Journal of Real-Time Image Processing (2021) 18:2441–2451
https://doi.org/10.1007/s11554-021-01129-4

ORIGINAL RESEARCH PAPER

Parallel algorithm for fringe pattern demodulation

Francisco J. Hernandez-Lopez[1] · **Ricardo Legarda-Sáenz**[2] · **Carlos Brito-Loeza**[2]

Received: 26 September 2020 / Accepted: 12 May 2021 / Published online: 8 June 2021

Abstract

In this paper, we present a parallel implementation of a fixed-point algorithm for finding the solution of the total variation model for phase demodulation. The total variation model is efficient in estimating discontinuous phase maps, background illumination, and amplitude modulation from a single fringe pattern. The implementations include execution in a multi-core CPU and a GPU using OpenMP and CUDA, respectively. We show performance comparisons of the parallel implementations with 64-bit and 32-bit precision floating-point numbers using synthetic and real experimental data. Results show that our parallel implementations achieve speedups over the serial implementation of 9x for multi-core CPU and 103x for GPU.

Keywords Fringe analysis · Total variation · Multi-core · GPU

1 Introduction

Fringe analysis is a widely used technique in optical metrology to recover physical quantities such as displacement, strain, surface profile, and refractive index from interferograms. Interferograms are two-dimensional recordings made by a digital camera of interference patterns. Encoded in the interference fringes or bands of the interferogram is the shape of the wavefront [13]. Fringe analysis is then the extraction of the quantitative measurement data from either a single fringe pattern or a collection of them [25]. Fringe analysis consists of one or two processes: phase demodulation and phase unwrapping [30].

The mathematical model of a fringe pattern is given by

$$I(x, y) = a(x, y) + b(x, y)\cos(\psi(x, y) + \phi(x, y)), \qquad (1)$$

where $I(x, y)$ is the image intensity, $a = a(x, y)$ is the background illumination, $b = b(x, y)$ is the amplitude modulation, $\phi = \phi(x, y)$ is the phase, and $\psi = \psi(x, y)$ is the spatial carrier frequency. The main task of fringe analysis algorithms is to recover the phase term ϕ, and in some cases, the background illumination and amplitude modulation as well. That is an inverse problem, because only the fringe pattern $I(x, y)$ and sometimes the carrier frequency ψ are known.

In the last years, many techniques have appeared for the solution to the problem mentioned above. They depend on obtaining one or more fringe patterns from the experiments. For instance, phase-shifting techniques recover the phase map ϕ by acquiring a collection of fringe patterns shifted one from another by a certain amount [32]. The phase difference between two consecutive fringe patterns is usually a constant term [1, 8, 15, 29, 30].

Among the variety of techniques for demodulating the phase map using a single fringe pattern, Takeda's method was one of the first to be proposed. This method is based on the Fourier transform and considers the phase map as a continuous and smooth function [33]. In recent years, some papers appeared with solutions based on regularized Bayesian estimation costs. Regularization uses a priori information to impose restrictions on the estimated solution. The success rate of these techniques at recovering the phase term ϕ from highly noisy patterns is high; however, their numerical solution is usually quite expensive [18, 21, 28]. An extensive review of these methods is available in [30] and references therein.

All demodulation methods described above fail to recover discontinuous or piece-wise phase maps. Up to our knowledge, there are very few methods reported in the literature capable of recovering these kinds of phase maps [2, 3, 9, 17, 31, 36]. One of them, based on a regularized cost function,

✉ Francisco J. Hernandez-Lopez
fcoj23@cimat.mx

1 CONACYT – Centro de Investigación en Matemáticas A.C., CIMAT Unidad Mérida, PCTY, Sierra Papacal, 97302 Mérida, YUC, Mexico

2 CLIR at Facultad de Matemáticas, Universidad Autónoma de Yucatán, Mérida, YUC, Mexico

uses a second-order edge-preserving potential [9]; another is based on wavelets using total variation (*TV*) regularization [36]. The rest are based on variational formulations using either *TV* or mean curvature regularization [20, 26]. Except for the reference [31], these methods estimate the phase term, the background illumination, and the amplitude modulation from a single fringe pattern. We remark that all methods described in this paragraph report serial realizations for their models. Consequently, their running CPU times are much slower than the parallel implementation of our model presented here.

For a model to succeed in the transition from applied research to industrial applications, a reliable and fast numerical realization of the model needs to be available. Despite the accurate recovery of the phase term, the computational times reported in the demodulation techniques discussed above show the need to provide them with fast computational solvers. In this work, we address this issue for the *TV* model by introducing a very quickly parallel realization of this model based on the fixed-point algorithm introduced in [17] and whose convergence was proved in [3].

Other methods for solving the model in [17] are a very slow gradient descent algorithm and a recently published augmented Lagrangian algorithm [16]. The realization of both algorithms is serial hence slower than the parallel one introduced here.

The outline of this paper is as follows. In Sect. 2, we review shortly the *TV* model. In Sect. 3, we review the fixed-point algorithm. In Sect. 4, we present the parallel realization for multi-core CPU and GPU architectures. The experimental results on both synthetic and experimental data are presented in Sect. 5 and our conclusions are given in Sect. 6.

2 Total variation based model

The *TV* model presented in [17] amounts to solving the following problem:

$$\underset{a,b,\phi}{\arg\min} \quad TV = TV(a, b, \phi, \psi, g), \tag{2}$$

where

$$\begin{aligned} TV \equiv & \int_\Omega (I-g)^2 \mathrm{d}\Omega + \frac{1}{\lambda_a}\int_\Omega |\nabla a| \mathrm{d}\Omega \\ & + \frac{1}{\lambda_b}\int_\Omega |\nabla b|\mathrm{d}\Omega + \frac{1}{\lambda_\phi}\int_\Omega |\nabla \phi|\mathrm{d}\Omega, \end{aligned} \tag{3}$$

where $\Omega \subseteq \mathbb{R}^2$, $g = g(x, y)$ is the acquired fringe pattern, and $\lambda_a, \lambda_b, \lambda_\phi$ are positive regularization parameters.

Due to the total variation regularization, this model can recover sharp phase-transitions, background illumination, and amplitude modulation.

The solution of (2) is obtained by numerically solving the following set of second-order nonlinear Euler–Lagrange equations, one for each variable:

$$-\nabla \cdot \frac{\nabla a}{|\nabla a|} + \lambda_a (a + bc_\phi - g) = 0, \tag{4}$$

$$-\nabla \cdot \frac{\nabla b}{|\nabla b|} + \lambda_b (a + bc_\phi - g)c_\phi = 0, \tag{5}$$

$$-\nabla \cdot \frac{\nabla \phi}{|\nabla \phi|} + \lambda_\phi (a + bc_\phi - g)(-bs_\phi) = 0 \tag{6}$$

with $c_\phi = \cos(\psi + \phi)$, $s_\phi = \sin(\psi + \phi)$, and boundary conditions

$$\frac{\partial a}{\partial \nu} = 0, \quad \frac{\partial b}{\partial \nu} = 0, \quad \frac{\partial \phi}{\partial \nu} = 0, \tag{7}$$

where ν denotes the unit outer normal vector to the boundary.

