this work so that fair comparisons between the different programming models are possible.

In the following, an overview of the log-polar mapping is first provided (Sect. 2). Then, a sequential algorithm for the log-polar transform and the proposed parallel implementations are described (Sect. 3). These implementations are tested and compared in terms of running time, performance speed-up and other criteria (Sect. 4). Based on these results, some final remarks are then provided, including ideas for further work (Sect. 5), and the main conclusions are drawn (Sect. 6).

2 The log-polar mapping

Given a square $M \times M$ cartesian image, the log-polar mapping results in a $R \times S$ log-polar image, with R and S being the number of rings and sectors of the transform, which define the radial and angular resolution. Usually, $R \cdot S \ll M^2$, hence the computational saving of image processing performed on the much smaller log-polar image. For debugging or didactic purposes and/or as a requirement of the application itself, it is often necessary to reconstruct the cartesian image from the log-polar image itself. This results in a cartesian image of the same size as the original one, but with the corresponding spatially resolution effects of the log-polar transform: high resolution at the center of the transform, and radially decreasing resolution. Both forward and backward mappings are illustrated in Fig. 1. This figure also provides an idea of how small the log-polar image can be in comparison with the cartesian image. Actual sizes will be discussed later in the experiments section (Sect. 4).

Although there are several possible models to map cartesian coordinates (x, y) to log-polar ones (ξ, η), the following popular model is employed here:

$$\begin{cases} \xi = \log_a\left(\frac{\rho}{\rho_0}\right) \\ \eta = q \cdot \theta \end{cases} \tag{1}$$

where (ρ, θ) are the corresponding polar coordinates, a is the logarithmic base, ρ_0 is the radius of a central area which is left out of the mapping to avoid the singularity of the logarithm, and q is the angular resolution for a given size of the log-polar image. The blind central area can be appreciated in the reconstructed cartesian image on the right of Fig. 1. Since the coordinates ξ and η are real-valued, the indices for the log-polar images are the integer part of them and denoted as (u, v). Therefore, $u = \lfloor \xi \rfloor$, $1 \le u \le R$ and $v = \lfloor \eta \rfloor$, $1 \le v \le S$. Similarly, we use (i, j) to represent the integer values corresponding to real-valued cartesian coordinates x and y.

Given an input cartesian image of side length M (i.e. a $M \times M$-sized image), the user input parameters of the transform are R, S, and ρ_0. From these parameters, the other ones (a and q) required in (1) are computed as follows:

$$a = \exp\left(\ln\left(\frac{\rho_{\max}}{\rho_0}\right)/R\right), \quad q = \frac{S}{2\pi}$$

where the maximum radius of the transform, $\rho_{\max}$, is usually set as $\rho_{\max} = M/2$ to cover the most of the input image.

Besides the geometric transformation, the actual log-polar transform consists of performing some sampling strategy to obtain an actual log-polar image from an input cartesian image. Several sampling methods are possible,

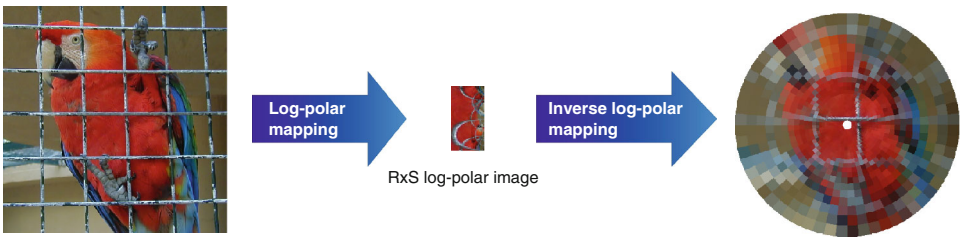

Fig. 1 The log-polar (direct, forward) and inverse (or backward) log-polar mappings (see *text* for details)

but in this work an *averaging* procedure is adopted: the pixel values corresponding to the region of the input cartesian image whose positions map to the same log-polar pixel are averaged to get the value for the corresponding log-polar pixel. Those patches in the cartesian image corresponding to a single log-polar pixel are referred to as *receptive fields* (RFs) after the biological motivation of the transform.

3 Implementation

The implementation of the log-polar transform followed here proceeds, conceptually, in two steps: map building, and actual transformation of an input cartesian image. First, a look-up table (LUT) $\mathcal{L}$ as big as the size of the log-polar image (i.e. $R \times S$) is built so that each of its entries $\mathcal{L}(u,v)$ keeps the list of cartesian pixels $\{\mathcal{L}(u,v)_k = (i_k, j_k)\}_{k=1}^{K}$, with $K = |\mathcal{L}(u,v)|$ being the length of the list of cartesian pixels contributing to (u, v). This LUT is also referred to the log-polar map, or just map, in this paper. For a given geometry of the cartesian and log-polar images (namely, given M, R, S, ρ_0 and $\rho_{\max}$), the map $\mathcal{L}$ can be computed only once and reused for every input cartesian image to be transformed.

In the following, the sequential algorithm (Sect. 3.1), the GPU-based solutions (Sect. 3.2), as well as the port of the log-polar transform to multi-core architectures (Sect. 3.3) are described.

3.1 The sequential algorithm

Once the map $\mathcal{L}$ has been built, the transformation can be computed by a simple sequential procedure (Algorithm 1), which, essentially, averages gray-level values from all cartesian pixels falling into a receptive field. A detail that is missing from this algorithm for the sake of clarity is that of oversampling. Oversampling happens when, at the center of the mapping, at the fovea area, the RFs are smaller than a single cartesian pixel. In this case, the averaging procedure does not make sense as it does when many cartesian pixels fall into the area of a single RF. Therefore, the value of the corresponding log-polar pixel is simply obtained by copying the value of the corresponding cartesian pixel.

Algorithm 1 Log-polar transform (sequential algorithm)

Input: the map $\mathcal{L}$ and the input cartesian image I
Output: the log-polar image L

for $u \leftarrow 1$ to R **do**
  for $v \leftarrow 1$ to S **do**
    $L(u,v) \leftarrow 0$
  end for
end for
for $u \leftarrow 1$ to R **do**
  for $v \leftarrow 1$ to S **do**
    $K \leftarrow |\mathcal{L}(u,v)|$
    for $k \leftarrow 1$ to K **do**
      $(i_k, j_k) \leftarrow \mathcal{L}(u,v)_k$
      $L(u,v) \leftarrow L(u,v) + I(i_k, j_k)$
    end for
    $L(u,v) \leftarrow L(u,v)/K$
  end for
end for

3.2 Graphics processors

Graphics processors have become a realistic alternative for general-purpose high-performance computing implementations. Even though general-purpose algorithms can be developed using the classical graphics-oriented nature of the GPUs, the introduction of the Compute Unified Device Architecture (CUDA) and software infrastructure has dramatically increased the interest on these type of processors for certain non-graphical tasks.

Therefore, GPU programming can be approached in two basic different ways: as a *general-purpose* processor or as a *graphics* processing unit. In the first case, when programmed as a general-purpose processor, the GPU is considered as an array of scalar processors (*Streaming Processors*, SPs) grouped in clusters (or *Streaming Multiprocessors*, SMs). The GPU supports a highly multithreaded execution model, with threads executing a common program (or *kernel*) in a *Single Instruction Multiple Data* (SIMD) fashion. Threads are grouped into *blocks of threads*, and communication among them is performed through different memory spaces: *global* off-chip

memory for the communication of all the executing threads, and small on-chip *shared* memories per SM for the communication of threads within the same block of threads.

Alternatively, when programmed as a purely graphics processing unit, the GPU is viewed as a sequence of specialized functional units in which each of the stages of the graphics pipeline is executed. Parallelism is extracted internally within each stage of the execution pipeline, exploiting the parallel capabilities of the architecture mentioned above.

Graphics Processors have evolved from being configurable to being programmable. Within this programmable rendering pipeline, it is possible to use a set of software instructions that is utilized to perform rendering effects on graphics hardware with a considerable flexibility. This set of instructions is usually known as a *shader*, and the language specification of the possible instructions to be included in a shader is named the *shader language*. Most-known shader languages are GLSL [23] and HLSL [11], which were created and developed for their use with OpenGL and Direct3D API, respectively. Thus, each of the programmable processors contained in the processing pipeline can be programmed using a shader. In particular, current processors for the OpenGL processing pipeline are the vertex, tessellation control, tessellation evaluation, geometry, and fragment processors [23]. Functionalities offered by the different processors are beyond the aim of this work. However, we underline vertex and fragment processor as the programmable units we will use to perform the log-polar transform.

On the one hand, *vertex* processors perform operations on input vertices and their associated data. Operations are executed once per vertex and the resulting vertex is input to the next stage in the graphics pipeline. On the other hand, *fragment* processors are the last programmable units in the pipeline and they directly operate on the color of the generated pixels, which are used to update the frame buffer or texture memory, depending on the render target.

One goal of this work is to compare both interpretations of the architecture (general-purpose and graphics processing) in the context of the log-polar transform. This comparison is possible from different perspectives: performance, accuracy of the results, and ease of development. The following subsections describe the developed GPU implementations using CUDA (Sect. 3.2.1) and Shaders (3.2.2).

3.2.1 CUDA-based implementation

The implementation of the log-polar transform with CUDA does not use the look-up table $\mathcal{L}$ to know the mapping between cartesian and log-polar pixels, but rather, the conversion between cartesian and log-polar coordinates is carried out on-the-fly while an input cartesian image is processed to get the log-polar one. This decision is made to favor time spent on computation over that spent on memory accesses, since recomputation instead of redundant memory access is one of the main strategies adopted in CUDA to hide memory latency and to take benefit from the huge data processing power of modern GPU [30]. Despite this general rule-of-thumb, we tested an alternative implementation using the LUT to check whether this general guideline really holds in the context of our problem (Sect. 4.2).

