

# A comparison of multi-hardware parallel implementations for a phase unwrapping algorithm

Francisco J. Hernandez-Lopez<sup>a,\*</sup>, Mariano Rivera<sup>b,c</sup>, Adan Salazar-Garibay<sup>d</sup>,  
Ricardo Legarda-Saenz<sup>e</sup>

<sup>a</sup>CONACYT Centro de Investigación en Matemáticas A.C., CIMAT Unidad Mérida, PCTY, Zip code 97302 Sierra Papacal, Mérida, Yucatán, México.

<sup>b</sup>Centro de Investigación en Matemáticas A.C., CIMAT, Guanajuato, Gto., México.

<sup>c</sup>Centro Nacional de Supercómputo IPICYT, San Luis Potosí, S.L.P., México.

<sup>d</sup>CONACYT - Centro de Investigación en Geografía y Geomática, CentroGEO Unidad Mérida, Mérida, Yucatán, México.

<sup>e</sup>CLIR at Facultad de Matemáticas, Universidad Autónoma de Yucatán, Mérida, Yucatán, México.

**Abstract.** Phase unwrapping is an important problem in the areas of Optical Metrology, Synthetic Aperture Radar (SAR) image analysis and Magnetic Resonance Imaging (MRI) analysis. These images are becoming larger in size and, particularly, the availability and need for processing of SAR and MRI data has increased significantly with the acquisition of remote sensing data and the popularization of magnetic resonators in clinical diagnosis. Therefore, it is important to develop faster and accurate phase unwrapping algorithms. In this paper, we propose a parallel multigrid algorithm of a phase unwrapping method named accumulation of residual maps (ARM), which builds on a serial algorithm that consists of the minimization of a cost function; minimization achieved by means of a serial Gauss-Seidel kind algorithm. Our algorithm also optimizes the original cost function, but unlike the original work, our algorithm is a parallel Jacobi class with alternated minimizations. This strategy is known as the chessboard type where red pixels can be updated in parallel at same iteration since they are independent. Similarly, black pixels can be updated in parallel in an alternating iteration. We present parallel implementations of our algorithm for different parallel multicore architecture such as CPU-multicore, Xeon Phi coprocessor and Nvidia GPU. In all the cases, we obtain a superior performance of our parallel algorithm when compared to the original serial version. In addition, we present a detailed comparative of the performance of the developed parallel versions.

**Keywords:** Phase unwrapping, SAR interferograms, parallel computing, multicore CPU, Xeon Phi, GPU.

\*Francisco J. Hernandez-Lopez, [fcoj23@cimat.mx](mailto:fcoj23@cimat.mx)

## 1 Introduction

Phase unwrapping is an important problem in the areas of Optical Metrology, Synthetic Aperture Radar (SAR) image analysis and Magnetic Resonance Imaging (MRI) analysis.

In this paper, the case study is developed for unwrapping large dimensions SAR interferograms<sup>1-3</sup> using parallel computing.

A SAR signal contains amplitude and phase information. Amplitude is the strength of the radar response and phase is the fraction of one complete sine wave cycle.<sup>4</sup> Therefore, the SAR interferogram is generated by two complex SAR images that observe the same area from slightly different look angles. This can be done with two radars mounted on the same platform or with a radar at different times by exploiting repeated orbits of the same satellite. Then the interferogram is obtained by cross-multiplying, pixel by pixel, the first SAR image  $u_1 = |u_1|e^{i\phi_1}$  with the complex conjugate of the second  $u_2^* = |u_2|e^{-i\phi_2}$ .<sup>5-7</sup> Thus, the interferogram amplitude is the amplitude of the first image multiplied by that of the second one  $|u_1||u_2|$ , whereas its interferometric phase is the phase difference between the images  $\phi = \phi_1 - \phi_2$ .

The relationship between the actual phase  $\phi$  and the computed wrapped phase  $g$  can be mathematically expressed by

$$g_r = W\{\phi_r + \eta_r\}, \quad (1)$$

where  $r$  indicates a pixel position in a regular lattice  $\mathcal{L}$ ;  $\eta_r$  represents noise and in the case

of SAR phase it is correlated with the signal magnitude; finally,

$$W\{z\} \stackrel{def}{=} z + 2n\pi, \quad (2)$$

is the non-linear wrapping operator, with  $n$  an integer such that  $W\{z\} \in (-\pi, \pi]$ . If the Nyquist criterion is violated,

$$\phi_r + \eta_r - \phi_s - \eta_s \neq W(g_r - g_s), \quad (3)$$

with  $s \in \mathcal{N}_r$  and  $\mathcal{N}_r = \{s \in \mathcal{L} : \|r - s\|_2 = 1\}$  the set of first neighbor pixels to the pixel  $r$ , then the unwrapping process needs be implemented as the solution to a ill-posed inverse problem (*i.e.* a problem with many possible solutions) in order to estimate the unwrapped phase  $f$ . To this end, the efficient accumulation of residual maps (ARM) method was recently reported in Ref. 8, which unwraps phase maps with the additional advantage that can be implemented in parallel.

Parallel computing consists of the simultaneous execution of calculations, instructions, processes, or tasks using more than one processor. There are different architectures to achieve parallel computing; for example: multicore computers, graphics processing units (GPUs), Xeon Phi coprocessors (XPCs), computer clusters and field programmable gate arrays (FPGAs). According to each one of these architectures, there are different program-

ing paradigms and languages, which exploit their capacities. Some of them are OpenMPI for clusters; OpenMP for multicore CPUs and XPCs; Cg, CUDA, OpenCL and OpenACC for GPUs and Xilinx tools for FPGAs. Furthermore, the architectures can be merged in order to improve the performance of some problems. It is possible to have a multicore computer with one or more GPUs, or to have a cluster with one or more GPUs and FPGAs in each node,<sup>9</sup> or a grid of GPUs, etc.

This work was planned to present a comparison as fair as possible between parallel implementations of the ARM method using a multicore CPU, XPC and GPU, in order to help implementers to take a decision in future parallel implementations of phase unwrapping algorithms that are being developed. In the state of the art it is common to find this type of comparisons using implementations of simple algorithms of linear algebra methods (vector and matrix multiplications, factorizations), transformations (Fourier transform), or interpolations (bilinear, bicubic, splines). These implementations are useful as benchmark tests but are far from reflecting the actual working conditions of current image analysis methods. Our study is complementary to such tests and throws another type of conclusions, more focused on the implementation of image analysis algorithms. For this reason, we are using a complex algorithm of the state of the art (multigrid phase unwrapping) implemented in different frameworks and we evaluate them in production conditions. The evaluated frameworks are more similar to the actual working conditions. These frame-

works are summarized in Table 1. Framework 1 consist of using MatLab with OpenMP directives into a mex-C file to execute the program in the multicore CPU or XPC, and using CUDA kernel integration in MatLab (without mex-C) to process in the GPU. Framework 2 consist of using C/C++ with OpenMP directives to execute the program in the multicore CPU or in the XPC, and with CUDA kernel functions to execute the program in a GPU. The hardware architectures we are not considering are computer clusters and FPGAs. Our reasons are that computer clusters seem not to be proper for cellular processes with high messaging interchange (the kind of processing often used in image processing), and FPGA requires of specific and dedicated hardware.

**Table 1** Parallel implementation frameworks.

<b>Architecture</b>	<b>Framework 1 (MatLab)</b>	<b>Framework 2 (C/C++)</b>
<b>Multicore CPU or XPC</b>	OpenMP with mex-C file	OpenMP
<b>GPU</b>	CUDA kernel integration in MatLab	CUDA kernels

The remaining sections of this paper are organized as follows: Section 2 gives a brief review of the serial ARM method. Section 3 describes the parallel implementation of the method, and gives the implementation details of the most demanding process. Sections 4 and 5 present a comparison of the processing time and speedup between the multicore CPU, XPC and GPU architectures, using simulated data and real data respectively. Section 5 also discusses the wrapped phase generation from two SAR images. Finally, Section 6

gives our remarks and conclusions.