3 The fixed-point algorithm

In this work, we focus on developing a parallel realization of a fixed-point algorithm framework for solving each partial differential equation (PDE) presented in (4–6). This algorithm introduced in [3] is convergent for any initial guess, and its structure is suitable for parallelization. We proceed to review the fixed-point algorithm.

To solve each PDE, an algorithm of the form (for general u)

$$L(u^k)u^{k+1} = f^k \tag{8}$$

is constructed, where $L = L(u)$ is a linearized operator given by

$$L = \begin{cases} -\nabla \cdot \frac{\nabla}{|\nabla a^k|} + \lambda_a I & \text{in (4)} \\ -\nabla \cdot \frac{\nabla}{|\nabla b^k|} + \lambda_b c_{\phi^k}^2 & \text{in (5)} \\ -\nabla \cdot \frac{\nabla \phi^k}{|\nabla \phi^k|} + \lambda_\phi b^2 s_{\phi^k}^2 & \text{in (6)} \end{cases}$$

and

$$f^k = \begin{cases} \lambda_a(-b^k c_{\phi^k} + g) & \text{in (4)} \\ \lambda_b(-a^k + g)c_{\phi^k} & \text{in (5)} \\ -\lambda_\phi s_{\phi^k}(-a^k b^k + gb^k - \\ (b^k)^2(c_{\phi^k} - s_{\phi^k}\phi^k)) & \text{in (6).} \end{cases}$$

Therefore, given an arbitrary initial guess, each fixed-point algorithm constructs a convergent sequence of solutions of the type $\{u^k\}_{k\geq 1}$. Note that L is a linear operator that in (4) and (5) is obtained by lagging the nonlinear diffusion coefficients at every k-iteration and in (6) using the Taylor expansion of first order of the cosine function. The linear system (8) does not need to be solved very accurately, and few iterations of any sparse linear solver suffice. The operator L has some nice properties; it is symmetric, positive definite, and diagonally dominant for a and b, even though it is only semi-positive definite and weakly diagonally dominant for ϕ.

4 Parallel implementation

We develop two parallel implementations in C/C++ of the serial phase demodulation algorithm (see Algorithm 1): the first one uses a multiple core system (here referred to as multi-core CPU) and the second one uses a GPU-based architecture. We use OpenMP for multi-thread programming on the multi-core CPU system, while on the GPU architecture, we use CUDA programming. We also use the open computer vision library OpenCV [12] and the high-performance vector mathematics library Blitz++ [34] in both implementations: OpenCV only for reading and writing images, and Blitz++ for array management in the multi-core CPU.

We now proceed to describe the discretization scheme for the PDEs. For this purpose, let $\Omega = [0, n] \times [0, m]$ be a continuous domain and let (h_x, h_y) represent a vector of finite mesh sizes. Then, the discrete domain Ω_h can be defined as $\Omega_h = \Omega \cap G_h$, where $G_h = \{(x, y) : x = x_i = ih_x, y = y_j = jh_y; i, j \in \mathbb{Z}\}$ is an infinite grid. Take u as an $n \times m$ array where each entry $u_{i,j}$ for $i = 1, \ldots, n$ and $j = 1, \ldots, m$ is the discrete value of the continuous variable on the grid Ω_h at some point (x, y). In what follows, we use u to represent any of the variables a, b, or ϕ.

Algorithm 1 evaluates the TV functional in (3), the boundary conditions (BC), and the Gauss–Seidel (GS) method in an iterative way, until the normalized error Q between previous and current solutions is less than a given threshold value ϵ or the maximum number of iterations MaxIter is achieved. The procedure in Algorithm 2 computes the value of (3) by approximating the gradient operator $\nabla u_{i,j}$ as follows:

$$\nabla u_{i,j} = \left(\frac{u_{i+1,j} - u_{i,j}}{h_x}, \frac{u_{i,j+1} - u_{i,j}}{h_y} \right). \tag{9}$$

The BC procedure computes the Neumann boundary conditions with the following equations:

$$\begin{aligned} u_{i,1} &= u_{i,2}, u_{i,m} = u_{i,m-1}, \\ u_{1,j} &= u_{2,j}, u_{n,j} = u_{n-1,j}, \end{aligned} \tag{10}$$

for $i = 1, \ldots, n$ and $j = 1, \ldots, m$.

Let $\mathbf{v}_{i,j} = [u_{i+1,j}, u_{i-1,j}, u_{i,j+1}, u_{i,j-1}]$ be a column vector containing the four neighbors of u and let

$$\begin{aligned} \mathbf{w}_{i,j} &= [w_1, w_2, w_3, w_4] \\ &= \left[\frac{1}{|\nabla u_{i+1,j}|_\beta}, \frac{1}{|\nabla u_{i-1,j}|_\beta}, \frac{1}{|\nabla u_{i,j+1}|_\beta}, \frac{1}{|\nabla u_{i,j-1}|_\beta} \right] \end{aligned}$$

be the corresponding vector of regularized nonlinear terms approximated by

$$|\nabla u_{i+1,j}|_\beta = |\nabla u_{i,j+1}|_\beta = \sqrt{(u_x^1)^2 + (u_y^1)^2 + \beta}, \tag{11}$$

$$|\nabla u_{i-1,j}|_\beta = \sqrt{(u_x^2)^2 + (u_y^2)^2 + \beta}, \tag{12}$$

$$|\nabla u_{i,j-1}|_\beta = \sqrt{(u_x^3)^2 + (u_y^3)^2 + \beta}, \tag{13}$$

where $\beta > 0$ is a small parameter to avoid division by zero, and

$$\begin{aligned} u_x^1 &= (u_{i+1,j} - u_{i,j})/h_x, u_y^1 = (u_{i,j+1} - u_{i,j})/h_y, \\ u_x^2 &= (u_{i,j} - u_{i-1,j})/h_x, u_y^2 = (u_{i-1,j+1} - u_{i-1,j})/h_y, \\ u_x^3 &= (u_{i+1,j-1} - u_{i,j})/h_x, u_y^3 = (u_{i,j} - u_{i,j-1})/h_y \end{aligned}$$

are the derivatives approximated by finite differences. In our simulations and without loss of generality, we considered the spatial step sizes to be equal in both directions, that is, $h = h_x = h_y$. For the regularization parameters, it was enough to select them all equal, i.e., $\lambda = \lambda_a = \lambda_b = \lambda_\phi$.

The GS procedure shown in Algorithm 3 computes an approximate solution of a, b, and ϕ using the GS method with red-black ordering. The update of each variable is as follows:

$$u_{i,j}^{q+1} = \frac{\mathbf{v}_{i,j}^T \mathbf{w}_{i,j} + f_{i,j}}{D_{i,j} + 2w_1 + w_2 + w_4}, \tag{14}$$

where all the terms in the right hand are evaluated at the qth iteration, and

$$D_{i,j} = \begin{cases} \lambda & \text{in (4)} \\ \lambda c_{\phi_{i,j}}^2 & \text{in (5)} \\ \lambda b_{i,j}^2 s_{\phi_{i,j}}^2 & \text{in (6).} \end{cases}$$

For computing the normalized error Q, we use

$$Q(z_1, z_2) = \|z_1 - z_2\| / (\|z_1\| + \|z_2\|), \quad (15)$$

as reported in [19, 24]. This equation defines a relative error without considering physical dimensions. The Q values live in the interval [0, 1], with Q approaching zero, while z_1 and z_2 get closer to each other.