This implementation requires to allocate three areas into the global memory of the GPU: one area is devoted to the cartesian image (I) that has to be transformed; another memory area is for the resulting log-polar image (L), and the third one is used to compute and keep the *pixel count* (K). For each log-polar pixel, K stores, at each step of the algorithm, the number of cartesian pixels that have been mapped into it. The transformation is based on the following procedure (Algorithm 2): the cartesian image is scanned and, for each cartesian pixel position (i, j), the corresponding log-polar pixel (u, v) on which (i, j) contributes is computed. Then, the value of $L(u, v)$ is updated according to the value of the cartesian pixel, and its pixel counter $K(u, v)$ is increased (line 6). Therefore, the updating rule corresponds to a running average (line 5). The parallelization of this procedure is obtained by delegating the computation of a column of the cartesian image to each thread. The index of the column i that is computed by a thread is determined by the thread and block indices, t and b, as $i = t + b \cdot n$, where n is the number of threads per block. The number of blocks is therefore $B = \lceil \frac{M}{n} \rceil$.

Algorithm 2 CUDA kernel: cartesian to log-polar mapping

Input: the input cartesian image I
Output: the partially computed log-polar image L, and updated counters K

1: $i \leftarrow t + b \cdot n$
2: **for** $j \leftarrow 1$ to M **do**
3: $(u, v) \leftarrow \mathsf{cart2lp}(i, j)$
4: **if** $u < R$ **then**
5: $L(u, v) \leftarrow \frac{L(u,v) \cdot K(u,v) + I(i,j)}{K(u,v)+1}$
6: $K(u, v) \leftarrow K(u, v) + 1$
7: **end if**
8: **end for**

Better performances of the algorithm are obtained when n facilitates coalesced reading of the global memory access. Coalescing, a popular term in CUDA programming, refers to combining individual memory accesses performed by consecutive threads into single wide memory transactions under certain restrictions [30]. Among the several values that allow coalescing, $n = 64$ was chosen empirically.

The direct updating of the values of L and K is performed directly using the global memory. Some improvements to the performance could be achieved using the shared memory. However, this approach would have two drawbacks. First, the size of the shared memory is usually not big enough to contain the log-polar image; thus we would need to know beforehand which portion of the log-polar image is modified by a block. Second, we would need to synchronize the threads before copying.

Once this main procedure is completed, the effect of oversampling is dealt with by assigning the gray-level of the corresponding cartesian pixel to the log-polar pixel for which the corresponding pixel count (K) is zero. This procedure (Algorithm 3) is parallelized by delegating the computation of a column (a ring) of the log-polar image to each thread. In this case, the number of blocks B was computed as $B = \lceil \frac{R}{n} \rceil$.

Algorithm 3 CUDA kernel: Oversampling

Input: the input cartesian image I, and computed counters K
Output: the partially computed log-polar image L

1: $u \leftarrow t + b \cdot n$
2: **for** $v \leftarrow 1$ to S **do**
3: $(i, j) \leftarrow$ lp2cart(u, v)
4: **if** $K(u, v) = 0$ **then**
5: $L(u, v) \leftarrow I(i, j)$
6: **end if**
7: **end for**

3.2.2 Shaders-based implementation

In general, the adaptation of a sequential and iterative algorithm to the graphics pipeline is not a straightforward task. The Shaders-based implementation of the log-polar mapping follows the workflow outlined in Fig. 2, which is made up of two fundamental tasks, preprocessing and rendering, which are described subsequently.

At the *preprocessing* step, the graphics pipeline is prepared to perform the log-polar transform. On the one hand, it is necessary to allocate space into the graphics memory for the source image to be transformed I, and for the resulting log-polar image L. In Fig. 2, they are represented as the *Cartesian Image* and *Log-polar Image*, respectively. On the other hand, a *Geometry map* must be constructed and uploaded to the graphics memory as well. Since loops are not suitable for the graphics pipeline, this map is a data structure derived from the look-up table $\mathcal{L}$, and adapted to start the log-polar transformation in the vertex processors. These processors are the first programmable units in the pipeline, and they are essential in this algorithm. Computations into the graphics pipeline are directly related to the number of vertices to be processed. In this case, a vertex is created for each contribution of the cartesian image into the log-polar image. Therefore, the *Geometry map* contains a list of vertices, where each vertex is a 2D pair (u, v) with its associated data (i, j). In this way, every contribution of the cartesian image is computed as a vertex into the graphics pipeline. Therefore, if an entry in the look-table $\mathcal{L}(u, v)$ has K cartesian pixels contributing to (u, v), there will exist K vertices into the *Geometry map*.

Before rendering, it is also fundamental to establish a viewport of $R \times S$ pixels so that every vertex in the *Geometry map* gets mapped to exactly one pixel into the final image L. In computer graphics, the usual output of the graphics pipeline is the screen. However, we are interested in obtaining a log-polar image. To that end, the rendering target is changed to be a texture instead of the screen, and the log-polar image is indeed a texture. Thus, just a single pass is required to create the final image, which can be reused as a texture source with no more rendering passes being required.

At *rendering* time, transforming operations are performed in the vertex and pixel shaders units. First, rendering starts by issuing an OpenGL command that sends all the vertices from the *Geometry map* to the vertex shader units. Vertex shaders process every vertex by simply passing down the information to the pixel shader units where transforming operations take place. Thus, for each

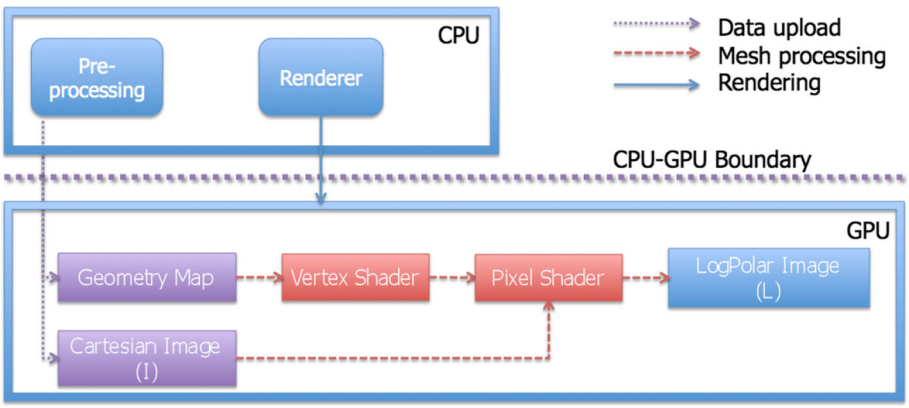

Fig. 2 Workflow in the Shaders-based log-polar transform

existing contribution of the cartesian image into the log-polar image a pixel shader is executed. The pixel shader receives the information sent by the vertex shader, that is, the 2D pair (u, v) and its (i, j) data. This second pair is used by the pixel shader to read from the Cartesian image, I, the contributing color, which is linearly interpolated to the existing color in the pixel at (u, v). Once the rendering stage is completed, a texture, actually corresponding to the log-polar image L, is obtained.

3.3 Implementation based on multi-core processors

If we consider the sequential log-polar transformation illustrated in Algorithm 1, as the transformation advances, each one of the R *rings* of the output image is obtained. Thus, the calculation of a ring does not start until the previous ring has been completely obtained, in a sequential fashion. However, the computation of a given ring in the log-polar image is completely independent from the computation for the *subsequent* rings. Therefore, the computation of each one can effectively proceed in parallel. As a result, we propose a parallelization of the log-polar transform targeting multi-core processors in which the iterations of the outer loop in Algorithm 1 (from the first to the R-th ring) are divided and assigned to subsequent threads at runtime. Proceeding in this manner, in a certain point of the execution, as many rings as threads will have been created and will be processed in parallel.

Alternatively, the described strategy could be applied to process *sectors* of the log-polar image in parallel instead of the rings. The success or failure of each alternative will depend on the amount of rings and sectors (parameters R and S) in the log-polar image with respect to the total number of threads performing the transformation. The more rings or sectors are available, the more concurrency is exposed, and thus, more effective is the parallelization process. Since $R > S$ is the most common practical situation [46], assigning threads to rings is therefore the most sensible choice in our problem to avoid falling in a waste of resources as a result of having threads idle during the computation. However, our tests showed that performance was similar in both strategies, and only in extreme cases these different strategies make a difference.

OpenMP has been used for the parallelization of the transformation on multi-cores. This infrastructure offers several scheduling policies to assign loop iterations to threads. The log-polar transformation, as implemented in Algorithm 1, presents different workloads depending on the size of the RFs at different eccentricities u. In the algorithm, this amount of work is dictated by the length of the set $\mathcal{L}(u, v)$. For this reason, the usage of a dynamic scheduling policy of iterations to threads has attained the best performance results in our tests for both ring-based and sector-based parallelization. This type of policy can hide the inherent irregularities in the amount of work assigned to each thread, at the expense of some extra overhead in terms of computation time.

All the experimental results shown in Sect. 4 correspond to a parallelization at ring-level, using exclusively a dynamic scheduling policy. This has been the combination of the parallelization strategy and the scheduling policy that has attained better results in our experimental process.

4 Experimental results

For the hardware resources used in the evaluation of our parallel implementations of the log-polar transform we have chosen the multi-core CPU and GPU architectures representative of hardware of a similar generation. A detailed description of this hardware is included in Table 1. Both the multi-core CPU and the GPU selected for our evaluation are high-performance representatives of current desktop systems. In particular, the Intel Xeon architecture is a dual-socket, quad-core processor running at 2.27 GHz. The Nvidia GPU is equipped with 480 cores running at 1.4 GHz (while the rest of the GPU runs at 700 MHz). The interconnection between the GPU and the rest of the system is performed via a PCIExpress 2.0 bus. Besides the main experiments with this set-up, an additional experiment has been developed (Sect. 4.2) to find out how cheaper GPUs behave in this problem.