## 2 Serial implementation of ARM method

The ARM method consists of an incremental scheme for unwrapping the wrapped phase

$g$ . Let

$$\rho_{rs} \stackrel{def}{=} W(g_r - g_s), \quad (4)$$

be the wrapped first differences of the wrapped phase and  $f^{(k)}$  be the current estimate of the unwrapped phase. Then we can compute the current residual wrapped differences as:

$$\rho_{rs}^{(k)} \stackrel{def}{=} W\{\rho_{rs} - f_r^{(k)} + f_s^{(k)}\}. \quad (5)$$

If the residual differences field  $\rho_{rs}^{(k)} \neq 0$ , then one could try to estimate an update field  $\delta^{(k)}$  such that:

$$\sum_{(r,s) \in \mathcal{L}} W\{\rho_{rs}^{(k)} - \delta_r^{(k)} + \delta_s^{(k)}\}^2 \leq \sum_{(r,s) \in \mathcal{L}} [\rho_{rs}^{(k)}]^2. \quad (6)$$

In any case, the field  $\delta$  will compensate only the conservative component of the residual differences field  $\rho_{rs}$ ; *i.e.* monopoles in the sense of wrapped phase could not be solved.<sup>10–13</sup>

In the ARM algorithm the current phase is updated (accumulated) with

$$f_r^{(k+1)} = f_r^{(k)} + \delta_r^{*(k)}, \quad (7)$$

the residual wrapped differences are computed with equation (5), and the update field  $\delta^{*(k)}$  is computed by minimizing the half-quadratic cost function:

$$\begin{aligned}
\delta^{*(k)}, \omega^{*(k)} &= \arg \min_{\delta^{(k)}, \omega^{(k)}} U(\delta^{(k)}, \omega^{(k)}; \rho^{(k)}) \\
&= \frac{1}{2} \sum_{r \in \mathcal{L}} \sum_{s \in \mathcal{N}_r} \left\{ (\omega_{rs}^{(k)})^2 [(\rho_{rs}^{(k)} - \delta_r^{(k)} + \delta_s^{(k)})^2 + \lambda(\delta_s^{(k)} - \delta_r^{(k)})^2] \right. \\
&\quad \left. + \mu(1 - \omega_{rs}^{(k)})^2 \right\}, \tag{8}
\end{aligned}$$

where  $\lambda$  and  $\mu$  are positive parameters of the algorithm (see Ref. 8 for a deeper discussion on their selection). The data term in equation (8) is a weighted version of the regularised Least-Square potential.<sup>13–15</sup> The second term [membrane potential:  $(\delta_s^{(k)} - \delta_r^{(k)})^2$ ] penalizes large local variations on the unwrapped phase,<sup>16,17</sup> which reduces noise.

The solution to equation (8) can be computed by alternating minimisation with respect to  $\delta^{(k)}$  and  $\omega^{(k)}$ . Thus, if  $\omega^{(k)}$  is fixed, then the solution of the positive-definite diagonal-dominant linear system given by  $\partial U / \partial \delta_r^{(k)} = 0$  can be computed with a Gauss-Seidel iterative scheme:

$$\delta_r^{*(k)} = \frac{\sum_{s \in \mathcal{N}_r} (\omega_{rs}^{(k)})^2 [\rho_{rs}^{(k)} + \delta_s^{(k)}(1 + \lambda)]}{\sum_{s \in \mathcal{N}_r} (\omega_{rs}^{(k)})^2 [1 + \lambda]}, \tag{9}$$

where the  $\delta_s^{(k)}$  (for  $s \in \mathcal{N}_r$ ) are the first neighbor values to the  $\delta_r^{(k)}$ . In the case of a Jacobi

update scheme,  $\delta_s^{(k)} \equiv \delta_s^{(k-1)}$ . In the Gauss-Seidel scheme, the updated values are used; *i.e.*, in a pixel scanning top-down/left-right, the left and upper pixels are at iteration  $k$  and the right and down pixels are at iteration  $k - 1$ . Our parallel implementation uses the red-black update rule, explained in Section 3. Similarly, from  $\partial U / \partial \omega_{rs}^{(k)} = 0$ , we obtain the closed formula:

$$\omega_{rs}^{*(k)} = \frac{\mu}{\mu + (\rho_{rs}^{(k)} - \delta_r^{(k)} + \delta_s^{(k)})^2 + \lambda(\delta_s^{(k)} - \delta_r^{(k)})^2}. \quad (10)$$

It is well known that Gauss-Seidel is prone to have a slow reduction of low-frequency residuals. Thus the convergence is accelerated by using a multigrid strategy;<sup>18</sup> this strategy is summarized in Algorithm 1. It shows a simple multigrid scheme where the solution at level  $N$  is used as initial guess for level  $N - 1$ . This proposal implements a full-nested multigrid strategy detailed in Algorithm 2. In the implementation of Algorithm 2, the computation of residual wrapped phase at the end of each iteration is required. The residual wrapped phase between two given phases  $g$  and  $f$  is denoted by

$$\text{residual}(g, f) \stackrel{\text{def}}{=} W(g - f). \quad (11)$$



To implement this function, one can use the formula

$$\text{residual}(g, f) = \text{atan2}\left(\sin(g - f), \cos(g - f)\right). \quad (12)$$

The serial ARM method was implemented in MatLab using mex-files and it is available in

Ref. 19.

---

**Algorithm 1** Multigrid Unwrapping. Multigrid strategy with  $N$  scale levels.

---

```

1: function MULTIGRIDUNWRAP( $g, f, \lambda, \mu, N, T$ )
2:   if  $N > 0$  then
3:      $\hat{g} = \text{DOWNSAMPLE}(g)$ ; ▷ Down sampling
4:      $\hat{f} = \text{DOWNSAMPLE}(f)$ ;
5:      $\hat{f} = \text{MULTIGRIDUNWRAP}(\hat{g}, \hat{f}, \lambda, \mu, N - 1, T)$  ▷ Change level
6:      $f = \text{UPSAMPLE}(\hat{f})$ ; ▷ Up sampling
7:   end if
8:   //Unwrap current level
9:   Set  $\delta = 0, \forall r$ ;
10:  Set  $\omega_{rs} = 1, \forall (r, s)$ ; ▷ A different initial value for  $\omega$  can be used
11:  Compute  $\tilde{\rho}_{rs} = W(g_r - g_s) - (f_r - f_s), \forall (r, s)$ ;
12:  for  $t = 1, 2, \dots, T$  do
13:    Update  $\delta$ , keeping fixed  $\omega$ , with (9);
14:    Update  $\omega$ , keeping fixed  $\delta$ , with (10);
15:  end for
16:   $f = f + \delta$ ;
17: end function
18: return  $f$ ;

```

---

### 3 Parallel implementation

In our parallel implementation we develop two frameworks (see Table 1). Framework 1 uses MatLab and improves the processing time of equations (9) and (10). We include

---

**Algorithm 2** Nested Multigrid Unwrapping. Nested multigrid strategy for Algorithm 1 with  $N$  iterations of  $N$  scale levels.