For the *TV* and Q evaluations, we need a parallel reduction procedure [4, 5] to compute the corresponding sum. For the *BC* procedure, only assignment statements are necessary, and they are independent of each other. In both implementations, serial and parallel, we use Gauss–Seidel with red-black ordering; thus, our serial and parallel results are the same. The following subsections describe the implementation of these procedures.

Algorithm 1 Phase demodulation using total variation and fixed-point.

Input: $a^0, b^0, \phi^0, \psi, g, \lambda, \zeta$
Output: Phase map ϕ^*
$f^0 = TV(a^0, b^0, \phi^0, \psi, g, \lambda)$
$k = 0$
repeat
  $BC(a^k, b^k, \phi^k)$
  $\tilde{a}^1 = a^k$, $\tilde{b}^1 = b^k$, $\tilde{\phi}^1 = \phi^k$
  for $q = 1$ to ζ **do**
    $(\tilde{a}^{q+1}, \tilde{b}^{q+1}, \tilde{\phi}^{q+1}) = GS(\tilde{a}^q, \tilde{b}^q, \tilde{\phi}^q, \psi, g, \lambda, 0)$
    $(\tilde{a}^{q+1}, \tilde{b}^{q+1}, \tilde{\phi}^{q+1}) =$
    $GS(\tilde{a}^{q+1}, \tilde{b}^{q+1}, \tilde{\phi}^{q+1}, \psi, g, \lambda, 1)$
  end for
  $a^{k+1} = \tilde{a}^q$, $b^{k+1} = \tilde{b}^q$, $\phi^{k+1} = \tilde{\phi}^q$
  $f^{k+1} = TV(a^{k+1}, b^{k+1}, \phi^{k+1}, \psi, g, \lambda)$
  $\Delta f = Q(f^{k+1}, f^k)$
  $\Delta a = Q(a^{k+1}, a^k)$
  $\Delta b = Q(b^{k+1}, b^k)$
  $\Delta \phi = Q(\phi^{k+1}, \phi^k)$
  $k = k + 1$
until $(k > MaxIter) || (\Delta f < \epsilon) || (\Delta a < \epsilon) || (\Delta b < \epsilon) || (\Delta \phi < \epsilon)$
$\phi^* = \phi^k$, $a^* = a^k$, $b^* = b^k$

Algorithm 2 Evaluation of the TV functional.

procedure $\mathrm{TV}(a, b, \phi, \psi, g, \lambda)$
  $h_x = 1/(n-1)$
  $h_y = 1/(m-1)$
  $s = 0$
  for $i = 1$ to n **do**
    for $j = 1$ to m **do**
      $d = a_{i,j} + b_{i,j} cos(\psi_{i,j} + \phi_{i,j}) - g_{i,j}$
      $s = s + \frac{1}{2}\lambda d^2 + |\nabla a_{i,j}| + |\nabla b_{i,j}| + |\nabla \phi_{i,j}|$
    end for
  end for
  return $s h_x h_y$
end procedure

Algorithm 3 GS with red-black ordering.

function $\mathrm{GS}(a, b, \phi, \psi, g, \lambda, R\_B)$
  for $i = 2$ to $n - 1$ **do**
    for $j = 2$ to $m - 1$ **do**
      if $(i + j)\%2 == R\_B)$ **then**
        Update $a_{i,j}$, $b_{i,j}$ and $\phi_{i,j}$ with (14)
      end if
    end for
  end for
  return a, b, ϕ
end function

4.1 OpenMP implementation

OpenMP is an API for shared-memory parallel programming in a multi-core CPU architecture [23]. Thus, OpenMP is suitable for systems in which each thread or process can access all available memory. OpenMP provides a set of directives or pragmas used to specify parallel regions. Among other things, these directives are efficient to manage threads inside parallel regions and to distribute *for loops* in parallel.

For the *TV* and Q evaluations, we use the directive `#pragma omp parallel for reduction(+ : sum)`. For the BC procedure, we use the directive `#pragma omp parallel for`. In the Gauss–Seidel algorithm with red-black ordering [7], the pixels are considered red or black following a chessboard pattern. We consider a pixel $r = (i, j)$ red if $i + j$ is even and black if $i + j$ is odd (see Algorithm 3). Then, when the red pixels are updated in the *for loop*, they only need the black pixel values and vice versa. This reordering aims to get an equivalent equation system in which there are more independent computations [27], resulting in efficient parallel implementations of the *GS*.

The directive `#pragma omp parallel` opens a parallel region and with the `for` directive, parallelizes the *for loop*. In shared-memory programs, the individual threads have private and shared memory. Communication is accomplished through shared variables. Inside the `#pragma omp for`, all variables are shared by default for all threads; then, we use only the `private` directive to declare private variables for each thread.

Additionally, we use the directive `num threads(N_T)` to specify the number of threads $\mathtt{N_T}$ to be launched in our program.

4.2 CUDA implementation

CUDA (*compute unified device architecture*) is an extension to the C language that contains a set of instructions for parallel computing in a GPU. The host processor spawns multi-thread tasks (Kernels) onto the GPU device, which has its internal scheduler that will then allocate the kernels to whatever GPU hardware is present [6]. The GPU is used for general-purpose computation; it contains multiple transistors for the arithmetic logic unit, based on the single instruction and multiple threads (SIMT) programming model, which is exploited when multiple data are managed from a single parallel instruction, similar to the single instruction multiple data (SIMD) model [5, 14].

Following the conventional programming model in GPU, once $a^0, b^0, \phi^0, \psi, g$ are allocated in the CPU memory, we reserve their corresponding memory space in the GPU device. Then, these variables are loaded in the GPU device from the CPU through a memory copy process. In our program, we define four main kernel functions for the *TV*, *BC*, *GS*, and *Q* procedures. For the *TV* and *Q* procedures, we implemented a parallel reduction with dynamic shared memory and the interleaved pair strategy [5]. Hence, the size of the array is divided into thread blocks; in each thread block, a partial sum is computed using shared memory; then, these partial sums are copied back to the CPU memory and summed in the CPU. Note that the last sum is computed in parallel in the multi-core CPU.

Another way to implement the reduction of the partial sum is using the atomic function `atomicAdd`. This function reads a value from some address in global or shared memory, adds a number to it, and writes the result in the same address; no other thread can access that address until the operation is complete. In this way, we avoid the memory copy of the partial sums using only the GPU for all processing. A consideration to take into account is that `atomicAdd` is only supported by Nvidia GPUs of computing capability 6 and higher when 64-bit floating-point data are used (see the atomic functions section in [22] for more details).