On the software side, the parallelization on the multi-core processor has been carried out using the OpenMP [5, 42] implementation included in the GNU compiler infrastructure, version 4.1.2. For the CUDA [2] implementation, we used version 3.2 of the toolkit and version 260.19.21 of the graphics driver provided by Nvidia. For the Shaders-based implementation, OpenGL Shading Language (GLSL) was used as the shader programming language [23]

Table 1 Summary of features of the hardware used in the experiments

	CPU	GPU
Processor model	Xeon E5520	Geforce GTX480
Processor codename	Intel Nehalem	Nvidia Fermi
Clock frequency	2.26 GHz	1.4 GHz
Max. performance	145 GFlops	1.35 TFlops
Interconnection bus	PCIExpress 2.0	
Memory frequency	2 × 400 MHz	1.85 GHz
Memory bus width	64 bit	384 bit
Memory peak bandwidth	12.8 Gb/s	177.4 Gb/s
Memory size and type	24 GBytes DDR2	1.5 Gbytes GDDR5

and OpenGL [4] as the supporting graphics library. We have used the CImg library [1] for basic image management (I/O operations and pixel access).

4.1 Computational performance

In order to know the relative merits of each architecture (namely, sequential algorithm, GPU with CUDA, GPU with Shaders, and multi-core with 8 cores) under different practical conditions, a series of log-polar transformations were carried out under each case for a number of different geometric parameters. In a first experiment, the size of the log-polar image was set to a fixed value, and the size $M \times M$ of the square cartesian image was varied as $M \in \{256, 512, 1024, 2048, 4096\}$. Conversely, in a second experiment, the size of the cartesian image was kept and that of the log-polar image was varied as $R \in \{32, 64, 128, 256\}$. The other parameters defining the geometry of the log-polar transform were set as $S = 2 \cdot R$, $\rho_0 = 5$ and $\rho_{\max} = \frac{M}{2}$, which are reasonable practical choices. Additionally, the tests were done both on gray-level and color images.

In all these cases, the transformation was repeated $k = 100$ times, and the average and standard deviation of the times were computed. However, the variance was observed to be very low and it is therefore not reported here. In the following, we refer to "CPU" to mean the implementation of the sequential algorithm. To make the comparison with the CPU as fair as possible, the times for the CPU case actually refer to the implementation on the multi-core with a single core, so that time differences are not due, for instance, to either biased software implementations, or hardware being from different generations, or of significantly different performances.

The resulting execution times (in milliseconds) are shown in Figs. 3 and 4 for the configurations explained above. While the absolute times are good figures to know, the speed-up factors provide interesting insight into the relative gain, in particular when the differences in absolute times are hard to notice in the plots (e.g. as it happens in Fig. 3 for $M \leq 512$). Thus, these figures are complemented with the corresponding speed-up factors shown in Figs. 5 and 6. Notice that the horizontal axis in all these figures is not representing values varying linearly; the image side length M doubles and, therefore, image size quadruples.

It can be readily observed (Figs. 3, 4) that significant time is saved with *any* of the proposed parallelization with respect to the sequential algorithm, and this advantage becomes more and more evident with the increase of the size of the images. When comparing the plots in Fig. 3 with those in Fig. 4, it can be seen that the increase in the size of the cartesian image (M) affects every implementation more than the increase in the size of the log-polar image (R), and higher speed-up factors are generally achieved when this happens. This is an interesting result since this is one of the motivations behind this work: whether parallel software-based implementations of log-polar transform can make it to be affordable for stringent real-world conditions and/or for significant large input cartesian images. As another practical consequence, these speed-ups would also make possible processing images at a much higher frame rates.

When comparing the plots on the left with those on the right in all these figures, it can be appreciated that performing the log-polar transform on color images (3 channels) is more costly than on gray-level images (single channel) in all cases, except for the Shaders case, which does not benefit from single-channel gray-level images since these are actually treated as color images. It is important to notice that for the Shaders case we treated gray-level images as color ones because it does not actually affect the results. In fact, we read only a channel from the gray images when processing them. Certainly, storing only a channel for gray images is more efficient in terms of GPU memory requirements, but it has not an impact on time perfomance.

In general, the best times are obtained with the CUDA implementation especially for large images. As an exception, the Shaders-based implementation is an appealing solution for small and medium-sized color images compared with the CUDA-based implementation (see, Fig. 6, top-right). Many factors can benefit this remarkable behavior. First, the Shaders-based implementation, as designed, is especially efficient in the management of color images. Second, the CUDA model delegates a bunch of implementation decisions to the programmer's hands (namely register usage, data coalescing parameters, number of blocks and threads per block,…). This type of low-level decisions is usually hidden in the Shaders model, and execution parameters are usually decided and optimized at run time by the software infrastructure, depending on the specific details of the input data or the renderization process to be performed. In this sense, although the parameters chosen for the CUDA implementation are appropriate for large images (as demonstrated, e.g., in the plots at the bottom of Fig. 6), the implicit execution configuration in the Shaders model seem optimal for small- or medium-sized images.

The general benefits and strengths of using the GPU as the execution platform, independently from the chosen programming model, are revealed as the size of the data sets to be processed increases. As an example, consider the high performance attained by the multi-core implementation for medium-sized images in Fig. 6 (top-left). While the multi-core implementation is highly competitive for this specific configuration, its benefits are lost as the size of

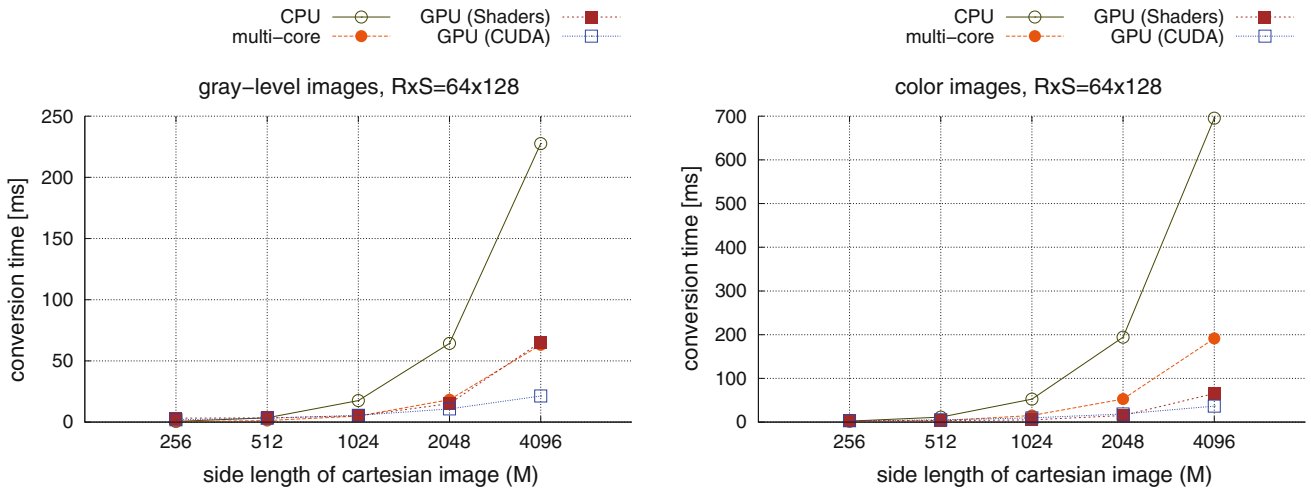

Fig. 3 Conversion times for gray-level (*left*) and color images (*right*), by varying the size of the cartesian image for a fixed size of the log-polar image

