GPU-based parallel solution for a phase field model

Juan J. Tapia[1], Rigoberto Alvarado[1], and Fernando A. Villalbazo[1]

[1]Instituto Politécnico Nacional, CITEDI Research Center
Av. del Parque no. 1310, Mesa de Otay. Tijuana, México 22510
{jjtapia,ralvarado,fvillalbazo}@citedi.mx
http://www.citedi.mx

Paper received on 22/09/12, Accepted on 17/10/12.

Abstract. In this work, the implementation of an algorithm for the explicit solution of nonlinear equations based on the finite difference method in a Graphic Processing Unit is presented. The work focus on the solution for the Allen-Cahn equation that describes a phase-field model. Our algorithm implementation represents an easy way to solve in a GPU a nonlinear problem for interface phenomena, taking advantage of the characteristics of the problem.

Keywords: GPGPU, CUDA, Phase-field, FTCS

1 Introduction

Graphic Processing Units (GPU) were originally developed as an acceleration unit for graphic and video operations. However, they have been used in the last years for General Purpose Computing, called GPGPU. The key success of the use of GPUs in general computing applications lies on their relation cost/performance when compared with other processor types, i.e. multicore architectures. The GPUs constitute a relatively cheap tool that offers a peak number of FLOPS that is almost 10 times higher than the one reached by multicore processors [10]. This greater computational capacity is full exploited in many investigations and scientific fields.

The nonlinear problems are of especial interest in the scientific field because most physical systems and phenomena, e.g. chemical reactions, ecology, biomechanics and population growth, are nonlinear in nature. A nonlinear problem is described by a nonlinear equation which can be solved using an implicit method or an explicit one. The implicit method for a nonlinear equation involves the iterative solution of a nonlinear system; the explicit method involves the iterative solution of an equation. Generally, the implicit methods offer better numerical stability than the explicit methods and allow the use of bigger time steps in the numerical solutions. For these reasons, an ever increasing proportion of modern scientific research in the nonlinear field is devoted to the design of implicit solutions for nonlinear equations. Among this research, the develop of parallel algorithms that solve large nonlinear systems is a field of ongoing investigations, in particular, GPU-based algorithms that allow a faster and reliable solution for large nonlinear systems.

In [4], Dechevsky *et al.* developed a GPU-based nonlinear system parallel algorithm based on wavelet analysis. Garcia [7] implemented a GPU-based algorithm based on the biconjugate gradient method. Wei, Feng and Lin [13] made an algorithm for nonlinear systems based on the Newton-Raphson method implemented on a GPU. In [6], Galiano *et al.* derived a GPU-based solution that uses a nonlinear conjugate gradient method.

In this work, we focus on an explicit parallel solution for the Allen-Cahn (AC) equation via a GPU-based algorithm. The AC equation is a nonlinear second order differential equation that describes a phase-field model. Many scientific problems involve multiphase and multicomponent flows, e.g. the impact of a droplet on a solid surface, bubbly and slug flows in a microtube, petroleum engineering, combustion and reaction flows, realistic simulations for computer graphics among other applications. The phase-field models are used to model and simulate multiphase and multicomponent flows.

Based on the characteristics of the AC equation and the Courant-Friedrichs-Lewy stability criterion, an explicit solution is an option to solve it. Furthermore, an explicit solution makes unnecessary the storage of large matrices involved with an implicit solution. Due the non data-dependencies and the number of arithmetic operations involved in an explicit method, a parallel solution is a good approach to solve the problem. These characteristics makes a GPU-based implementation an option to solve the AC equation.

In section 2 an introduction to the theory of the phase field models is presented and the Allen-Cahn equation is described. The CUDA programming model and the GPU general architecture are discussed in section 3. Section 4 presents the finite difference method and a detailed description of our algorithm. The results are presented in section 5 and finally, these results are discussed in section 6, where some ideas for future work are highlighted.

2 Phase field model: the Allen-Cahn equation

Phase field models are used to describe the behavior of multiphase and multicomponent flows. In a phase field model, the interfaces are replaced by extremely thin transition regions. The main idea is to introduce an order parameter ϕ that varies sharply but continuously across the thin interfacial region and has an almost uniform value on the bulk phases [2]. This concept is illustrated in Fig. 1, in which ϕ changes its value from phase 1 to phase 2 in a continuous manner across the interface region. In fact, the interface region is kind of a linear interpolation of ϕ, which is used to characterize the phases and could be density, concentration or mass fraction among others parameters.