5 Experimental results

The experiments were executed on a server with Intel(R) Xeon(R) Gold 5222 CPU 3.80 GHz, Ubuntu 18.04 (64-bits), 16 hyper-threading cores, 48GB RAM, and a video card Nvidia Quadro RTX 8000 with compute capability 7.5. We fix the parameters $\lambda = 10$, $MaxIter = 5 \times 10^5$, and $\epsilon = 10^{-7}$. For the *GS* procedure, only a few iterations ζ are used as recommended in [3]; thus, we fix $\zeta = 4$. We develop two versions of our serial and parallel implementations, one version using 64-bit floating-point data (double precision) and the other using 32-bit floating-point data (float precision).

We note that similar models for demodulating discontinuous maps [2, 9] are only equipped with gradient descent algorithms with serial realizations. Therefore, their running times are very slow compared with those obtained with our parallel algorithm. For instance, in [17], it was reported a processing time of 800 s to solve the problem of Fig. 2 for an image of size 250×250 pixels using the model in [9], while a processing time of 144 seconds was reported for the same problem also using a gradient descent algorithm for the *TV* model [17]. We ran our parallel algorithm for the same problem obtaining a processing time at least ten times faster than the one reported in [17]. We remark that serial gradient descent algorithms do not scale well with the size of the images, while our parallel algorithm does. Furthermore, the computational realization of the model in [2] is even slower, being that a fourth-order and highly nonlinear PDE has to be solved.

In our experiments, Fig. 1 shows the number of iterations, processing time in seconds, and normalized error Q between the desired phase ϕ and the estimated phase ϕ^* of our parallel phase demodulation implementation with double precision. As is shown there, we ran the GPU simulations for different values of β and image sizes. We can see that $\beta = 10^{-4}$ is a good compromise between processing time and precision; we use this value for the rest of our synthetic experiments.

Figure 2 shows a result of our phase demodulation method with double precision using synthetic data. The images have a size of 240×320 pixels. The first row shows the spatial carrier frequency ψ, the fringe pattern g, and the desired phase to estimate ϕ, respectively. The second row shows the initial value of background illumination a^0, amplitude modulation b^0, and phase estimation ϕ^0 at iteration $k = 0$. The third row shows the optimal estimations a^*, b^*, and ϕ^*. We obtain a normalized error $Q(\phi, \phi^*) = 0.0279$.

The speedup of a parallel program is defined as

$$S = \frac{T_s}{T_p}, \tag{16}$$

where T_s is the processing time of serial program and T_p is the processing time of parallel program [23].

Figure 3 shows the speedup of our parallel implementation using the multi-core CPU with double precision, launching from $N_T = 2$ to $N_T = 32$ threads. The server has 16 hyper-threading cores, and then, each physical core is divided into two virtual or logical cores, sharing resources such as the instruction pointers, integer registers, floating-point registers, scheduling queues, caches, and execution units. The performance of a parallel implementation declines provided a virtual-core monopolizes some critical resources such as the floating-point registers or the caches. Increasing the performance of a parallel implementation

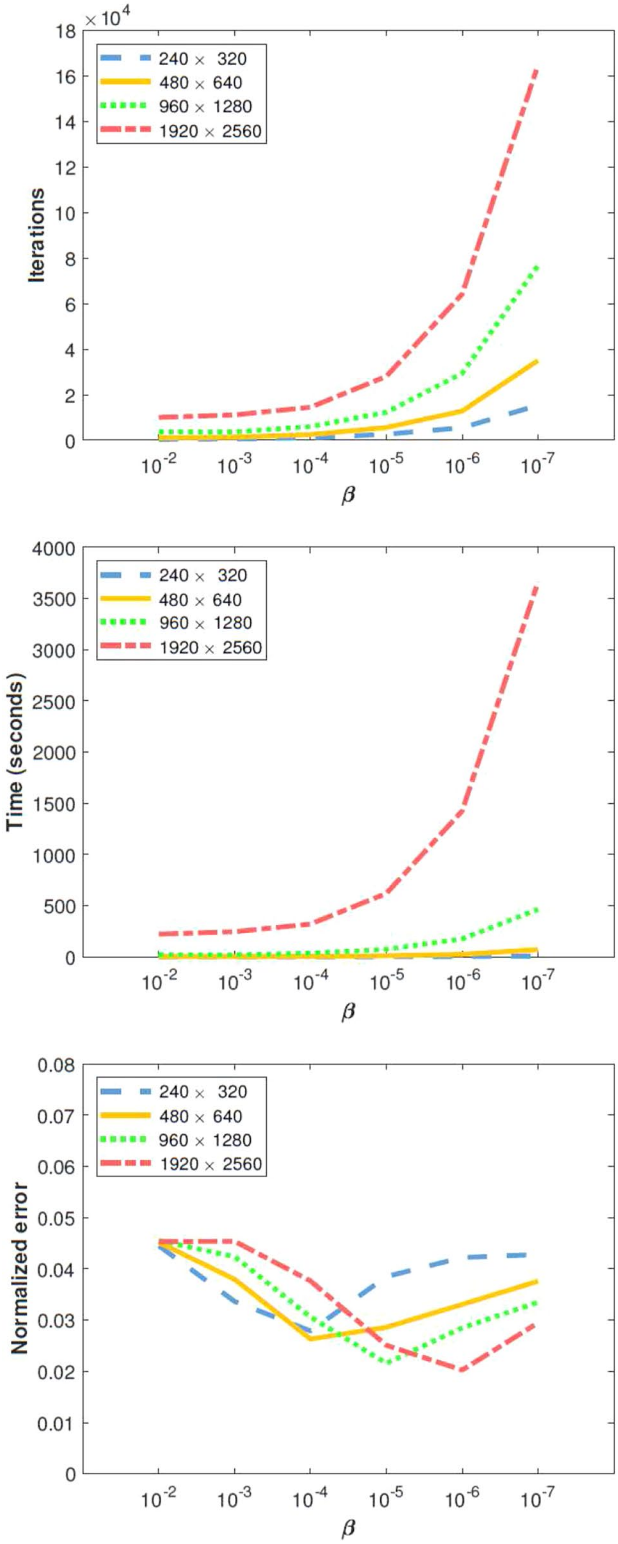

Fig. 1 Number of iterations, processing time, and normalized error of our parallel phase demodulation method, for different values of β and different image sizes

is fundamentally an optimization problem, which is very difficult due to different memory hierarchies between platforms and the variation of core connection on a single processor [11]. We achieve the best speedup in our parallel implementation when launching $N_T = 16$ threads for the different image sizes, while for $N_T > 16$, the performance declines.

Figure 4 shows the speedup of our parallel implementation using the GPU with double precision. In our parallel GPU implementation, we split the reduction process into two sums. The first one is executed in the GPU, and the second one can be executed in parallel in the CPU. We launch from $N_T = 2$ to $N_T = 32$ threads in our multi-core CPU; note that the speedup is almost constant for $N_T <= 16$ and has a small decline for $N_T > 16$. We can see that it is enough to use only $N_T = 2$ threads to obtain the best performance, demonstrating that the more demanding process is computed in the GPU.