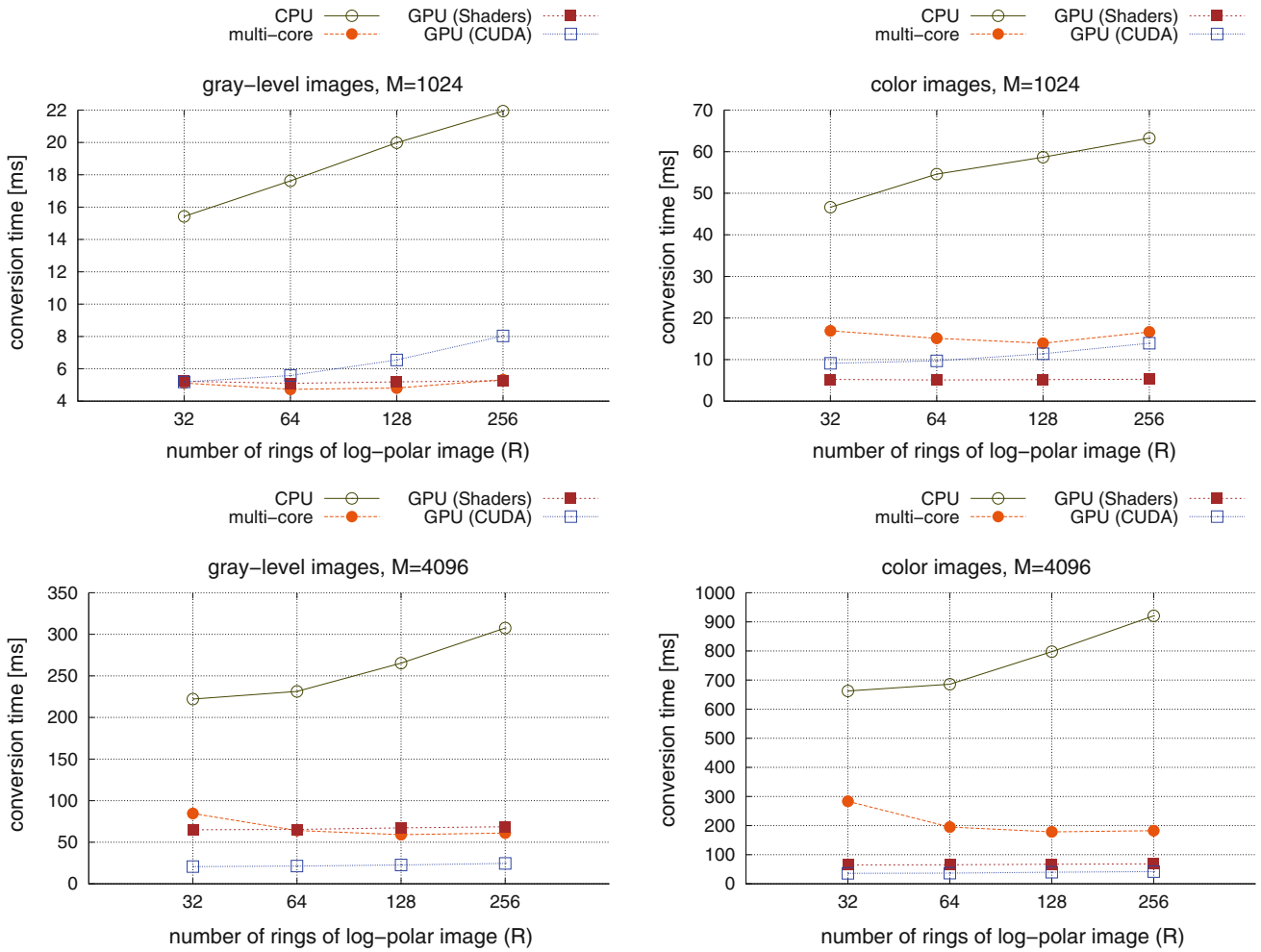

Fig. 4 Conversion times for gray-level (*left*) and color images (*right*), by varying the size of the log-polar image for a fixed size of the cartesian image ($M = 1024$ at *top row*, $M = 4096$ at *bottom row*)

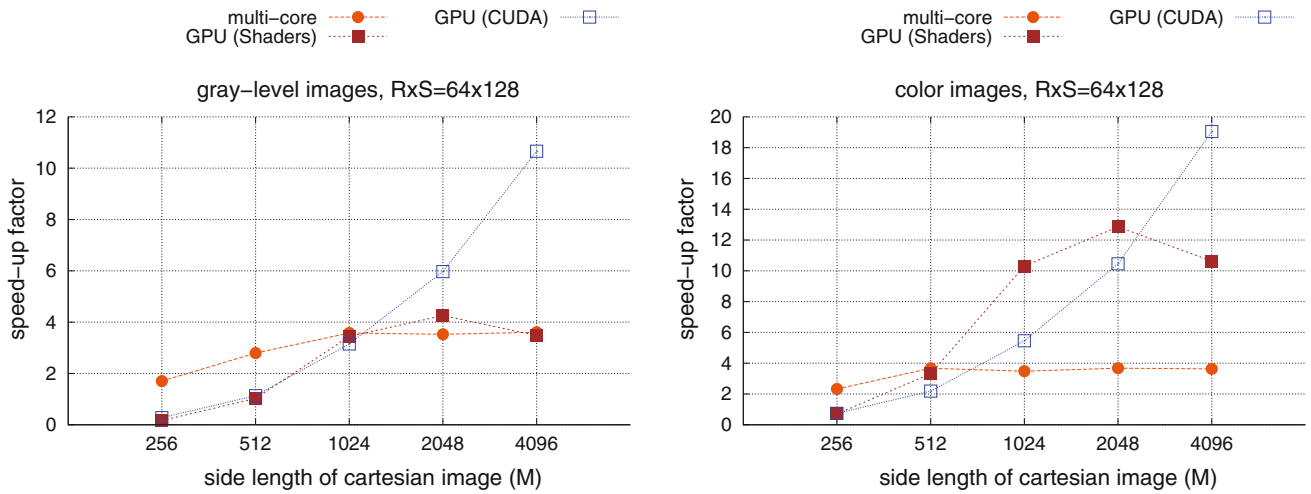

Fig. 5 Speed-up factors for gray-level (*left*) and color images (*right*), by varying the size of the cartesian image for a fixed size of the log-polar image

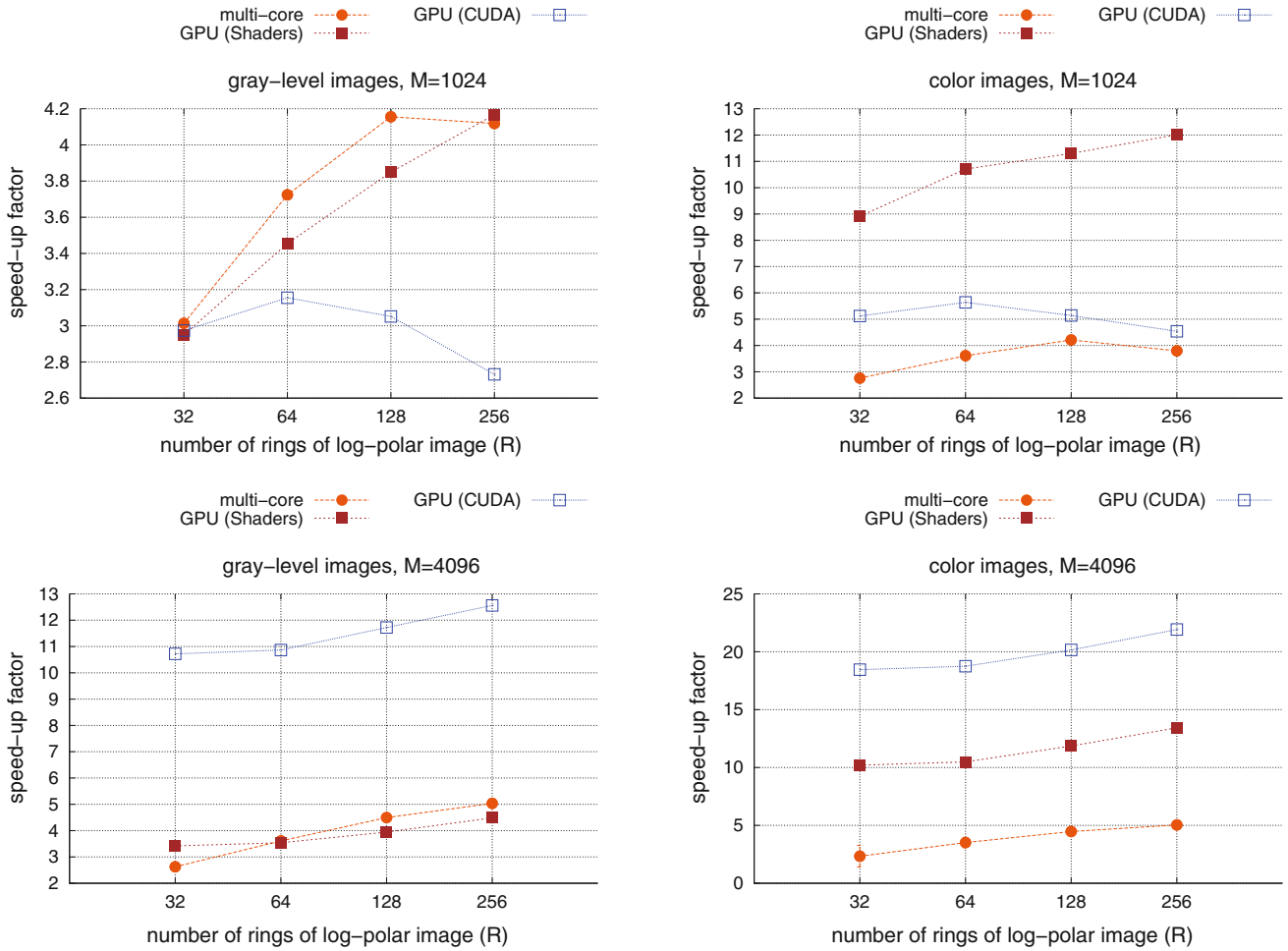

Fig. 6 Speed-up factors for gray-level (*left*) and color images (*right*), derived by varying the size of the log-polar image for a fixed size of the cartesian image ($M = 1024$ at *top row*, $M = 4096$ at *bottom row*)

the cartesian image grows, for both color and gray-level images. Thus, the adaptation of a given architecture to this particular problem does not only depend on the specific algorithm used, but also on the particular characteristics of the input data to be processed, mainly its size.

These results are not easily comparable to related published works, because not much work exists on this topic, not extensive experimental work is done or reported, and the test conditions are quite different. As an example, an FPGA-based implementation of the log-polar transform [53] is reported to achieve an speed-up factor of around 1.5×, which is smaller than most of the speed-up factors we get in most geometric configurations tested, even when the reference time in [53] is taken from the execution of a Matlab-based implementation and, as it is well known, implementations based on Matlab are generally of very low performance and therefore, not a fair reference for the performance of a sequential algorithm.

4.2 Influence of using a precomputed map in CUDA

In the case of the CUDA-based implementation, we wanted to test whether (and how) the performance would change using a precomputed transform map $\mathcal{L}$ with respect to computing the map implicitly "on-the-fly". In other words, the values (u, v) are obtained directly by the LUT $\mathcal{L}(i, j)$ instead of being computed. This implies testing whether the cost of GPU memory access pays off. To that end, an alternative CUDA-based implementation was developed which precomputes the map $\mathcal{L}$ off-line (on the CPU) and uses it during the transformation of actual images. Additionally, besides the 480-core GPU used in the rest of the experiments, a cheaper and more modest 16-core GPU was included in the test. Resulting speed-up factors are compared in Fig. 7.

Even though the general guidelines [30] suggest to limit memory accesses even at the cost of performing more computations, these results indicate otherwise: using the LUT $\mathcal{L}$ is beneficial since higher speed-ups are obtained. This is due to the high computational cost required by the arithmetics operations used in the algorithm (such as the logarithm or the square root). However, the algorithm that was tested was not optimized to minimize the computational cost and slight improvements in the implementation not using the LUT can enhance the performance to obtain results close to those offered by the implementation using the LUT.