---

```
1: function NESTMULTIGRIDUNWRAP( $g, \lambda, \mu, N, T$ )
2:    $f = 0; N_0 = N$ 
3:   while  $N > 0$  do
4:      $f' = \text{MULTIGRIDUNWRAP}(g, \lambda, \mu, N_0, T);$            ▷ Simple multigrid
5:      $f = f + f'$ ;
6:      $g = \text{residual}(g, f);$                                    ▷ Residual wrapped phase
7:      $N = N - 1;$ 
8:   end while
9:   return  $f + g;$ 
10: end function
```

---

OpenMP directives into the mex-C file to execute the heavy work in the multicore CPU or in the XPC. On the other hand, we use the CUDA kernel integration in MatLab (without mex-C) to process in the GPU. Framework 2 uses C/C++ and improves the processing time of the whole method. Then, the algorithms 1 and 2 are translated from MatLab scripts to C/C++ files, this framework uses only OpenCV for reading and writing input images. With the whole code in C/C++ we can include OpenMP directives to execute the program in the multicore CPU or in the XPC, and we can create CUDA kernel functions to execute the program in a GPU. An implementation of the proposed frameworks can be found in the following code ocean capsules: <https://codeocean.com/2018/04/09/comparison-of-multi-hardware-parallel-implementations-for-a-phase-unwrapping-algorithm/> and <https://codeocean.com/2018/04/23/comparison-of-multi-hardware-parallel-implementations-for-a-phase-unwrapping-algorithm/>.

From the serial ARM method, we note that the solver for equations (9) and (10) is the computationally heaviest part, then we parallelize these equations as a first approach. Note that, for updating  $\omega$  in equation (10) as well as the sentences in the algorithms 1 and 2, we need only pixel-wise operations, applying an independent parallelism model,<sup>20</sup> whereas for updating  $\delta$  in equation (9), we need a strategy to parallelize the Gauss-Seidel method, because the communication between a pixel and its neighborhoods. The following subsections describe the Gauss-Seidel method and the implementation details, focusing on the parallelization of equation (9).

### 3.1 Gauss-Seidel method

The Gauss-Seidel (GS) method is one of the basic iterative methods for solving linear systems.<sup>21</sup> In our particular case we want to compute equation (9) with a GS iterative scheme. An advantage of GS is that it is only necessary one array to allocate and update the values of  $\delta_r$ . On the other hand, depending on the order in which we loop the grid pixels, we will get different implementations of the GS method.

We implement our algorithm taking into account two ordering ways: the natural ordering (see Algorithm 3) and the red-black ordering (see Algorithm 4). Note that in both algorithms the for loops run over all image pixels  $r$  per row. The difference is that in the red-black ordering the pixels are considered red and 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. Then, when the

red pixels are updated in the first for loop, they only need the black pixel values and vice versa in the second loop. With the red-black ordering we can implement the algorithm in parallel using a multicore CPU, XPC or GPU.

---

**Algorithm 3** GS with natural ordering. Computes  $\delta$ ;  $T$  is the maximum iteration number;  $cols$  and  $rows$  are the height and width of the image respectively;

---

```

1: function GS( $\delta, \omega, \tilde{\rho}, \lambda, T$ )
2:   for  $t = 1, 2, \dots, T$  do
3:     for  $r = (1, 1)$  to  $(cols, rows)$  do
4:       Update  $\delta_r$  with equation (9);
5:     end for
6:   end for
7:   return  $\delta$ ;
8: end function

```

---



---

**Algorithm 4** GS with red-black ordering. Computes  $\delta$ ;  $T$  is the maximum iteration number;  $cols$  and  $rows$  are the height and width of the image respectively;

---

```

1: function GS RED-BLACK( $\delta, \omega, \tilde{\rho}, \lambda, T$ )
2:   for  $t = 1, 2, \dots, T$  do
3:     for  $r = (1, 1)$  to  $(cols, rows)$  that are red do
4:       Update  $\delta_r$  with equation (9);
5:     end for
6:     for  $r = (1, 1)$  to  $(cols, rows)$  that are black do
7:       Update  $\delta_r$  with equation (9);
8:     end for
9:   end for
10:  return  $\delta$ ;
11: end function

```

---

### 3.2 Implementation details

In both frameworks, we use the OpenMP and CUDA languages to parallelize the code.

In the following, we describe the directives and functions created in order to parallelize

equation (9).

OpenMP is an API for shared-memory parallel programming.<sup>22</sup> The “M” in OpenMP stands for “multiprocessing,” a term that is synonymous with shared-memory parallel computing. Thus, OpenMP is designed for systems in which each thread or process can potentially have access to all available memory. OpenMP provides a set of pragma directives, which are used to specify parallel regions, to manage threads inside parallel regions, to distribute for loops in parallel, among other things.

Note that, the internal for loops of Algorithm 4 can be parallelized with OpenMP directives as follows:

```
omp_set_num_threads(NUM.THREADS);  
#pragma omp parallel  
{  
    #pragma omp for private()  
    for (:, :, :) {  
    }  
}
```

where *NUM\_THREADS* is the thread number to launch. *#pragma omp parallel* opens a parallel region and *#pragma omp for* parallelizes the for loop. In shared-memory programs, the individual threads can have private and shared memory. Com-

munication is usually done through shared variables. Inside the *#pragma omp for*, all variables are shared by default for all threads, then, to declare shared and private variables we use only the *private* directive, in order to declare the variables which are private for each thread launched. In this way, we can process a mex-C or a C/C++ code in a multicore CPU.

Intel XPC, also known as Intel many integrated core architecture (or Intel MIC) is a coprocessor computer architecture which offers additional power-efficient scaling, vector support, and local memory bandwidth, while maintaining the programmability and support associated with Intel Xeon processors.<sup>23</sup> The XPC runs Linux and has its own IP address. Then we can log onto the XPC in a terminal window. There are two programming models for computing in an XPC, the native and the offload model. In the native model, the application runs in the coprocessor, while in the offload model, the application runs a main host program in the CPU and offloads work to the coprocessor. We use the offload model, because our application can run the main program from MatLab or OpenCV and offloads the heavy work in the coprocessor through mex-C file or the C/C++ code. Moreover, the XPC provides a wide interoperability with OpenMP, bringing a set of specialized directives; thus we add the following at Algorithm 4:

```
#pragma offload target(mic:0) inout() in()
{
```

```

omp_set_num_threads(NUM_THREADS);
for (t=0;t<T;t++){
    #pragma omp parallel
    {
        #pragma omp for private()
        for(, , :){//for red pixels
        }
        #pragma omp for private()
        for(, , :){//for black pixels
        }
    }
}
}

```

where *#pragma ofload target(mic : 0)* opens a code region which will run in the XPC. Note that we use the attribute *inout* for including variables that will be transferred from CPU to XPC memory and vice versa. Also we use the attribute *in* for including variables that will only be transferred from CPU to XPC. The rest are pragmas of OpenMP to parallelize the internal for loops. These pragma directives can be ignored and the program should simply work in a non parallel mode (sequential). When a compiler recognizes OpenMP directives (requires the `-openmp` switch on the Intel compilers), then the directives are interpreted to give direction on how to create parallel tasks in order to speed the

execution of a program through parallelism.