Tables 1, 2, 3, and 4 show a comparative performance of our serial and parallel implementations with double and float precision respectively, for different image sizes of our synthetic data. The processing time is measured from the start of Algorithm 1 until it terminates, which means that we are considering both the processor work and the memory transfer time between the CPU and the GPU. We execute the parallel implementations 20 times in each experiment, reporting the mean and standard deviation of the number of iterations k, the processing time, and the normalized error Q. We can see that our implementations using GPU have the shortest times.

When floating-point arithmetic is used, the rounding errors can lead to unexpected results. There are calculations with real numbers that produce quantities that are not exactly represented in float or double precision [10]. Rounding modes for basic operations, including subtraction, multiplication, and division, are specified in the IEEE-754 standard; the most frequently used is the round-to-nearest mode [35]. In the results of our implementations using double precision (see Tables 1 and 2), we obtain the same number of iterations and the same normalized error to 12 decimal places.

The TV and Q procedures of our algorithm need to compute a summation of all elements of an array of the size of the image to be processed; this operation may propagate rounding errors as can be seen in Tables 3 and 4, where we use float precision. Note that the number of iterations is different concerning each implementation. It is important to note that the serial-float implementation does not scale linearly with the size of the images. We expect the algorithm to get a better speedup for larger size images than that one for smaller ones. Table 3 shows

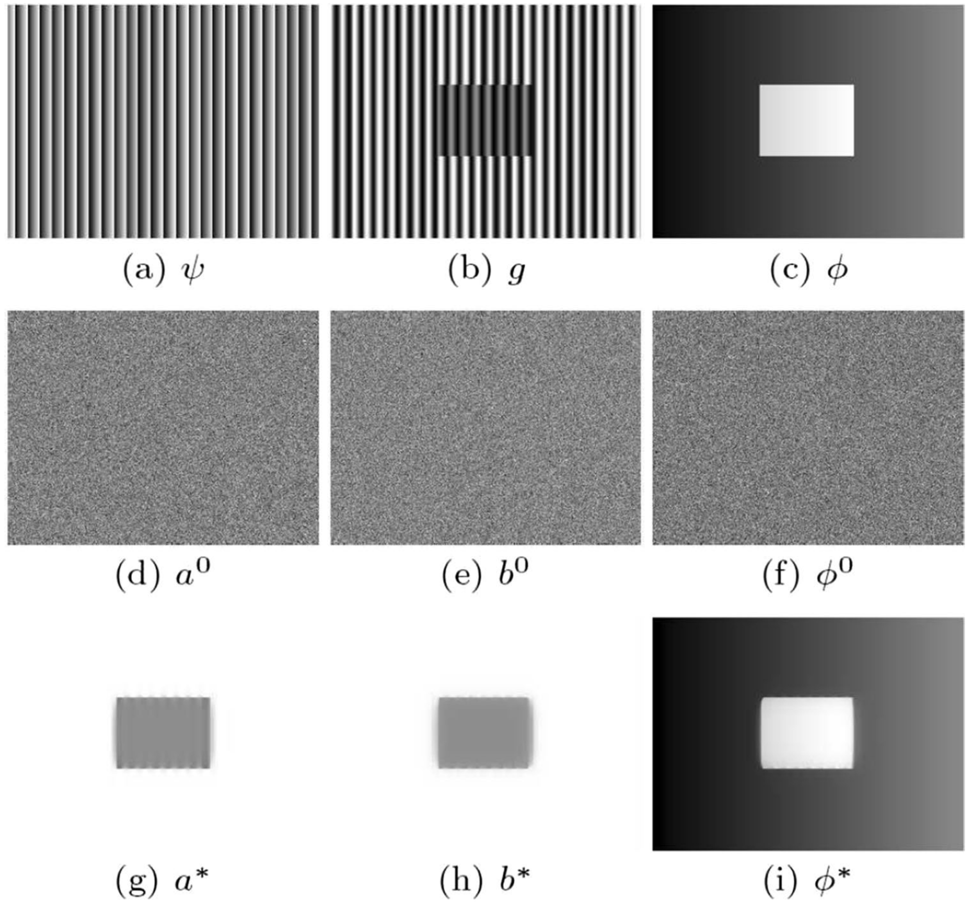

Fig. 2 Phase demodulation using synthetic data

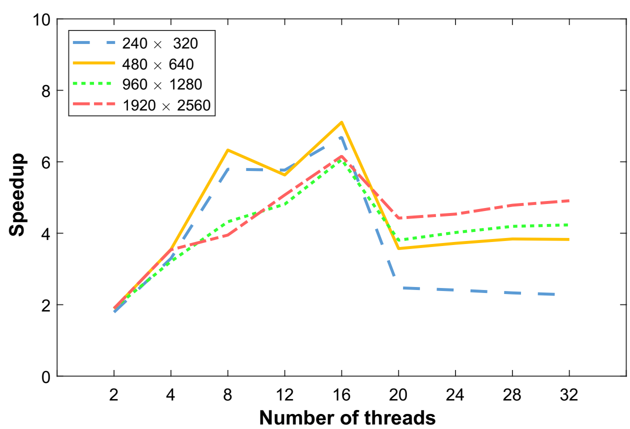

Fig. 3 Evaluation of speedups of parallel phase demodulation algorithm, using the multi-core CPU from 2 to 32 threads for different image sizes

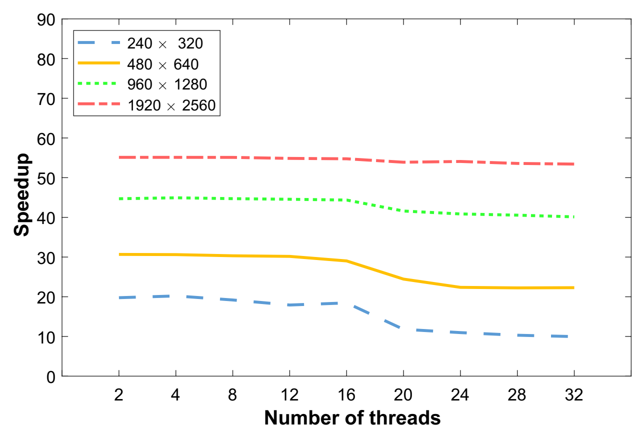

Fig. 4 Evaluation of speedups of parallel phase demodulation algorithm, using a GPU and the multi-core CPU from 2 to 32 threads for different image sizes

that this is not true for the image of size 1920×2560, for which the speedup is inferior to the one obtained for the image of size 960×1280. If we ignore this result, then the normalized errors are the same to four decimal places for all our implementations using float precision. They are the same to three decimal places taken into account the results with double precision. However, even when we consider the result of the serial implementation for an image of size 1920×2560, the normalized error is the same as rounding to two decimal places for all implementations with a float or double precision, which is not bad for the quality of the recovered phase map.