On the other hand, it can be seen that the speed-up factor obtained with the 16-core GPU is similar to those offered by the better 480-core GPU up to $M \approx 1024$. For cartesian images bigger than that, the speed-up factor stops growing for the 16-core GPU. This illustrates that interesting computational gains are possible even with modest GPUs; for instance, about a 5× improvement is achieved when obtaining color log-polar images from cartesian images of 1024×1024 and bigger. Intermediate results can be expected for GPUs of in between performances.

Regarding the use of the LUT for the 16-core GPU, it was observed, although not shown in the plots, that it was also advantageous. In this case, memory bandwidth demands are less strict, as there is a lower number of cores accessing it. It is thus a logical result that the use of a LUT is more advantageous in this scenario. Therefore, although the effect of the LUT cannot be generally predicted beforehand and it should be assessed empirically, the logical result derive.

4.3 Scaling with the number of cores in multi-core

For the multi-core implementation, it was interesting to find out how the speed-up relates to the number of cores

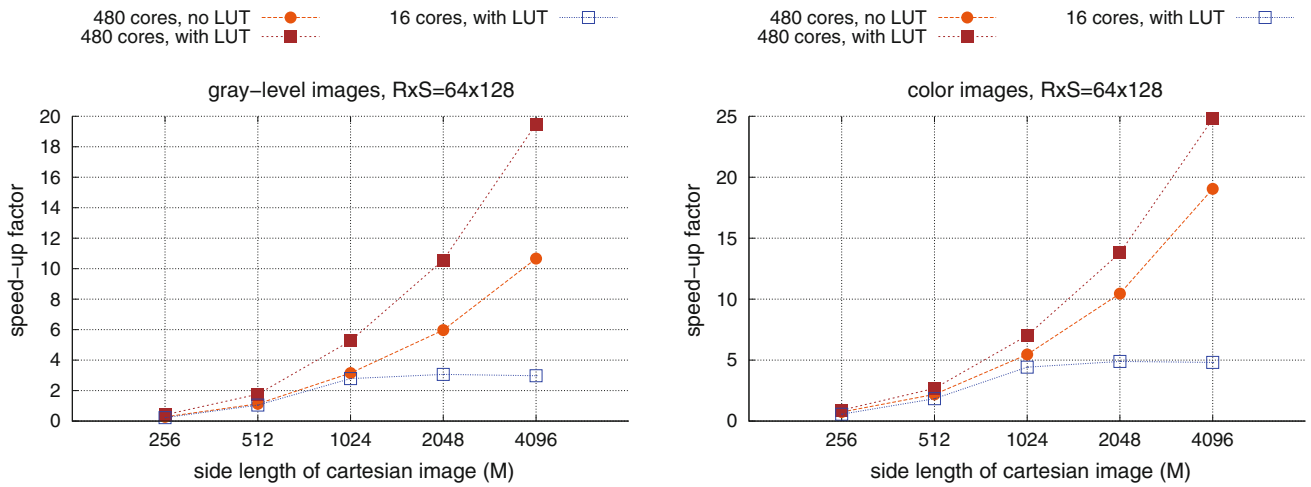

Fig. 7 Speed-up factors for gray-level (*left*) and color images (*right*), obtained by varying the size of the cartesian image for a fixed size of the log-polar image for CUDA implementations with and without the use of a precomputed LUT (the map $\mathcal{L}$)

used. Thus, log-polar transformations were repeated for $c \in \{2, 4, 8\}$ cores. Results for 16 cores were also conducted due to the Hyperthreading capabilities of the Intel processor, with no extra performance benefit. Speed-up results are shown in Fig. 8. The log-polar transform, as implemented, is a memory-bound algorithm. The ratio between floating-point operations (`flops`), and memory accesses (`memops`) is low, so the performance is mainly limited by the pace at which memory can provide data to computing units. This effect becomes worse as the number of processing units demanding data increases. Taking as an example the transform of gray-level images, speed-up factors vary between 1.5× using 2 cores, 2.5× using 4 cores and 3.5× using 8 cores. These factors give an idea of the degree of penalty introduced by memory accesses for this specific platform as the number of cores accessing it simultaneously is increased. Therefore, on systems with a reduced number of cores (2–4), as in current desktop computers, the scalability of the implementation is of wide appeal, but can suffer from bottlenecks on architectures with more cores. Additional experiments revealed better scalability results (near to optimal speedups) for higher values of R and S, i.e. $R = 1024$ and $S = 2R$; however, given that those setups do not usually appear in real problems, we have chosen not to consider that behavior as the standard behavior of the multi-core implementation.

4.4 Time breakdown

The times considered above are due to the actual log-polar mapping. Nevertheless, there are some other "overhead" times deserving attention. Both GPU-based solutions need to transfer the input cartesian image from main memory to GPU memory and the output log-polar image from the GPU to the main memory. This is not the case for the multi-core architecture, since both images reside in main memory and no transfer is required.

Another datum to be transferred is the map defining the coordinates mapping, if it is used. Finally, the computation of the map itself may be considered. However, in most applications, this map need only be computed and transferred *once* (off-line) for a given set of geometric parameters (M, $R \times S$, ρ_0, $\rho_{\max}$). After that, at on-line time, all incoming images can therefore be processed by using the same precomputed map, with no extra cost. Therefore, times involved in map computation or transfer are not considered here.

There is a general reasonable concern that the speed-up gained by the GPU performance may be lost with the required data transfer between main and GPU memories [22]. This is particularly concerning when using toolkits where the programmer has little control to minimize such transfers [37]. To test whether this is the case also for the problem considered here, the times to transfer the cartesian image from the main memory to the GPU memory and to transfer the log-polar image from the GPU memory to main memory are considered for three representative sizes of the cartesian image: $M = 256$ ("small"), $M = 1024$ ("medium-size") and $M = 4096$ ("large").

It can be seen (Fig. 9) that most of the time is spent on the actual transformation, and only a (very) small fraction of the elapsed time is spent on data transfers. On the other hand, the situation is somehow different for different architectures/strategies depending on the size of the images. Therefore, paying attention to the size of the expected images is important to select, if possible, the most suitable architecture.

In the case of gray levels, for small images, the multi-core CPU is the best choice. For medium images, all architectures offer similar performance, but the overhead

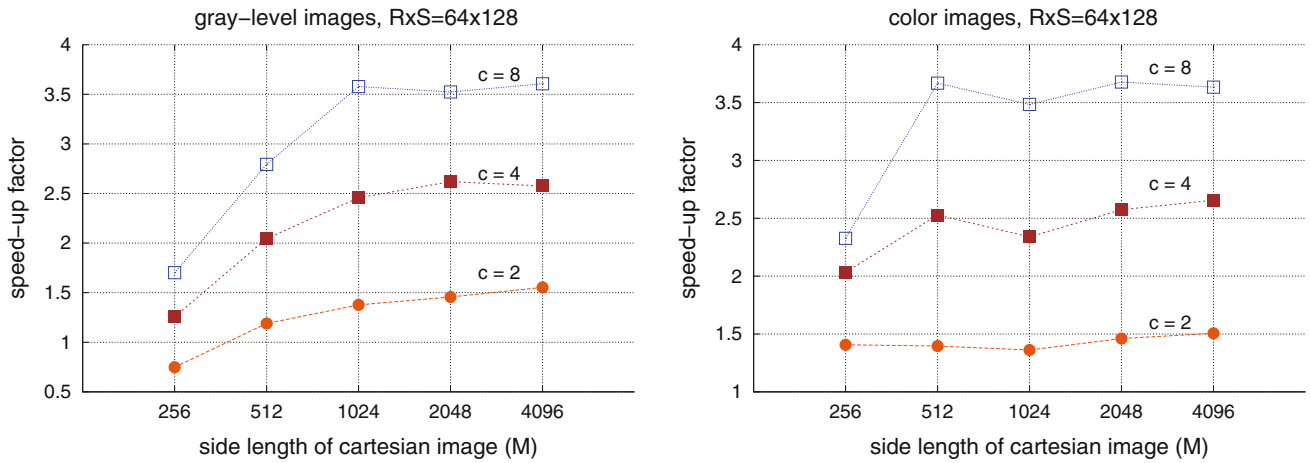

Fig. 8 Speed-up factors for gray-level (*left*) and color images (*right*), obtained by varying the size of the cartesian image for a fixed size of the log-polar image for multi-core implementations with different number of cores c