On the other hand, CUDA is a language that contains a set of instructions for processing in a GPU. The advantage of using a GPU is that 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 one simple instruction in parallel, similar to single instruction multiple data (SIMD) model.<sup>24,25</sup> In framework 1, we use the CUDA kernel integration in MatLab.<sup>26</sup> Then, we create an executable kernel from CU or PTX (parallel thread execution) files, and run that kernel on a GPU from MATLAB. The kernel is represented in MATLAB by a `CUDAKernel` object, which can operate on MATLAB array or `gpuArray` variables. We implement the kernel of Algorithm 4 as follows:

```
%Copy memory from CPU to GPU  
  
delta_dev=gpuArray(double(delta));  
  
omega_dev=gpuArray(double(omega));  
  
%Create the kernel functions  
  
GS_kernel=parallel.gpu.CUDAKernel('solverGS.ptx','solverGS.cu');  
  
Omega_kernel=parallel.gpu.CUDAKernel('Omega.ptx','Omega.cu');  
  
%Size of Grid and Thread Blocks  
  
blocksize_x=32; %Fix this parameter according  
  
blocksize_y=32; %to GPU capabilities
```



```

grid_x=ceil(rows/blocksize_x);
grid_y=ceil(cols/blocksize_y);
solverGS_kernel.ThreadBlockSize=[blocksize_x,blocksize_y,1];
solverGS_kernel.GridSize=[grid_x,grid_y];
Omega_kernel.ThreadBlockSize=[blocksize_x,blocksize_y,1];
Omega_kernel.GridSize=[grid_x,grid_y];
%kernel executions
for t=1:niter
    [delta_dev]=feval(GS_kernel,delta_dev,omega_dev,...,0);%Red-->0
    [delta_dev]=feval(GS_kernel,delta_dev,omega_dev,...,1);%Black-->1
    [omega_dev]=feval(Omega_kernel,omega_dev,delta_dev,...);
end
%Copy memory from GPU to CPU
delta=gather(delta_dev);

```

Note that in framework 1, using any device (an XPC or a GPU), the number of times that the program needs to transfer memory between CPU and the device is  $N \times N$ , where  $N$  is the number of levels for both the multigrid Algorithm 1 and the nested Algorithm 2. In order to reduce this number of transferences, we develop framework 2, where we translate the MatLab script to C/C++ code. In this framework, we can transfer the memory from Algorithm 2 just once.

With the implementations above described, we can run our program in a multicore

CPU, XPC or GPU. In the following section, we present some experiments and results comparing the execution time in such architectures.

#### 4 Synthetic experimental evaluation

The experiments were executed on two servers, the first one (server K20) is a server with Intel(R) Xeon(R) CPU E5-2620 v2 2.10 GHz, Ubuntu 14.04 (64-bits), 24 hyper-threading cores, a Xeon Phi Coprocessor 3120A 1.1GHz with 57 cores and 6GB RAM, and a video card Tesla K20 with 4GB RAM. The second one (server K40) is a server with Intel(R) Xeon(R) CPU E5-2690 v2 3.00 GHz, Ubuntu 14.04 (64-bits), 20 physical cores, a Xeon Phi Coprocessor 3120A 1.1GHz with 57 cores and 6GB RAM, and a video card Tesla K40 with 12GB RAM.

Fig. 1 shows a synthetic wrapped phase map with a size of  $512 \times 512$  pixels, which we use in our experiments. This synthetic wrapped phase map was used in Ref. 8 to compare the ARM method with different state of the art methods. The second and third row of the Fig. 1 show the re-wrapped phase from the unwrapped phase of the ARM method, in serial and parallel versions respectively, using different numbers of iterations  $T$ ,  $N = 5$  levels of nested and multigrid algorithms,  $\lambda = 0.1$  and  $\mu = \pi/10$ . With  $T = 2000$ , we note that the re-wrapped phase in the serial and parallel versions converge to the same result; thus, in the rest of the synthetic experiments we fix  $T = 2000$  iterations.

Tables 2 and 3 show a comparative performance of our two respective frameworks of

parallel ARM method for server K20, on the other hand, the Table 4 shows a comparative performance of the parallel ARM method framework 2 for server K40. These tables show processing times using the double-precision floating-point format in different architectures (multicore CPU, XPC and GPU) and different image sizes ( $512 \times 512$ ,  $1024 \times 1024$ ,  $2048 \times 2048$  and  $4096 \times 4096$ ). Processing time is measured in seconds from the start of Algorithm 2 until this is finalized, which means that we are considering both the processor work and the memory transfer time.

With a size of image  $\leq 512^2$  the best processing time is obtained using the multicore CPU of server K20 (see Tables 2 and 3), while with a size of  $\geq 1024^2$  the best time is obtained using the GPU. Server K20 has 24 hyper-threading cores, but physically it has two sockets with 6 cores per socket, what means the server has 12 physical cores and 24 logical cores. We can see that the best performance is obtained with 12 and 8 cores. On the other hand, the server K40 has 20 cores without hyper-threading. Note that the best times for the multicore CPU were obtained when we fixed the *NUM\_THREADS* to  $20C$  (see Table 4). When the hyper-threading is enabled, each physical core is divided in two logical cores, sharing resources such as the instruction pointers, integer and floating point registers, scheduling queues, caches and execution units. The performance of a parallel program using hyper-threading declines if a logical core monopolizes some critical resource as the floating point registers or the caches. The purpose of parallel programming

is increased performance, which is fundamentally a program optimization problem. Since the memory hierarchies are radically different between platforms and the connection of cores on a single processor vary widely, this optimization problem is complicated.<sup>27</sup> In our implementation, we can see that it is better to adjust *NUM\_THREADS* with the number of physical cores ( $N_{pc}$ ) of CPU than with the number of logical cores, in order to obtain good performance.

We observe that the results of the XPC in server K20 are better than the serial program and comparable with the *2C* multicore option, but these are overcome by the other architectures. One might think that the bad performance of the XPC in Table 2 is due to the memory transfers as the case of the GPU, but in Table 3 we can see that even with the reduction of memory transfers, the XPC has the worst times. However, in Table 4, we can see an improvement in the processing time of the XPC, this is due to the directive *#pragma omp simd* that we have included in the OpenMP code. This directive transforms a loop into a loop that will be executed concurrently using SIMD instructions to help with the vectorization inside the XPC.<sup>23</sup> Vectorization refers to apply a single instruction to a group of data items of the same data type (or vector) that can be processed at once. On the XPC the vector processing unit supports 512-bit vector width, our implementation uses double-precision floating-point format numbers, thus 8 values can be processed simultaneously.

With respect to the GPU, we can see that the memory transfers impact its performance (see Tables 2 and 3). Note that between the GPUs K20 and K40 (see Tables 3 and 4) there is a little difference in the processing time, which is due to the differences in the clock speed of both servers and the CUDA cores, 2688 of K20 and 2880 of K40.

**Table 2** Processing time in seconds of our parallel ARM method framework 1 in the server K20. The smallest time per row is shown in bold.

Size of Image	Serial 1C	Multicore CPU						XPC		GPU
		2C	4C	8C	12C	16C	20C	112C	224C	
512 <sup>2</sup>	36.5	19.3	11.2	6.8	<b>5.7</b>	10.1	7.1	17.9	24.6	10.8
1024 <sup>2</sup>	144.6	80.3	43.3	30.8	26.7	26.9	28.6	50.5	88.8	<b>21.9</b>
2048 <sup>2</sup>	814.3	449.6	249.1	154.6	155.8	158.4	150.1	316.2	360.7	<b>66.9</b>
4096 <sup>2</sup>	3529.9	1937.8	1067.9	616.3	604.6	779.3	700.3	1206.9	1291.1	<b>248.4</b>

**Table 3** Processing time in seconds of our parallel ARM method framework 2 in the server K20. The smallest time per row is shown in bold.