Table 1 Number of iterations, processing time (seconds), and normalized error Q of our serial and multi-core CPU implementations using double precision

Size of image	Serial			Multi-core CPU – 16C		
	Iter.	Time(s)	Q	Iter.	Time(s)	Q
240×320	1095	15.98	0.0278956108250142	1095	2.24 ± 0.02	0.0278956108250159
480×640	2510	159.08	0.0263059900148534	2510	21.63 ± 0.25	0.0263059900148562
960×1280	6031	1663.94	0.0306234524498726	6031	274.61 ± 4.25	0.0306234524498944
1920×2560	14526	17459.06	0.0376542821313397	14526	2921.45 ± 100.52	0.0376542821314670

Table 2 Number of iterations, processing time (seconds), and normalized error Q of our GPU implementations using double precision

Size of image	GPU – 2C			GPU – *atomicAdd*		
	Iter.	Time(s)	Q	Iter.	Time(s)	Q
240×320	1095	0.60 ± 0.02	0.0278956108250159	1095	0.60 ± 0.02	0.0278956108250159
480×640	2510	3.91 ± 0.02	0.0263059900148565	2510	3.87 ± 0.02	0.0263059900148565
960×1280	6031	32.86 ± 0.03	0.0306234524498950	6031	32.23 ± 0.02	0.0306234524498951
1920×2560	14526	308.02 ± 0.32	0.0376542821314705	14526	304.15 ± 0.35	0.0376542821314704

Table 3 Number of iterations, processing time (seconds), and normalized error Q of our serial and multi-core CPU implementations using float precision

Size of image	Serial			Multi-core CPU – 16C		
	Iter.	Time(s)	Q	Iter.	Time(s)	Q
240×320	1032	13.63	0.0279578808695077	1055 ± 8	1.69 ± 0.13	$0.027935 \pm 6.10e{-}6$
480×640	2283	117.59	0.0263535343110561	2365 ± 18	13.38 ± 0.17	$0.026329 \pm 3.72e{-}6$
960×1280	5435	1179.32	0.0307095143944025	5559 ± 46	138.81 ± 1.36	$0.030689 \pm 9.26e{-}6$
1920×2560	9079	8242.48	0.0429241806268692	13045 ± 79	1415.15 ± 10.10	$0.037793 \pm 1.28e{-}5$

Table 4 Number of iterations, processing time (seconds), and normalized error Q of our GPU implementations using float precision

Size of image	GPU – 2C			GPU – *atomicAdd*		
	Iter.	Time(s)	Q	Iter.	Time(s)	Q
240×320	1075	0.23 ± 0.08	0.0279189944267272	1079 ± 6	0.24 ± 0.09	$0.027916 \pm 4.41e{-}06$
480×640	2460	1.53 ± 0.08	0.0263126417994499	2432 ± 21	1.52 ± 0.07	$0.026316 \pm 3.24e{-}06$
960×1280	5766	11.89 ± 0.07	0.0306537337601184	5561 ± 63	11.41 ± 0.18	$0.030690 \pm 1.37e{-}05$
1920×2560	13341	107.82 ± 0.12	0.0377521179616451	13153 ± 159	104.49 ± 1.25	$0.037780 \pm 2.54e{-}05$

On the other hand, the results of our $GPU - 2C$ implementation with float precision (see Table 4) show that the number of iterations and normalized errors have zero standard deviation. They are the same in each execution for each size of the image. That is indeed a requirement in a deterministic algorithm.

To compare our parallel implementations, Figs. 5 and 6 show the speedup of each implementation for different image sizes using double and float precision, respectively. For the case of the multi-core CPU, we launch from $N_T = 2$ to $N_T = 20$ threads, while for the case of $GPU - 2C$, we launch only $N_T = 2$ threads in the CPU, and for the case of $GPU - atom$, we use the `atomicAdd` function to avoid some memory copies between CPU and GPU. Note that GPU implementations have almost the same speedup level in each of our experiments, achieving the best performance

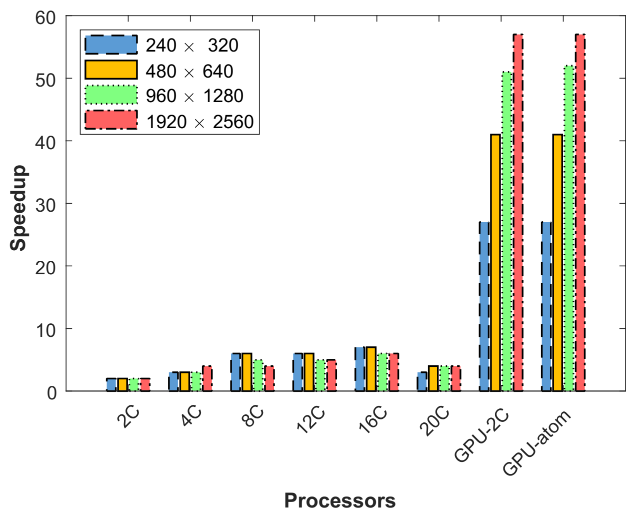

Fig. 5 Evaluation of speedups of parallel phase demodulation algorithm, using the multi-core CPU (from 2 to 20C), GPU with 2C, and the GPU with `atomicAdd` for different image sizes and double precision

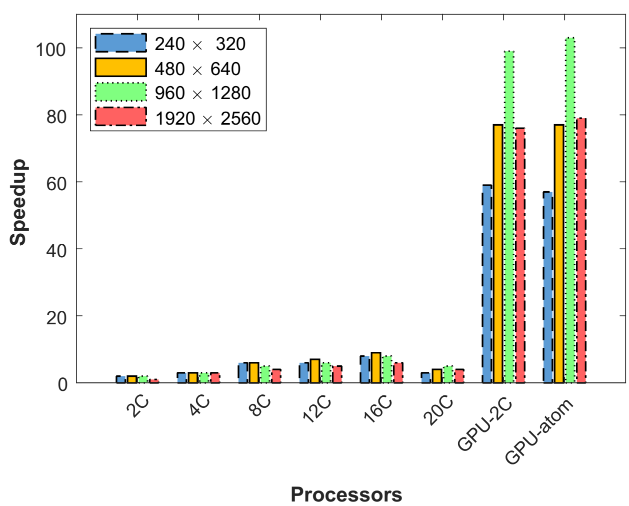

Fig. 6 Evaluation of speedups of parallel phase demodulation algorithm, using the multi-core CPU (from 2 to 20C), GPU with 2C, and the GPU with `atomicAdd` for different image sizes and float precision

for all image sizes. The advantage of $GPU - atom$ is that all computations are realized in the GPU. Thus, other programs can be running in the CPU without compromising performance. For the multi-core CPU implementation, the speedup goes from $S = 2$ to $S = 7$, and for the GPU implementations, it goes from $S = 25$ to $S = 58$ using double precision; while using float precision for multi-core CPU implementation, the speedup goes from $S = 4$ to $S = 9$, and for the GPU implementations, it goes from $S = 57$ to $S = 103$.

The first row in Fig. 7 shows the results of our parallel phase demodulation method with double and float precision when using real experimental data. The presented data consist of two images (ψ and g) of size 480×640 pixels. The second row shows the results with $\lambda = 10$ and $\beta = 10^{-4}$ (the same parameters as those used for the synthetic data experiments); the algorithm takes $k = 16269$ iterations, 880.84 s in serial version and 24.68 s in $GPU - 2C$ version with double precision, and takes $k = 13402$ iterations and 7.96 s in $GPU - 2C$ version with float precision. The third row shows the results when using $\lambda = 2$ and $\beta = 10^{-7}$; here, the algorithm takes $k = 45173$ iterations, 2446.78 s in the serial version and 68.58 s in the $GPU - 2C$ version with double precision, and takes $k = 41455$ iterations and 23.25 s in the $GPU - 2C$ version with float precision. Qualitatively, we can see that the results using double and float precision are very similar, even when the algorithm converges at a different number of iterations.