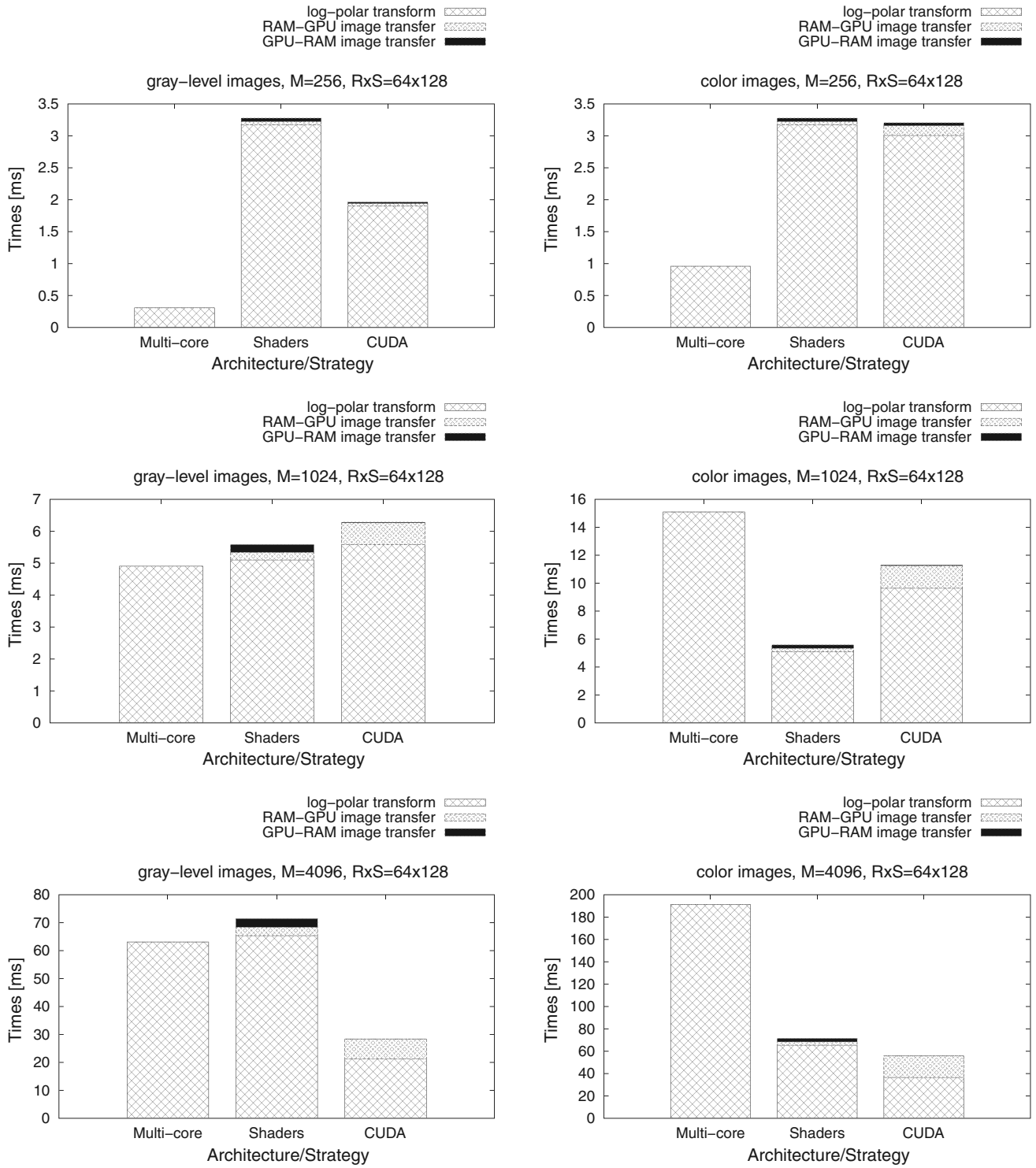

Fig. 9 Time breakdown for three different sizes of the cartesian image (*top*: $M = 256$, *middle*: $M = 1024$ and *bottom*: $M = 4096$), for gray-level (*left*) and color (*right*) images, for the three different architectures tested

cost for image transfers may still suggest not to use the GPU. For large images, this overhead is made up for the significant speed-up achieved in the actual transformation, and the GPU with CUDA is clearly preferred. In general, the latency introduced by the PCIExpress bus makes the GPU implementation only suitable for relatively large images, provided that the cartesian image resides on main memory prior to the transform. For large images, the high bandwidth of the bus is responsible of the negligible amount of time devoted to data transfers.

For the color case, results for small images with Shaders and CUDA are more similar between them than in the cartesian case and still worse than those with multi-core CPU. For medium images, the Shader-based implementation is preferred, and for large images, the CUDA choice is better, although not as clearly as in the gray-level case.

Notice that transfer times in the Shaders and CUDA case are different because, even though the GPU is the same in both cases, experiments were carried out on different computers. This means that besides the GPU itself, the performance of the rest of the setup may also have some impact, mainly because of the specific CPU, chipset and PCIExpress bus configuration used in each measurement.

4.5 Qualitative and quantitative visual assessment

For a complete study, besides the time performance, it is important to know the quality of the images obtained. Figure 10 shows the log-polar images obtained with the sequential algorithm as well as those obtained with the parallel hardware and algorithms considered here. To facilitate the qualitative visual assessment, the reconstructed cartesian images are also shown. Therefore, a visual inspection of these images reveal they are correct. Note that the reconstructed images are computed with a sequential algorithm in all cases (no parallel implementation of the inverse log-polar transform is used).

In order to quantify the quality of the log-polar image L_m computed with a given parallel method m, the log-polar image obtained with a CPU with the sequential algorithm, L, is taken as a reference, and a similarity measure between L_m and L can be used. Here we adopt the Normalized Cross Correlation (NCC), which is defined as

$$\frac{1}{2}\cdot\left(1+\frac{\sum_{i,j}(L(i,j)-\bar{L})\cdot(L_m(i,j)-\bar{L}_m)}{\sqrt{\sum_{i,j}(L(i,j)-\bar{L})^2\cdot\sum_{i,j}(L_m(i,j)-\bar{L}_m)^2}}\right) \quad (2)$$

where L and L_m are the compared gray-level images and $\bar{L}$ and $\bar{L}_m$ are their average values. By definition, the value of NCC lies in [0,1], and the higher its value the more similar the compared images are. Note that here NCC is computed between the actual log-polar images, and *not* between their reconstruction into the cartesian domain and the original cartesian image.

Figure 11 shows the NCC for the images obtained with the three parallelization techniques considered in this paper for the case $R = 64$, $S = 128$ and varying M. The multi-core algorithm yields the images resembling the most the ones computed with the CPU. The Shaders-based solution provides the images significantly less accurate, and the CUDA-based solution has an intermediate accuracy. In all cases, the NCC is quite stable for different sizes of the cartesian image. Notice that the lower value of NCC for the Shaders-based case can in fact be linked to the different

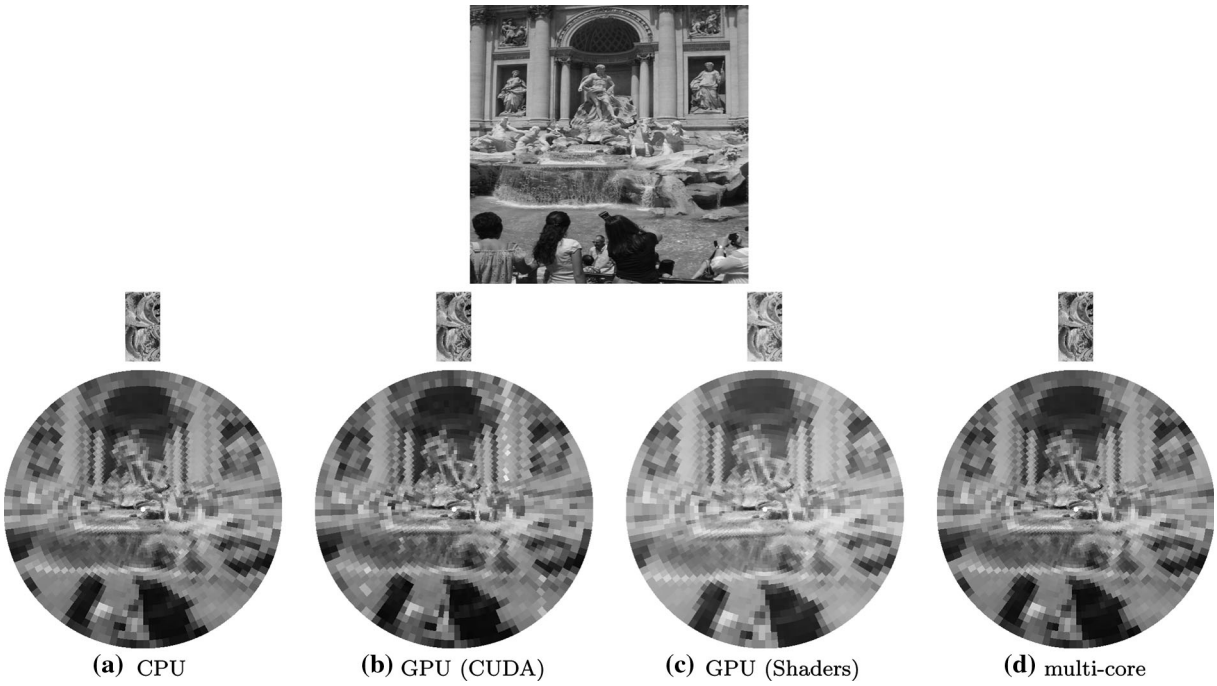

Fig. 10 Example of log-polar images (*middle row*) obtained from an input cartesian image (*top row*) with the different architectures for ($M = 512$, $R = 64$, $S = 128$, $\rho_0 = 5$). All reconstructed images (*bottom row*) are computed with a sequential implementation of the backward transformation

general brightness noticeable in the images in Fig. 10. The reason behind this lower value of NCC in comparison with the CUDA-based and multi-core solutions is a numerical problem: the limit of precision in the frame buffer. In general, the frame buffer has a default precision of 8 bits per channel which is accurate enough for screen visualization. However, in the context of this problem, more precision is necessary since many contributions of the cartesian image onto a given log-polar pixel can occur (for big RFs). The reason why the multi-core solution offers the highest similarity to the images computed with a sequential algorithm has probably to do with the fact that the implementation follows more closely the original sequential code. Note that the single-core CPU used to get the images computed with the sequential algorithm is different to the multi-core CPU used for the tests reported in this paper.

Regarding the lower value of NCC in the Shader case, two remarks are in order. First, although the actual pixel values are not exactly the same as those resulting from the sequential algorithm, the structure of the visual scene is correctly captured and this is what is important in many or most practical applications. Second, if higher brightness accuracy is desired for some application, a two-pass algorithm may be devised to cope with the numerical problem mentioned above. This two-pass algorithm would accumulate all image values corresponding to a receptive field in a first pass and perform the division (to compute the average) in a second pass, instead of accumulating the weighted values in a single pass, as performed in the current implementation. Since this alternative procedure would, however, involve some extra computation, the current implementation represents an adequate performance-precision trade-off.