Size of Image	Serial 1C	Multicore CPU						XPC		GPU
		2C	4C	8C	12C	16C	20C	112C	224C	
512 <sup>2</sup>	33.9	20.4	10.6	5.7	5.5	7.2	<b>5.3</b>	30.1	32.0	6.04
1024 <sup>2</sup>	136.0	81.8	49.3	27.4	22.9	23.5	19.9	60.3	60.6	<b>11.2</b>
2048 <sup>2</sup>	660.8	360.6	282.0	97.8	122.5	121.4	127.1	449.8	410.6	<b>34.2</b>
4096 <sup>2</sup>	2815.1	1452.6	775.0	411.6	603.7	514.6	499.3	3210.5	3192.5	<b>123.3</b>

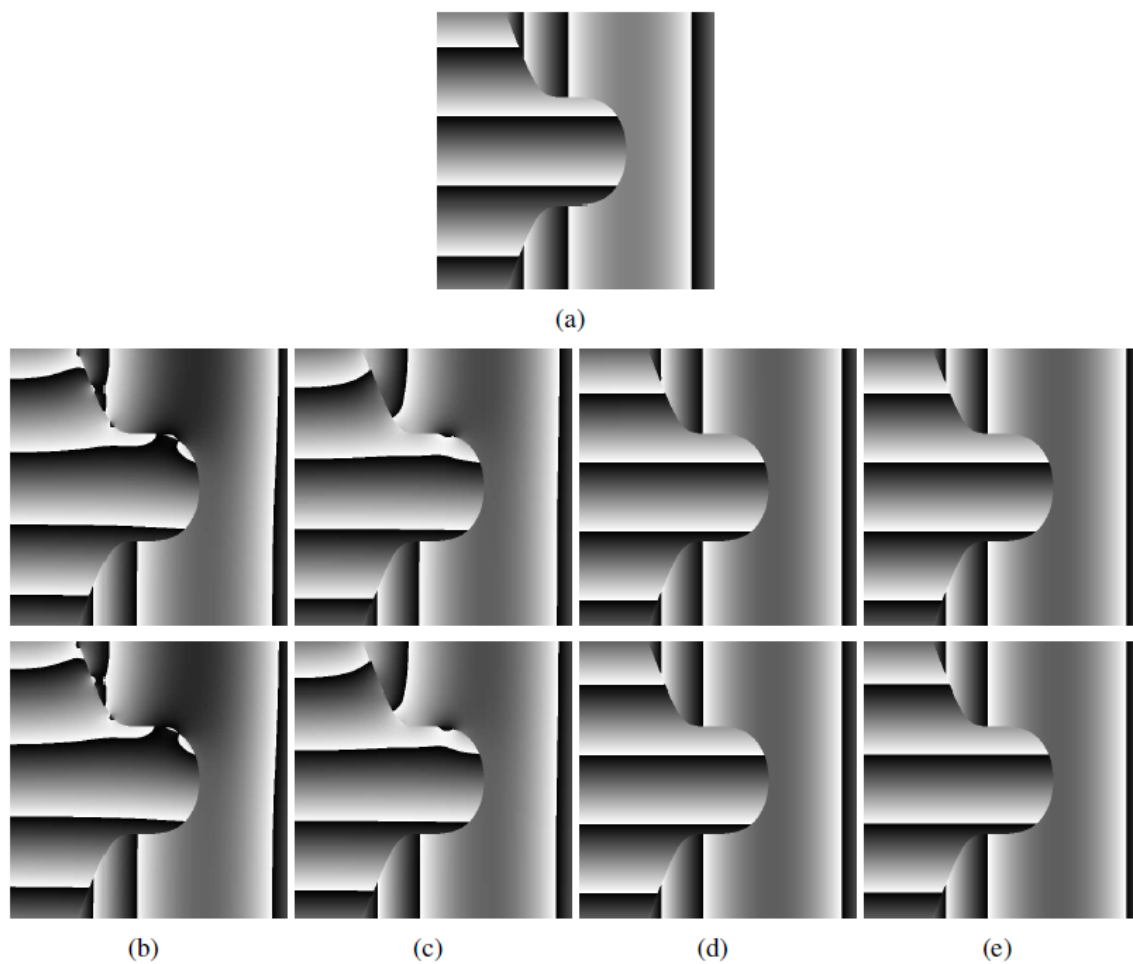
The speedup of a parallel program can be defined as

$$S = \frac{T_s}{T_p}, \quad (13)$$

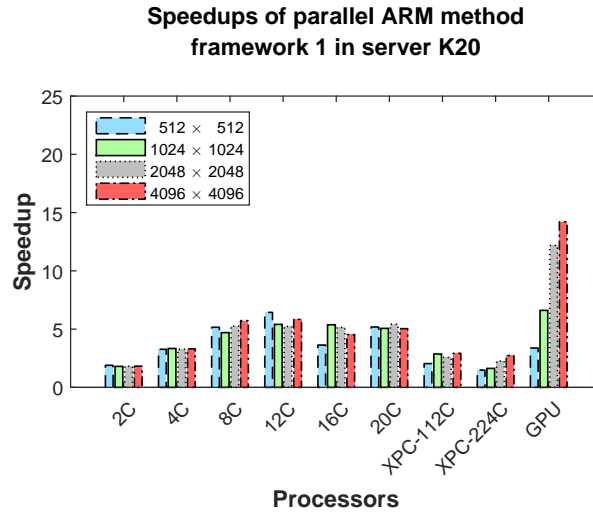
**Table 4** Processing time in seconds of our parallel ARM method framework 2 in the server K40. The smallest time per row is shown in bold.

Size of Image	Serial 1C	Multicore CPU						XPC		GPU
		2C	4C	8C	20C	24C	28C	112C	224C	
512 <sup>2</sup>	26.3	15.3	7.9	4.1	<b>2.2</b>	3.4	3.5	9.5	10.6	2.5
1024 <sup>2</sup>	107.2	62.4	31.5	16.2	<b>7.1</b>	12.7	11.5	24.8	23.0	7.2
2048 <sup>2</sup>	515.7	281.9	146.7	67.4	50.5	72.4	69.4	130.7	100.6	<b>25.7</b>
4096 <sup>2</sup>	2193.8	1047.1	536.6	273.2	208.8	285.4	273.0	716.9	636.2	<b>99.6</b>

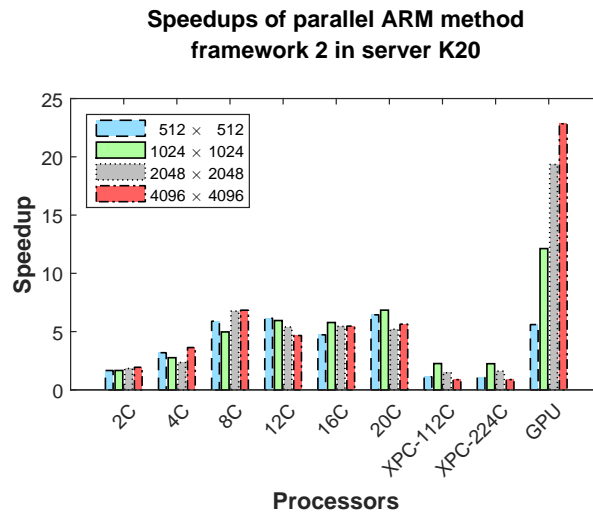
where  $T_s$  is the processing time of a serial program and  $T_p$  is the processing time of a parallel program.<sup>22</sup> Figures 2 and 3 show the speedups of the two frameworks of parallel ARM method in server K20, and Fig. 4 shows the speedups of framework 2 in server K40. In the case of multicore CPU, figures 2 and 3 show that  $S$  increases when we use from 2C to 8C or 12C, while this decreases when we use 16C and 20C. In Fig. 4, we can see an increase in  $S$  until  $N_{pc}$ , then with  $NUM\_THREADS > N_{pc}$ ,  $S$  decreases and it is comparable with the speedup of 8C. Next, in the case of the XPC, we can see that  $S$  is almost the same that 2C multicore in figures 2 and 3, while in Fig. 4 the speedup of XPC is over the speedup of 4C multicore, this due to the included *simd* directive. Note that in the case of the GPU,  $S$  has notable differences between the different sizes of processed images. Furthermore,  $S$  is higher in framework 2 than framework 1, reaching a speedup of approx. 23x for an image of  $4096 \times 4096$  pixels.



**Fig 1** Re-wrapped phases from serial (second row) and parallel (third row) versions at different numbers of iterations  $T$ . (a) Wrapped synthetic phase. (b)  $T = 250$ . (c)  $T = 500$ . (d)  $T = 1000$ . (e)  $T = 2000$ .