Assume a binary fluid made of A and B particles in which diffusion is the transport mechanism. Denote by $\phi = -1$ a phase completely made of A particles and by $\phi = 1$ a phase completely made of B particles. A free energy can be defined for times when the system is not in equilibrium. The system evolution is driven by the minimization of this free energy which is given by the functional [9]

$$E\left[\phi\right] = \int \left[F\left(\phi\right) + \frac{1}{2}\varepsilon \left|\nabla\phi\right|^2\right] d\Omega \tag{1}$$

where Ω is the volume of the system under consideration. The term $F\left(\phi\right)$ is the bulk energy potential defined as a classical double-well potential function with two minima for $\phi = -1$ and $\phi = 1$ corresponding to the two phases of the system (the function shape is shown in Fig. 1). The term $\left|\nabla\phi\right|^2$ is the gradient energy, called capillary term,

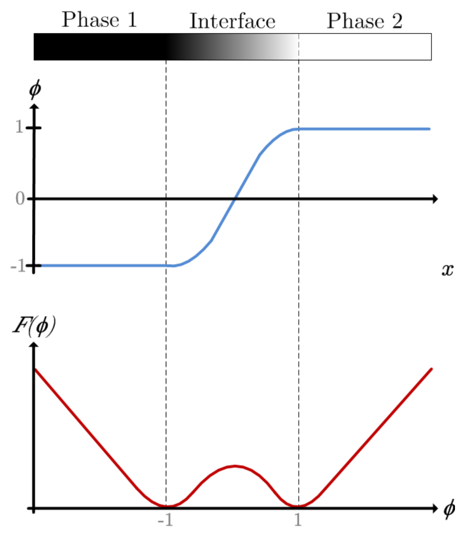

Fig. 1. Concept of phase field model.

which acts as a penalty for sharply varying concentration of ϕ; ε is its coefficient. The bulk energy, also called Helmholtz free energy, describes the entropy of the system. The gradient energy term describes the energy of the interactions between particles of type A and B [2,9].

Variation of E with respect to ϕ is quantifying how the energy changes when particles change position, i.e. the chemical potential of the system [2]. This chemical potential is found by applying variational calculus to (1)

$$\frac{\delta E}{\delta \phi} = \mu = F'(\phi) - \varepsilon \nabla^2 \phi. \tag{2}$$

The evolutionary equation for the system is obtained by making the chemical potential (2) a time-dependant system. The result is the Allen-Cahn equation [1]

$$\frac{\partial \phi}{\partial t} = \varepsilon \nabla^2 \phi - F'(\phi) \tag{3}$$

where $F(\phi) = \frac{1}{4}\left(1 - \phi^2\right)^2$. Replacing $F'(\phi)$ in eq. (3) we get

$$\frac{\partial \phi}{\partial t} = \varepsilon \nabla^2 \phi + \phi - \phi^3. \tag{4}$$

To completely specify the model of equation (4), it is assumed that the boundary and initial conditions are known [8].

The Allen-Cahn equation (4) is used to model the separation process of a binary fluid, in which the fluid is decomposed into a fine-grained mixture of particles. This process is known as spinodal decomposition [2].

3 Graphic Processing Unit

Recently, semiconductor industry has settled on two main design lines for microprocessors: the multicore processors and the manycore processors. The Graphic Processing Units (GPUs) are an example of the former.

The ratio between GPUs and multicore processors for peak floating-point operations is about 10 to 1. The reason for this gap in their performance lies in their architecture [10].

3.1 CUDA programming model

CUDA is an architecture of parallel computing for general purpose that offers high level access to the GPU. CUDA allows the use of a GPU to solve computationally intensive problems in a more efficient way than with a CPU [12].

The CUDA programming model allows the transparent scalability of applications through GPUs with different number of cores [12]. The base to get this transparent scalability are three key abstractions: a hierarchy of threads groups, shared memory and barrier synchronization. These abstractions allow the programmer to partition the problem into thread blocks that can be solved independently. The thread blocks can be considered as sub-problems and are executed in the available GPU cores, in any order and in parallel or sequential manner. This independence of the thread blocks execution is the characteristic that allows the scalability of the CUDA applications [12, 5].

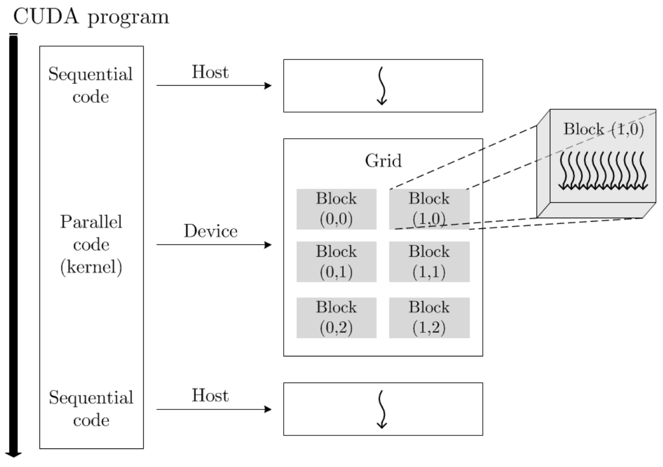

Fig. 2. CUDA heterogenous programming model.