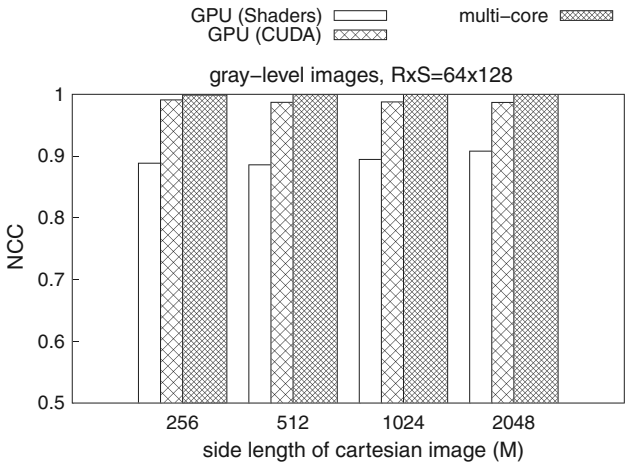

Fig. 11 Similarity of the log-polar images computed with the parallel architectures with that computed with the sequential algorithm (single-core CPU)

4.6 Conceptual and implementation complexity

The design and implementation complexity of the three considered parallelizations are as follows: The implementation based on multi-cores relies heavily on the existing sequential algorithm and code, and only a reduced number of OpenMP directives (`pragmas`) have had to be added to the sequential code in order to extract parallelism. More specifically, the loop-level parallelism is attained using the `pragma omp parallel for` construct, with additional clauses to denote the visibility of some variables and scheduling policies. The OpenMP runtime provided by the GNU compiler is in charge of orchestrating the parallel execution of the code.

The Shaders-based implementation is somehow more elaborate than that based on multi-cores, but it benefits from the simplicity of high-level programming model offered by GLSL. What is interesting in the Shaders solution is that it is portable to any GPU, unlike the CUDA solution which assumes Nvidia GPUs. Finally, the CUDA-based solution requires more effort in its design and implementation, and additional low-level design considerations have had to be made. The benefits of the three solutions in terms of time speed-ups correlate well with their implementation efforts. Therefore, the studied solutions are, in this sense, interestingly cost-effective.

Similarly, in economic terms, multi-core processors are generally the least expensive hardware, which also make the studied solutions cost-effective.

5 Discussion

Given the practical interest of fast implementations of the log-polar transform, this work has focused on proposing, evaluating and comparing three different parallel implementations based on GPU and multi-core processors. It has been shown that significant speed-ups with respect to a sequential implementation can be obtained, which makes these solutions appealing for a number of applications requiring log-polar images.

As a practical example, real-time (30 frames/s) log-polar transforms are possible, including the time for the image transfers between the GPU and the host CPU, for gray-level images as big as 4096×4096 pixels, or for 1024×1024 color images. Certainly, for most tasks, further processing must be performed on the log-polar images after they are obtained, and this cost must be included to estimate the frame rate. However, since efficient algorithms have been developed for many of these tasks, this processing cost can be easily assumed since the log-polar transform is generally the main bottleneck. For instance, a projection-based motion estimation algorithm [45] runs

quite fast (20 ms per frame on a modest CPU). Furthermore, some algorithms, such as this motion estimator, lend themselves to be parallelized, which would turn real-time processing even more feasible, and would make the most of having the images already in the GPU. These are interesting future directions to explore.

Another possibility is to devise a pipeline-like procedure so that a sequence of images such as those coming from a video stream might be processed in $O(1)$ temporal cost since computations performed on individual frames would dilute in the pipeline strategy.

The good performance improvements obtained can be explained by the fact that the log-polar transform would lie in "data processing dominated" application type [10], without decisions nor order dependencies. Not withstanding this, for some particular applications it may be convenient to force the transform to proceed in some particular order for the benefit of the overall performance. Further work might explore the design and performance issues involved in these scenarios.

In general, a trade-off between visual quality and speed may be desired. In the Shader case, if a given application does not require a perfect visual quality, less precision can be chosen, with the corresponding possible speed-up. Precision can be increased either with modern GPUs or by performing a two-pass algorithm on older GPUs. The higher precision with newer GPUs would not affect the speed-up. However, the visual quality would improve with two-pass algorithm at the expense of a lower speed-up. Therefore, it is the requirements of the final application that would determine the appropriate precision-speed trade-off. In our case, high visual quality was not an issue and we were particularly interested in exploring the speed-up with faster implementations on available (older) GPUs.

While the parallelization of the forward log-polar transform is probably the one more directly useful in most applications, two other parallelization tasks may be studied. On the one hand, the acceleration of the backward transform would be useful in those contexts where the reconstructed cartesian images are important *per se*, typically because they are meant to be visualized by some human user. This situations tend to arise in problems where transmission delays are to be minimized. In fact, the high data compression rates achievable with log-polar imaging was explored in the past for applications such as videotelephony [51] or teleoperation [49]. On the other hand, the map defining the log-polar transformation is usually the same if the conversion parameters remain constant. This is often the case in many applications and, in this sense, the cost of off-line computing this map is of no concern. However, if the parameters of the best transform are task-dependent and have to be decided on-line [46], quick recomputation of this map would be called for and parallel implementations of the map building itself would be appreciated.

Further considerations regarding variants of the log-polar transform itself (such as sub-pixel accuracy [26], Gaussian weighting for the RFs, overlapping RFs [12], etc.), and their correct adaptation to different architectures are also in the roadmap. Future work include the exploitation of the new capabilities of modern GPUs for this specific operation, and the selection of the Open Computing Language (OpenCL) infrastructure [3] to attain implementations that combine high-performance and portability to a wide range of graphics processors models (and even other type of accelerators).

6 Conclusions

In this work, implementations of the log-polar transform targeting a number of different parallel architectures (namely CUDA-capable GPUs, non-CUDA-capable GPUs and multi-cores) have been designed and thoroughly evaluated.

Several benefits can be drawn from the proposed parallel implementation. First, the quest of *high performance* was the main goal of this work and is the main contribution of the paper. Compared with a sequential implementation, our parallel implementations attain speed-up factors between $13\times$ and $22\times$ for *large* gray-level and color images, respectively, on the GPU, using CUDA as the implementation framework. Performance results for other GPU implementations and for multi-core processors have also shown to be cost-effective. These remarkable performance results offer the possibility of performing transformations of large images that otherwise would imply non-affordable execution times using a sequential implementation and thus open the door to new practical applications while enjoying the ease of parameter customization provided by software-based log-polar mappers.

Second, the *acquisition cost* of the utilized hardware allows fast log-polar transforms without the necessity of resorting to dedicated (and commonly expensive) high-performance hardware. Third, we have explored solutions targeting high- and low-level Nvidia GPUs (by using the CUDA infrastructure), GPUs from other manufacturers (by using Shaders-based implementations) or multi-cores (by using the widely utilized OpenMP standard), which offers *flexibility*.

This variety of implementations drives to a number of important insights regarding the most suitable architecture for a given scenario. Important factors as image size, transform parameters, impact of data transfers between memory spaces, etc. are of great relevance for the actual

performance brought by each specific implementation. Our experiments and analysis provide a detailed overview of the strengths and weaknesses of each platform and implementation under different conditions.

Finally, several interesting research avenues have also been identified along different dimensions, such as testing alternative programming languages and parallel architectures, designing and implementing variants of the log-polar transform, parallelizing related procedures, or exploring and improving the performance for particular, related application domains.

Acknowledgments The authors acknowledge the funding from the Spanish research programme Consolider Ingenio-2010 CSD2007-00018, from Fundació Caixa-Castelló Bancaixa under project P1-1A2010-11, from project DPI-2008-06636 and FPI grant BES-2009-027151 from the Spanish Ministerio de Ciencia e Innovación, and the research fellowship PREDOC/2006/02 from Jaume I University.

Source code availability The source code of the implementations is available at http://vision.uji.es/∼vtraver/logpolar.