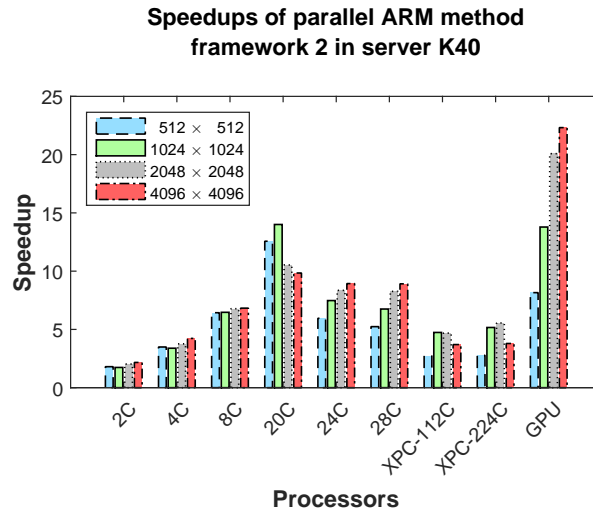


**Fig 2** Evaluation of speedups of parallel ARM method framework 1 in server K20, using different architectures (multicore CPU from 2C to 20C, XPC with 112C and 224C, and GPU K20) and different image sizes.



**Fig 3** Evaluation of speedups of parallel ARM method framework 2 in server K20, using different architectures (multicore CPU from 2C to 20C, XPC with 112C and 224C, and GPU K20) and different image sizes.

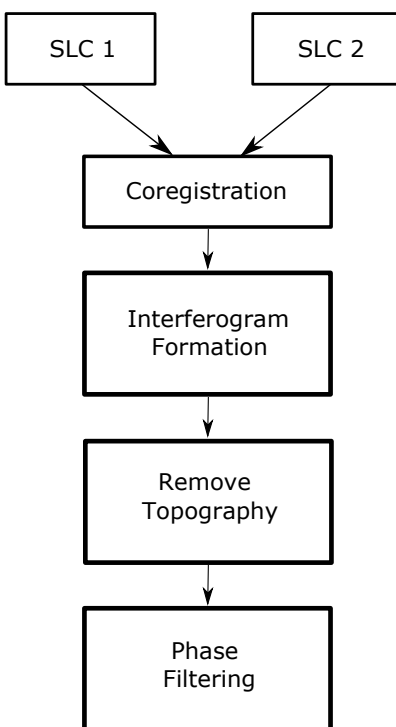




**Fig 4** Evaluation of speedups of parallel ARM method framework 2 in server K40, using different architectures (multicore CPU from 2C to 28C, XPC with 112C and 224C, and GPU K40) and different image sizes.

## 5 Phase unwrapping for large SAR interferograms

Before introducing the conducted experiments with large SAR inteferograms, we show the generation of interferometric synthetic aperture radar (InSAR) for RADARSAT-2, which is a Canadian satellite system equipped with a powerful synthetic aperture radar (SAR) instrument. It offers powerful technical advancements that enhances marine surveillance, environmental monitoring, resource management, mapping around the world, etc. From this process we obtain the wrapped phase that will be later unwrapped with the ARM method.



**Fig 5** Interferometric synthetic aperture radar (InSAR) formation as applied to RADARSAT-2 single look complex products.

### 5.1 InSAR formation

InSAR exploits the phase difference between two single look complex (SLC) images acquired over the same area from slightly different sensor positions.<sup>7</sup> Fig. 5 shows the processing chain for InSAR formation. The block diagram can be applied for SLC images using the opensource software called sentinel application platform (SNAP)<sup>28</sup> as follows:

- *Data coregistration.* In order to create the interferogram, the two SLC images must be coregistered into a stack. One image is selected as the master or reference and the other image is the slave. The pixels in the slave image will be moved to align

with the reference image to sub-pixel accuracy. This ensures that each ground target contributes to the same pixel in both the master and slave images.

- *Interferogram formation.* Once the images have been coregistered, the interferogram is formed by cross multiplying the master image  $u_m = |u_m|e^{i\phi_m}$  with the complex conjugate of the slave image  $u_s^* = |u_s|e^{-i\phi_s}$ . The amplitude of both images is computed by  $|u_m||u_s|$  while the phase represents the phase difference between the two images  $\phi = \phi_m - \phi_s$ .
- *Topographic removal.* The phase difference can have contributions from the Earth phase, which is the phase contribution due to the Earth curvature. In the interferometric processing, the flat Earth phase is removed in order to eliminate sources of error, to be left with only the contributor of interest, which is typically the elevation or the displacement. Therefore, the interferogram is flattened after removing the topographic phase.
- *Phase filtering.* Interferometric phase is normally corrupted with noise, therefore, to properly unwrap the phase, the signal to noise ratio (SNR) needs to be increased by filtering it.

## 5.2 Unwrapping SAR interferograms

For unwrapping SAR interferograms, we apply our ARM method framework 2 in server K40, with  $\lambda = 10$  and  $\mu = 100$ . Fig. 6, panel (a) shows a wrapped SAR phase taken from a tutorial available in Ref. 29, which has a size of  $561 \times 1591$  pixels. We apply our method with  $N = 3$  and  $T = 4000$ . Panels (b) and (c) show the unwrapping and re-wrapped phase, and panel (d) shows a mesh of the unwrapped phase, where we can see the shape of a volcano. The processing time is approximately 105.2 seconds using the serial code, 17.7 seconds using the XPC (a speedup of approx. 6x), 4.5 seconds using OpenMP with 20C (a speedup of approx. 23x), and 7.5 seconds using the GPU (a speedup of approx. 14x).