6 Conclusions

In this paper, we presented parallel implementations of the fixed-point algorithm for solving the total variation phase demodulation model. We presented results from synthetic and real experiments in a server with 16 hyper-threading cores and a GPU.

The multi-core CPU implementation achieves the best speedup (9x) using 16 threads. The two GPU versions of our implementations ($GPU - 2C$ and $GPU - atom$) obtain almost the same speedup. That outperforms the speed of the multi-core CPU implementations. In general, the GPU implementations achieve a speedup of up to 103x over the serial implementation.

The results show that the parallel implementations converge and obtain the same number of iterations and normalized error to 12 decimal places when we use double precision. In contrast, due to the propagation of rounding errors, we obtain a different number of iterations when using float precision; however, the resultant normalized errors of our parallel implementations with float precision are the same to three decimal places as that for double precision. Qualitatively, the estimated phases are very similar.

As future work, we intend to explore the usage of multigrid algorithms to speed the algorithm even further.

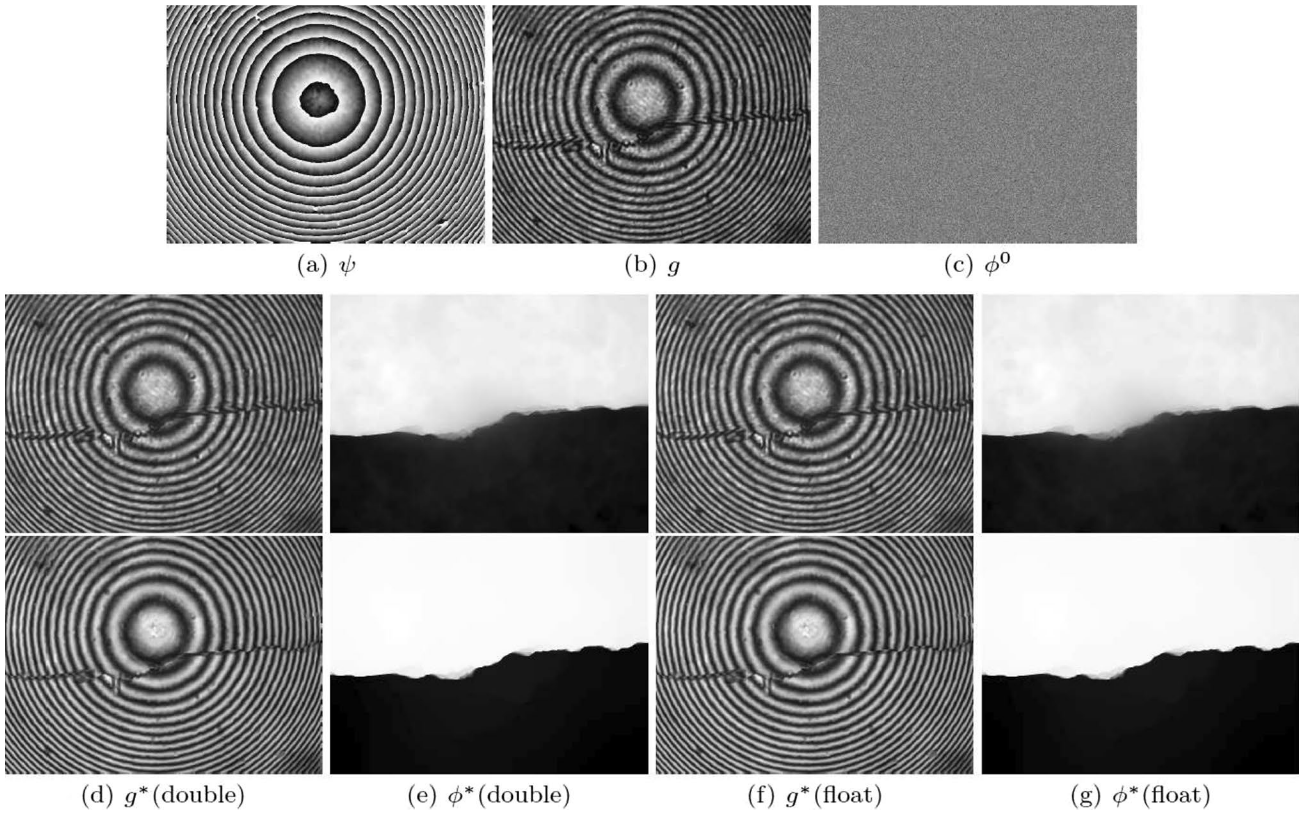

Fig. 7 Phase demodulation using real experimental data

Acknowledgements Authors acknowledge the support from "Laboratorio de Supercómputo del Bajío" through the Grant Number 300832 from CONACyT.

References

1. Ayubi, G.A., Duarte, I., Perciante, C.D., Flores, J.L., Ferrari, J.A.: Phase-step retrieval for tunable phase-shifting algorithms. Opt. Commun. **405**(June), 334–342 (2017). https://doi.org/10.1016/j.optcom.2017.08.045
2. Brito-Loeza, C., Legarda-Saenz, R., Espinosa-Romero, A., Martin-Gonzalez, A.: A mean curvature regularized based model for demodulating phase maps from fringe patterns. Commun. Comput. Phys. **24**(1), 27–43 (2018). https://doi.org/10.4208/cicp.OA-2017-0109
3. Brito-Loeza, C., Legarda-Saenz, R., Martin-Gonzalez, A.: A fast algorithm for a total variation based phase demodulation model. Numer. Methods Partial Differ. Equ. **36**(3), 617–636 (2020)
4. Chapman, B., Jost, G., Van Der Pas, R.: Using OpenMP: Portable Shared Memory Parallel Programming, vol. 10. MIT Press, Cambridge (2008)
5. Cheng, J., Grossman, M., McKercher, T.: Professional CUDA C Programming. Wiley, Indianapolis, Indiana (2014). https://www.books.google.com.mx/books?id_Z7rnAEACAAJ
6. Cook, S.: CUDA Programming: A Developer's Guide to Parallel Computing with GPUs, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco, CA (2012)
7. Demmel, J.W.: Applied Numerical Linear Algebra. Society for Industrial and Applied Mathematics, Philadelphia, PA (1997)
8. Flores, V.H., Reyes-Figueroa, A., Carrillo-Delgado, C., Rivera, M.: Two-step phase shifting algorithms: where are we? Opt. Laser Technol. **126**(January), 106105 (2020). https://doi.org/10.1016/j.optlastec.2020.106105
9. Galvan, C., Rivera, M.: Second-order robust regularization cost function for detecting and reconstructing phase discontinuities. Appl. Opt. **45**(2), 353–359 (2006). https://doi.org/10.1364/AO.45.000353
10. Goldberg, D.: What every computer scientist should know about floating-point arithmetic. ACM Comput. Surv. (CSUR) **23**(1), 5–48 (1991)
11. Hwu, W.M., Keutzer, K., Mattson, T.G.: The concurrency challenge. IEEE Des. Test. Comput. **25**(4), 312–320 (2008). https://doi.org/10.1109/MDT.2008.110
12. Itseez: OpenCV. Website (2020). http://opencv.org//. Accessed 24 Sept 2020
13. Karpinsky, N., Zhang, S.: High-resolution, real-time 3d imaging with fringe analysis. J. Real Time Image Process. **7**(1), 55–66 (2012)
14. Kirk, D.B., Wen-Mei, W.H.: Programming Massively Parallel Processors: A Hands-on Approach (Applications of GPU Computing Series), 1st edn. Morgan Kaufmann, Burlington, MA, USA (2010).
15. Kulkarni, R., Rastogi, P.: Two-step phase demodulation algorithm based on quadratic phase parameter estimation using state space analysis. Opt. Lasers Eng. **110**(April), 41–46 (2018). https://doi.org/10.1016/j.optlaseng.2018.05.012
16. Legarda-Saenz, R., Brito-Loeza, C.: Augmented lagrangian method for a total variation-based model for demodulating phase