References

1. CImg.: http://cimg.sourceforge.net. Accessed Sept 2012
2. CUDA.: http://developer.nvidia.com/object/gpucomputing.html. Accessed Sept 2012
3. OpenCL.: http://www.khronos.org/opencl. Accessed Sept 2012
4. OpenGL.: http://www.opengl.org. Accessed Sept 2012
5. OpenMP.: http://www.openmp.org. Accessed Sept 2012
6. Visual Computer Vision on GPUs.: http://www.cs.unc.edu/∼jmf/CV_GPU_program.html. CVPR 08 Workshop (2008). Accessed Sept 2012
7. Babenko, P., Shah, M.: MinGPU: a minimum GPU library for computer vision. J. Real-Time Image Process. **3**(4), 255–268 (2008)
8. Belongie, S., Malik, J., Puzicha, J.: Shape matching and object recognition using shape contexts. PAMI **24**(4), 509–522 (2002). doi:10.1109/34.993558
9. Bernardino, A., Santos-Victor, J.: Visual behaviors for binocular tracking. Robot. Auton. Syst. **25**, 137–146 (1998)
10. Blake, G., Dreslinski, R., Mudge, T.: A survey of multicore processors. IEEE Signal Process. Mag. **26**(6), 26–37 (2009)
11. Blythe, D.: The Direct3D 10 system. ACM Trans. Graph **25**(3), 724–734 (2006)
12. Bolduc, M., Levine, M.D.: A real-time foveated sensor with overlapping receptive fields. Real Time Imaging **3**(3), 195–212 (1997)
13. Böttger, J., Balzer, M., Deussen, O.: Complex logarithmic views for small details in large contexts. IEEE Trans. Visual. Comput. Graphics **12**((5), 845–852 (2006)
14. Bustos, P., Recio, F., Guinea, D., Garcia-Alegre, M.C.: Cortical representations in active vision on a network of transputers. In: ECPD International Conference on Advanced Robotics and Intelligent Automation, pp. 259–264, Athens, Greece (1995)
15. Capurro, C., Panerai, F., Sandini, G.: Dynamic vergence using log-polar images. Int. J. Comput. Vis. **24**(1), 79–94 (1997)
16. Chen, T., Budnikov, D., Hughes, C., Chen, Y.K.: Computer vision on multi-core processors: articulated body tracking. In: IEEE International Conference on Multimedia and Expo, 2007, pp. 1862–1865 (2007)
17. Chen, Y., Chakrabarti, C., Bhattacharyya, S., Bougard, B.: Signal processing on platforms with multiple cores. Part 2—applications and design [from the guest editors]. IEEE Signal Process. Mag. **27**(2), 20–21 (2010)
18. Chen, Y.K., Chakrabarti, C., Bhattacharyya, S., Bougard, B.: Signal processing on platforms with multiple cores. Part 1—overview and methodologies [from the guest editors]. IEEE Signal Process. Mag. **26**(6), 24–25 (2009)
19. CVGPU—Computer Vision on GPUs.: http://www.cvgpu.org. ECCV 10 Workshop, Crete, Greece (2010)
20. El-Mahdy, A., El-Shishiny, H.: An efficient load-balancing algorithm for image processing applications on multicore processors. In: Proceedings of the 1st international forum on Next-generation multicore/manycore technologies, IFMT'08, pp. 8:1–8:5. ACM, New York (2008)
21. Fisher, T.E., Juday, R.D.: A programmable video image remapper. In: SPIE Conference on Pattern Recognition and Signal Processing, vol. 938 (Digital and Optical Shape Representation and Pattern Recognition), pp. 122–128 (1988)
22. Franchetti, F., Püschel, M., Voronenko, Y., Chellappa, S., Moura, J.M.F.: Discrete Fourier transform on multicore. IEEE Signal Process. Mag. **26**(6), 90–102 (2009)
23. Group, K.: OpenGL shading language (GLSL), v. 4.1. http://www.opengl.org/documentation/glsl (2010)
24. Hartley, T.D.R., Ç atalyü rek, Ü.V., Ruiz, A., Igual, F.D., Mayo, R., Ujaldon, M.: Biomedical image analysis on a cooperative cluster of GPU s and multicores. In: ACM International Conference on Supercomputing (ICS), pp. 15–25 (2008)
25. Hoan, L., Youngjae, C., Kyoungsu, O.: Image mosaic using log-polar binning. In: First Asian Conference on Pattern Recognition (ACPR), pp. 144–148 (2011)
26. Jurie, F.: A new log-polar mapping for space variant imaging. Application to face detection and tracking. Pattern Recognit. **32**, 865–875 (1999)
27. Kim, D., Lee, V., Chen, Y.K.: Image processing on multicore x86 architectures. IEEE Signal Process. Mag. **27**(2), 97–107 (2010)
28. Kim, H., Bond, R.: Multicore software technologies. IEEE Signal Process. Mag. **26**(6), 80–89 (2009)
29. Massone, L., Sandini, G., Tagliasco, V.: 'Form-Invariant' topological mapping strategy for 2D shape recognition. Comput. Vis. Graph. Image Process. **30**, 169–188 (1985)
30. NVIDIA.: CUDA best practices guide (version 3.0). http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide.pdf (2010)
31. Pardo, F., Dierickx, B., Scheffer, D.: Space-variant nonorthogonal structure CMOS image sensor design. J. Solid-State Circ. **33**(6), 842–849 (1998)
32. Peng, F., Guo, R.S., Li, C.T., Long, M.: A semi-fragile watermarking algorithm for authenticating 2D CAD engineering graphics based on log-polar transformation. Comput.-Aid. Des. **42**(12), 1207–1216 (2010)
33. Poli, G., Saito, J.H., Mari, J.a.F., Zorzan, M.R.: Processing neocognitron of face recognition on high performance environment based on GPU with CUDA architecture. In: Proceedings of the 20th International Symposium on Computer Architecture and High Performance Computing, pp. 81–88 (2008)
34. Powell, K.: Biomedical imaging ecosystem and the role of the GPU . In: IEEE International Symposium on Biomedical Imaging: From Nano to Macro (ISBI'09) , pp. 1291–1292 (2009)
35. Rofouei, M., Stathopoulos, T., Ryffel, S., Kaiser, W., Sarrafzadeh, M.: Energy-aware high performance computing with graphic processing units. In: Proceedings of the 2008 conference on Power aware computing and systems (HotPower'08) (2008)
36. Saini, N., Sinha, A.: Optics based biometric encryption using log polar transform. Opt. Commun. **283**(1), 34–43 (2010)

37. Samsi, S., Gadepally, V., Krishnamurthy, A.: MATLAB for signal processing on multiprocessors and multicores. IEEE Signal Process. Mag. **27**(2), 40–49 (2010)
38. Sandini, G., Dario, P., DeMicheli, M., Tistarelli, M.: Retina-like CCD sensor for active vision. In: Computer and Systems Sciences (NATO ARW on Robots and Biological Systems). Springer, Il Ciocco, Tuscany (1989)
39. Schwartz, E.L., Greve, D.N., Bonmassar, G.: Space-variant active vision: definition, overview and examples. Neural Networks **8**(7–8), 1297–1308 (1995)
40. Setoain, J., Prieto, M., Tenllado, C., Tirado, F.: GPU for parallel on-board hyperspectral image processing. Int. J. High Perform. Comput. Appl **22**, 424–437 (2008)
41. Shin, C.W., Inokuchi, S., Kim, K.I.: Retina-like visual sensor for fast tracking and navigation robots. Mach. Vis. Appl. **10**, 1–8 (1997)
42. Slabaugh, G., Boyes, R., Yang, X.: Multicore image processing with OpenMP. IEEE Signal Process. Mag. **27**(2), 134–138 (2010)
43. del Solar, J.R., Nowack, C., Schneider, B.: VIPOL: a virtual polar-logarithmic sensor. In: Scandinavian Conference on Image Analysis (SCIA), pp. 739–744, Finland (1997)
44. Traver, V.J., Bernardino, A.: A review of log-polar imaging for visual perception in robotics. Robot. Auton. Syst. **58**(4), 378–398 (2010)
45. Traver, V.J., Pla, F.: Similarity motion estimation and active tracking through spatial-domain projections on log-polar images. Comput. Vis. Image Underst. **97**(2), 209–241 (2005)
46. Traver, V.J., Pla, F.: Log-polar mapping template design: from task-level requirements to geometry parameters. Image Vis. Comput. **26**(10), 1354–1370 (2008)
47. Wallace, R.S., Ong, P.W., Bederson, B.B., Schwartz, E.L.: Space variant image processing. Int. J. Comput. Vis. **13**(1), 71–90 (1994)
48. Wang, Y.C., Donyanavard, B., Cheng, K.T.: Energy-aware real-time face recognition system on mobile CPU-GPU platform. In: http://www.cvgpu.org. ECCV 10 Workshop, Crete, Greece (2010)
49. Weiman, C.F.R.: Video compression via log-polar mapping. In: SPIE Symposium on OE/Aerospace Sensing, Orlando, Florida (1990)
50. Wilson, J.C., Hodgson, R.M.: Log-polar mapping applied to pattern representation and recognition. In: Computer Vision and Image Processing, pp. 245–277 (1992)
51. Woelders, W.W., Frowein, H.W., Nielsen, J., Questa, P., Sandini, G.: New developments in low-bit rate videotelephony for people who are deaf. J. Speech Lang. Hearing Res. **40**, 1425–1433 (1997)
52. Wong, W.K., Choo, C.W., Loo, C.K., Teh, J.: FPGA implementation of log-polar mapping. Int. J. Comput. Appl. Technol. **39**(1/2/3), 12–18 (2010)
53. Wong, W.K., Choo, C.W., Loo, C.K., Teh, J.P.: FPGA implementation of log-polar mapping. In: 15th International Conference on Mechatronics and Machine Vision in Practice (M2VIP08), Auckland, New Zealand (2008)
54. Wong, W.K., Loo, C.K., Lim, W.S., Tan, P.N.: Thermal condition monitoring system using log-polar mapping, quaternion correlation and max-product fuzzy neural network classification. Neurocomputing **74**(1–3), 164–177 (2010)
55. Zhang, X., Tay, L.P.: A spatial variant approach for vergence control in complex scenes. Image Vis. Comput. **29**(1), 64–77 (2011)
56. Zokai, S., Wolberg, G.: Image registration using log-polar mappings for recovery of large-scale similarity and projective transformations. IEEE Trans. Image Process. **14**(10), 1422–1434 (2005)

Author Biographies

Marco Antonelli received the M.Sc. degree in computer engineering, with an industrial automation specialism, from Universita degli Studi di Padova, Padova, Italy, in 2008. He is currently working towards the Ph.D. degree at the Robotic Intelligence Lab, Universitat Jaume I, Castellón, Spain. His research is mainly focused on biologically inspired humanoid robotics. He has published in international conferences and journals and participated to specialized courses on the subject. He is collaborating with the Active Perception Lab (Boston University) to develop a neural model of egocentric representation of the 3-D space based on both monocular and binocular visual cues.

Francisco D. Igual received his BS degree from the University Jaume I in Castellón in 2006, and his MS degree from the same University in 2007. He obtained his PhD in Computer Science in 2011 from the Department of Computer Science and Engineering of the University Jaume I. His research interests include the optimization and adaptation of numerical libraries to platforms with one GPU or multiGPU systems, runtime support for automatic parallelization of numerical codes, and development of accelerated routines for distributed memory architectures. Additionally, he is involved in the implementation and optimization of biomedical and image processing applications on hardware accelerators.

Francisco Ramos is an associate professor in the Department of Computer Languages and Systems at the University Jaume I of Castellón. He got his bachelor and master degrees in computer science from this university. He got his Ph.D. with honors from University Jaume I of Castellon in 2008. His research interests are in the areas of computer graphics, geometric modeling, visualization, GIS and mobile devices.

V. Javier Traver earned a B.Sc. degree in Computer Science, Technical University of Valencia (Valencia, Spain) in 1995, and a Ph.D. degree in Computer Engineering, Jaume-I University (Castellón, Spain) in 2002. He is an Associate Professor of Computer Engineering at Jaume-I University. His research at the Institute of New Imaging Technologies includes video analysis, foveal imaging and active vision.