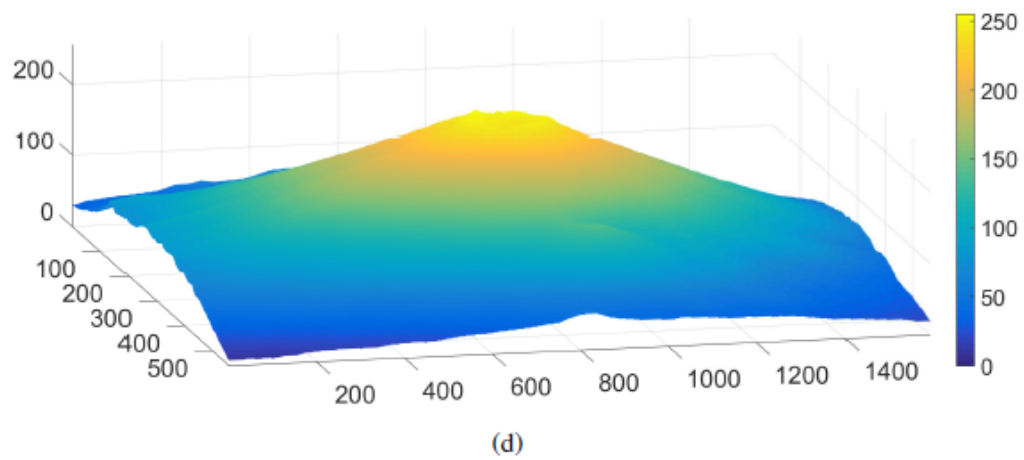
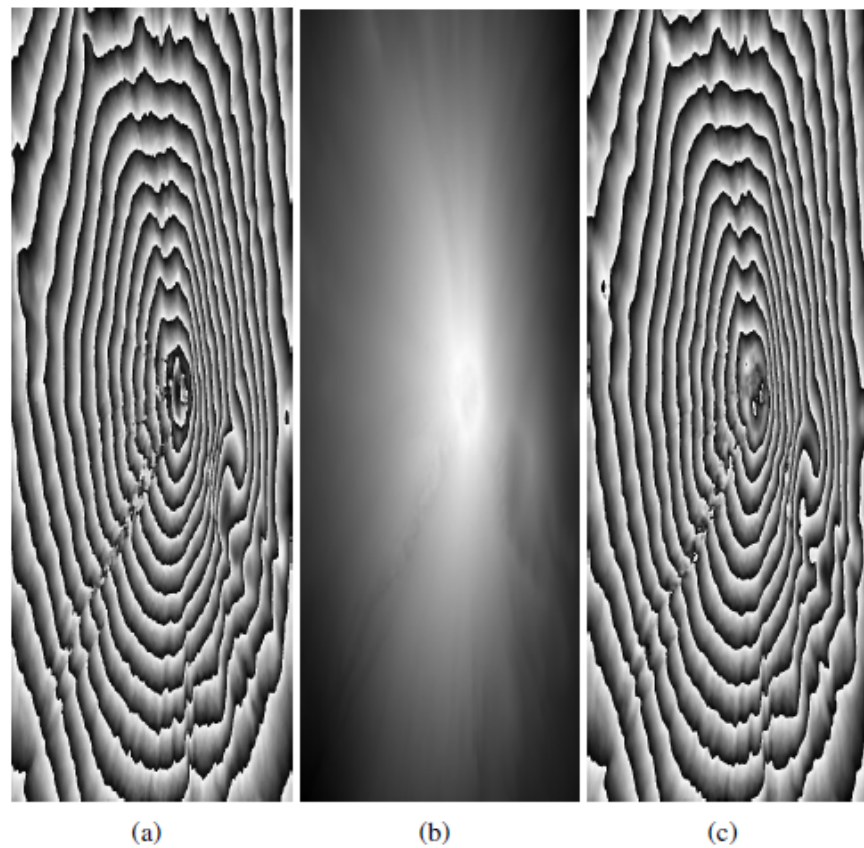
On the other hand, the panel (a) of Fig. 7 shows a huge wrapped SAR phase with size of  $8000 \times 10500$  pixels, that we create from two SLC RADARSAT-2 images using SNAP and following the steps above described for InSAR formation. The images were acquired on May 4th 2008 and May 28th 2008 over the Phoenix area. Here we are interested in full-resolution SAR interferograms, but a subset around a particular area in which you are interested can be created to reduce the amount of processing needed. For unwrapping the wrapped SAR phase, we fix  $N = 7$  and  $T = 1000$ . Panels (b) and (c) show the unwrapped and re-wrapped phase, and panels (d) and (e) show the unwrapped phase and mesh of the region bounded by the red rectangle of size of  $801 \times 2001$  pixels. The processing time is approximately 6216.8 seconds using the serial code, 739.7 seconds using OpenMP with

20C (a speedup of approx. 8x), and 397.4 seconds using the GPU (a speedup of approx. 16x). In this experiment, the memory of the XPC was insufficient.

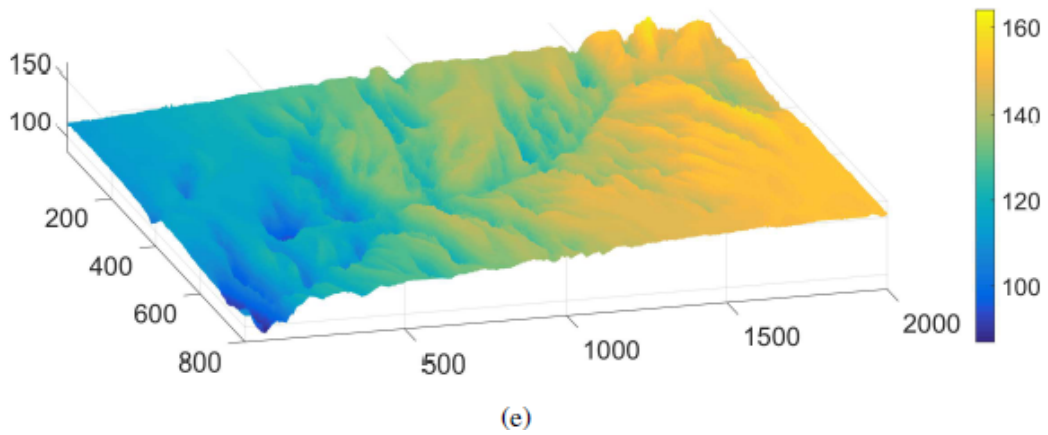
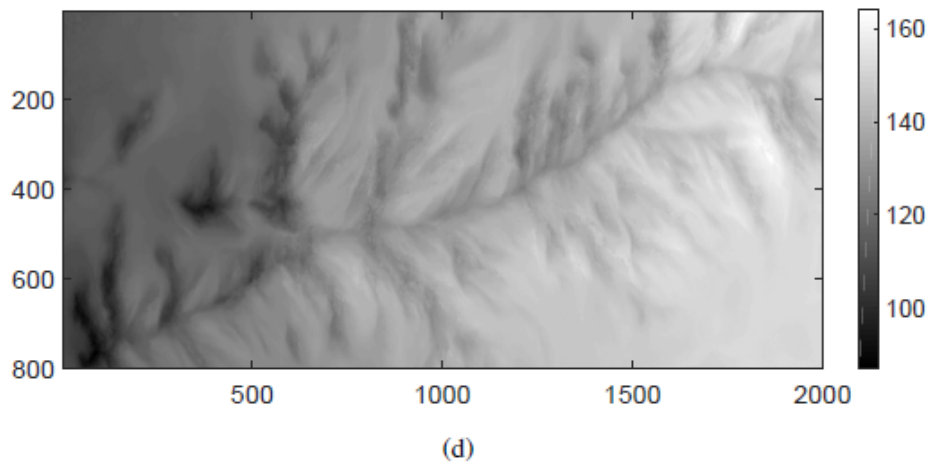
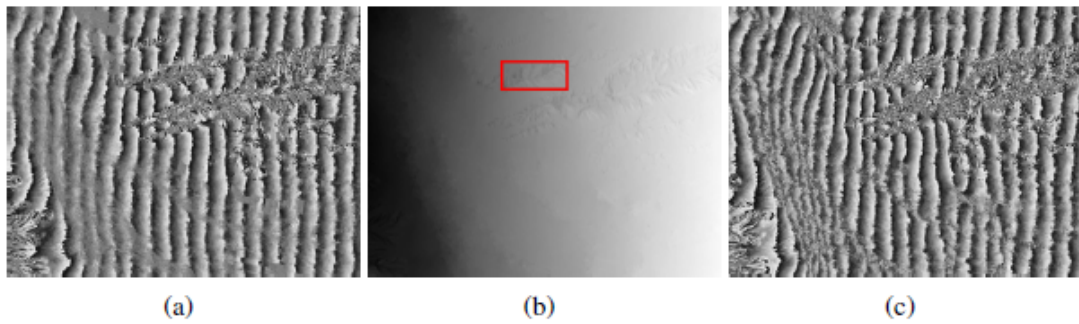
## **6 Remarks and conclusions**

In this work, different parallel architectures were used, comparing its processing time for solving a phase unwrapping problem. We translate the whole code from MatLab to C/C++ in order to parallelize the method and improve the speedup, avoiding several copies of memory between CPU-RAM and the coprocessors XPC or GPU RAM. For our parallel implementation using a multicore CPU, it is important to know the number of physical cores, because with this number, we obtain the highest speedup between the possible configurations for launching threads in the CPU. In the case of XPC, it is notable that its speedup can be improved if we consider the vectorization, however it is very poor compared with the rest. Although it is easier to translate a serial program to parallel using OpenMP than CUDA, we can see that the GPU remains a powerful tool, with which is possible to obtain high speedup values above 15x, considering image sizes greater than  $1024^2$  pixels such as SAR interferograms, and thus it is worthwhile to invest time in to translate the code using CUDA.