discontinuities. J. Algorithm Comput. Technol. **14**, 1–8 (2020). https://doi.org/10.1177/1748302620941413
17. Legarda-Saenz, R., Brito-Loeza, C., Espinosa-Romero, A.: Total variation regularization cost function for demodulating phase discontinuities. Appl. Opt. **53**(11), 2297–2301 (2014)
18. Legarda-Saenz, R., Osten, W., Juptner, W.P.: Improvement of the regularized phase tracking technique for the processing of non-normalized fringe patterns. Appl. Opt. **41**(26), 5519–5526 (2002). https://doi.org/10.1364/AO.41.005519
19. Legarda-Saenz, R., Tellez Quinones, A., Brito-Loeza, C., Espinosa-Romero, A.: Variational phase recovering without phase unwrapping in phase-shifting interferometry. Int. J. Comput. Math. **96**(6), 1217–1229 (2019)
20. Vese, L.A., Le Guyader, C.: Variational Methods in Image Processing, 1st edn. Chapman and Hall/CRC, Abingdon, UK (2015)
21. Marroquin, J.L., Rivera, M., Botello, S., Rodriguez-Vera, R., Servin, M.: Regularization methods for processing fringe-pattern images. Appl. Opt. **38**(5), 788–794 (1999). https://doi.org/10.1364/AO.38.000788
22. NVIDIA Corporation: CUDA C++ Programming Guide. Website (2020). https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed 24 Sept 2020
23. Pacheco, P.: An Introduction to Parallel Programming, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco, CA (2011)
24. Perlin, M., Bustamante, M.D.: A robust quantitative comparison criterion of two signals based on the sobolev norm of their difference. J. Eng. Math. **101**(1), 115–124 (2016)
25. Rajshekhar, G., Rastogi, P.: Fringe analysis: premise and perspectives. Opt. Lasers Eng. **50**(8), iii–x (2012). https://doi.org/10.1016/j.optlaseng.2012.04.006
26. Rudin, L.I., Osher, S., Fatemi, E.: Nonlinear total variation based noise removal algorithms. Phys. D **60**(1–4), 259–268 (1992). https://doi.org/10.1016/0167-2789(92)90242-F
27. Rünnger, G., Rauber, T.: Parallel Programming: for Multicore and Cluster Systems, 2nd edn. Springer-Verlag, Berlin Heidelberg (2013)
28. Servin, M., Marroquin, J.L., Cuevas, F.J.: Fringe-follower regularized phase tracker for demodulation of closed-fringe interferograms. J. Opt. Soc. Am. A **18**(3), 689–695 (2001). https://doi.org/10.1364/JOSAA.18.000689
29. Servin, M., Padilla, M., Choque, I., Ordones, S.: Phase-stepping algorithms for synchronous demodulation of nonlinear phase-shifted fringes. Opt. Express **27**(4), 5824 (2019). https://doi.org/10.1364/OE.27.005824
30. Servin, M., Quiroga, J.A., Padilla, M.: Fringe Pattern Analysis for Optical Metrology: Theory, Algorithms, and Applications. Wiley-VCH, Weinheim (2014)
31. Singh, M., Khare, K.: Single-shot interferogram analysis for accurate reconstruction of step phase objects. J. Opt. Soc. Am. A **34**(3), 349 (2017). https://doi.org/10.1364/JOSAA.34.000349
32. Surrel, Y.: Fringe Analysis. In: P.K. Rastogi (ed.) Photomechanics, Topics in Applied Physics, vol. 77, pp. 55–102. Springer, Berlin, Heidelberg (2000). https://doi.org/10.1007/3-540-48800-6_3
33. Takeda, M., Ina, H., Kobayashi, S.: Fourier-transform method of fringe-pattern analysis for computer-based topography and interferometry. J. Opt. Soc. Am. **72**(1), 156 (1982). https://doi.org/10.1364/JOSA.72.000156
34. Veldhuizen, T.L.: Arrays in blitz++. In: Caromel, D., Oldehoeft, R.R., Tholburn, M. (eds.) Computing in Object-Oriented Parallel Environments, pp. 223–230. Springer, Berlin, Heidelberg (1998)
35. Whitehead, N., Fit-Florea, A.: Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs. Technical report, rn (A+ B) **21**(1), 18749–19424 (2011)
36. Zhu, X., Tang, C., Li, B., Sun, C., Wang, L.: Phase retrieval from single frame projection fringe pattern with variational image decomposition. Opt. Lasers Eng. **59**, 25–33 (2014). https://doi.org/10.1016/j.optlaseng.2014.03.002

Francisco J. Hernandez-Lopez received the B.E. degree in computer systems engineering from the Instituto Tecnológico de San Luis Potosí, México, in 2005. He received the M.Sc. and D.Sc. degrees in Computer Science from the Centro de Investigación en Matemáticas (CIMAT), México, in 2009 and 2014 respectively. Since 2014, he is in the Computer Science Department at the CIMAT, Mérida, México. His main interests are in the area of computer vision, machine learning and in the development of efficient algorithms using parallel computing for the processing and analysis of video sequences. He is fellow of the National System of Researchers (SNI) of the Mexican Government.

Ricardo Legarda-Sáenz received his MSc degree in electronic engineering (computation) from the Instituto Tecnológico de Chihuahua (México) in 1997, and his PhD in optics from the Centro de Investigaciones en Óptica (México) in 2000. Since 2004, he has been a professor at Universidad Autónoma de Yucatán, México. His current interests are image processing applied to fringe pattern analysis.

Carlos Brito-Loeza received a Ph.D. degree in mathematics from the University of Liverpool, U.K. He is currently a professor at the Faculty of Mathematics in the Universidad Autónoma de Yucatán, México where he is part of the computational learning and imaging research group. His research interests are on variational modeling and PDE based methods with applications to different imaging and learning problems. He also works on developing efficient numerical solvers for a diversity of scientific computing applications.