As future work, we think that another interesting comparison could be using OpenCL, in order to create a kernel function, which can be offloaded in the different architectures here used. We want to improve the performance of our GPU implementation using texture



**Fig 6** Unwrapped phase from SAR interferogram taken from Ref. 29. (a) Wrapped phase (b) Unwrapped phase. (c) Re-wrapped phase. (d) Mesh of unwrapped phase.



**Fig 7** Unwrapped phase from our SAR interferogram taken over Phoenix area. (a) SAR wrapped phase. (b) Unwrapped phase. (c) Re-wrapped phase. (d) A region (red rectangle) of the unwrapped phase. (e) Mesh of unwrapped phase region (red rectangle).

and unified memory. Moreover, we want to compare the ARM method with another one (such as Snaphu<sup>30</sup>) that is used to process SAR interferograms.

### *References*

- 1 A. Moreira, P. Prats-Iraola, M. Younis, *et al.*, “A tutorial on synthetic aperture radar,” *IEEE Geoscience and remote sensing magazine* **1**(1), 6–43 (2013).
- 2 D. Massonnet and K. L. Feigl, “Radar interferometry and its application to changes in the earth’s surface,” *Reviews of Geophysics* **36**(4), 441–500 (1998).
- 3 R. Bürgmann, P. A. Rosen, and E. J. Fielding, “Synthetic aperture radar interferometry to measure earths surface topography and its deformation,” *Annual review of earth and planetary sciences* **28**(1), 169–209 (2000).
- 4 European Space Agency (ESA), “Sentinel Online.” Website (2017). <https://sentinel.esa.int/web/sentinel/home/>.
- 5 R. Bamler and P. Hartl, “Synthetic aperture radar interferometry,” *Inverse problems* **14**(4), R1 (1998).
- 6 G. Franceschetti, S. Merolla, and M. Tesauro, “Phase quantized SAR signal processing: theory and experiments,” *IEEE Transactions on Aerospace and Electronic Systems* **35**(1), 201–214 (1999).
- 7 M. Simons and P. Rosen, “Interferometric synthetic aperture radar geodesy,” *Elsevier:Treatise on Geophysics (Second Edition)* **3**(2), 339–385 (2015).



- 8 M. Rivera, F. J. Hernandez-Lopez, and A. Gonzalez, "Phase unwrapping by accumulation of residual maps," *Optics and Lasers in Engineering* **64**, 51–58 (2015).
- 9 K. H. Tsoi and W. Luk, "Axel: A heterogeneous cluster with fpgas and gpus," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '10*, 115–124, ACM, (New York, NY, USA) (2010).
- 10 M. A. Gdeisat, D. R. Burton, F. Lilley, *et al.*, "Aiding phase unwrapping by increasing the number of residues in two-dimensional wrapped-phase distributions," *Applied Optics* **54**(34), 10073 (2015).
- 11 J. C. de Souza, M. E. Oliveira, and P. A. M. dos Santos, "Branch-cut algorithm for optical phase unwrapping," *Optics Letters* **40**, 3456 (2015).
- 12 M. Arevalillo-Herraez, F. R. Villatoro, and M. A. Gdeisat, "A Robust and Simple Measure for Quality-Guided 2D Phase Unwrapping Algorithms," *IEEE Transactions on Image Processing* **25**, 2601–2609 (2016).
- 13 D. C. Ghiglia and M. D. Pritt, *Two-Dimensional Phase Unwrapping: Theory, Algorithms, and Software*, Wiley-Interscience, New York (1998).
- 14 B. R. Hunt, "Matrix formulation of the reconstruction of phase values from phase differences," *J. Opt. Soc. Am.* **69**, 393–399 (1979).
- 15 D. C. Ghiglia and L. A. Romero, "Robust two-dimensional weighted and unweighted

- phase unwrapping that uses fast transforms and iterative methods,” *Journal of the Optical Society of America A* **11**, 107 (1994).
- 16 J. L. Marroquin and M. Rivera, “Quadratic regularization functionals for phase unwrapping,” *Journal of the Optical Society of America A* **12**, 2393–2400 (1995).
- 17 M. Rivera, J. L. Marroquin, M. Servin, *et al.*, “Fast algorithm for integrating inconsistent gradient field,” *Applied Optics* **36**(32), 8381–8390 (1997).
- 18 W. L. Briggs, V. E. Henson, and S. F. McCormick, *A Multigrid Tutorial (2Nd Ed.)*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2000).
- 19 M. Rivera, “Phase unwrapping by accumulation of residual maps.” MathWorks Website (2015). <http://www.mathworks.com/matlabcentral/fileexchange/48094-phase-unwrapping-by-accumulation-of-residual-maps>.
- 20 H. Sutter and J. Larus, “Software and the concurrency revolution,” *Queue* **3**, 54–62 (2005).
- 21 J. W. Demmel, *Applied Numerical Linear Algebra*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (1997).
- 22 P. Pacheco, *An Introduction to Parallel Programming*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st ed. (2011).

- 23 J. Jeffers and J. Reinders, *Intel Xeon Phi Coprocessor High Performance Programming*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st ed. (2013).
- 24 D. B. Kirk and W. H. Wen-Mei, *Programming Massively Parallel Processors: A Hands-on Approach (Applications of GPU Computing Series)*, Morgan Kaufmann, 1 ed. (2010).
- 25 J. Cheng, M. Grossman, and T. McKercher, *Professional CUDA C Programming*, EBL-Schweitzer, Wiley (2014).
- 26 MathWorks-Inc., “Run CUDA or PTX Code on GPU.” Website (2015). <http://www.mathworks.com/help/distcomp/run-cuda-or-ptx-code-on-gpu.html>.
- 27 W.-m. Hwu, K. Keutzer, and T. G. Mattson, “The concurrency challenge,” *IEEE Design Test of Computers* **25**, 312–320 (2008).
- 28 European Space Agency (ESA), “Sentinel Application Platform (SNAP).” Website (2016-12). <http://step.esa.int/main/toolboxes/snap/>.
- 29 SAR-EDU remote sensing education initiative, “DEM generation with MATLAB.” Website (2017). <https://saredu.dlr.de/unit/dem.matlab>.
- 30 C. Chen and H. Zebker, “Phase unwrapping for large SAR interferograms: statistical segmentation and generalized network models,” *IEEE Transactions on Geoscience and Remote Sensing* **40**, 1709–1719 (2002).

**Francisco J. Hernandez-Lopez** received the M.Sc. and D.Sc. degrees in Computer Science from the Center for Research in Mathematics (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, and the development of efficient algorithms using parallel computing. He is Fellow of the National System of Researchers (SNI) of the Mexican Government.

**Mariano Rivera** received the D.Sc. degree in optics from the Center for Research in Optics (CIO), León, México, in 1997. Since 1997, he has been with the Computer Science Department, CIMAT, México. He is Academic Dean of the National Supercomputing Center, Institute Potosino for Scientific and Technological Research (IPICYT), México. His research interests include computer vision, image processing, numerical optimization, machine learning, and optical metrology. He is Fellow of the National Researcher System (SNI) of the Mexican Government.

**Adan Salazar-Garibay** is a Mexican-French PhD researcher at CONACYT, Mexico. Adan Salazar holds a PhD in computer science and automatic control from INRIA Sophia Antipolis and cole des Mines de Paris (Mines Paris Tech) since 2010. His expertise area is in image processing. Publications regarding the outcomes of the research work were presented in some of the main international conferences in robotics and computer vision. His current research interests include 3D reconstruction and SAR image processing.

**Ricardo Legarda-Sáenz** received his PhD in optics from the Centro de Investigaciones en Optica (México) in 2000. Since 2004, he has been an Associate Professor at Universidad Autónoma de Yucatán. His current interests are image processing applied to fringe pattern analysis, moire and fringe projection techniques, and the development of automatic methods for optical metrology. Since 2016, he is Senior member of SPIE.