The CUDA programming model let the use of a GPU as a co-processor of the CPU. In this context, GPU is called device and CPU is called host, as it is shown in Fig. 2. A CUDA program is composed of sequential code sections for the host and parallel code sections for the device. In the parallel code sections, thousands of threads are executed concurrently in order to reduce the computation time [10].

In a CUDA program the main thread is executed in the host, as it is shown in Fig. 2. When a kernel is invoked, the execution is moved from the host to the device where a massive number of threads are executed concurrently to perform the parallel operations. A kernel is a subroutine that is executed K times by K threads within the device [10]. The threads generated by a kernel are grouped in thread blocks, and the total of thread blocks within a kernel is called a grid. The kernel calls are asynchronous, which implies that after the invocation of a kernel, the host can execute the rest of the sequential code or just wait to the termination of the kernel execution [10].

3.2 Optimization strategies

The optimization of a CUDA program is based on three main aspects: maximize parallel execution; optimize memory usage to achieve maximum memory bandwidth; and optimize instruction usage to achieve maximum instruction throughput [12, 3].

The most important rule to optimize the memory space usage of the GPU is to minimize the data transfer operations between the CPU and the GPU, because these memory operations have a lower memory bandwidth than the internal transfers within the GPU. It is also important to minimize the kernel access to the global memory and maximize the use of the shared memory [3].

4 GPU implementation of the explicit nonlinear solver

For the explicit solution of eq. (4) we use the Forward-time Central-Space scheme (FTCS), which is a finite difference method used for numerically solve time-dependant partial differential equations (PDE).

4.1 Allen-Cahn equation in 1-D

The scheme for the 1-D FTCS method is show in Fig. 3, where the domain is $[-1, 1]$, N is the number of points of the spatial grid and M is the number of iterations. The black dots represent the initial value for ϕ^0. The red dots are the boundary conditions, which are constant throughout all the iterations. The unknowns that will be calculated are represented as white dots.

In the FTCS scheme, the eq. (4) is discretized in space using the centered finite difference equation for the second order derivative

$$\frac{\partial^2 \phi_i}{\partial x^2} = \frac{\phi_{i-1} - 2\phi_i + \phi_{i+1}}{\Delta x^2} + O(\Delta x^2), \tag{5}$$

and is discretized in time with the forward Euler method

$$\frac{\partial \phi_i}{\partial t} = \frac{\phi_i^{t+1} - \phi_i^t}{\Delta t} + O(\Delta t). \tag{6}$$

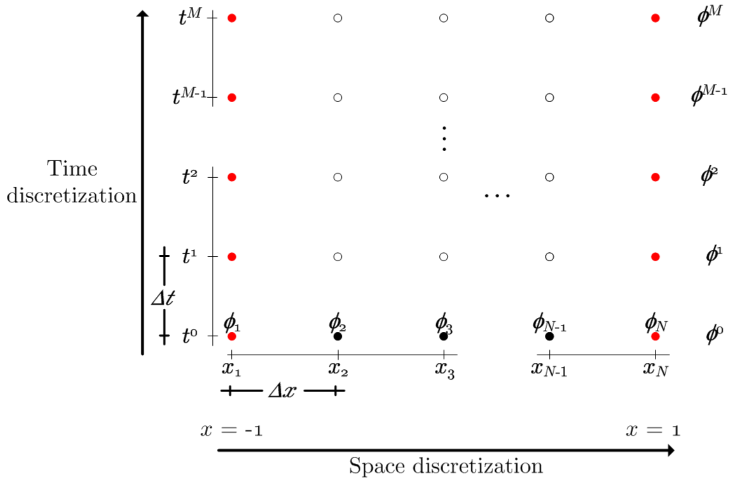

Fig. 3. FTCS scheme.

Therefore, the FTCS scheme is first-order of convergence in time and second order error in space.

Replacing eq. (5) and eq. (6) in eq. (4) we get the explicit discrete version of the AC equation

$$\frac{\phi_i^{t+1} - \phi_i^t}{\Delta t} = \varepsilon \left[\frac{\phi_{i-1}^t - 2\phi_i^t + \phi_{i+1}^t}{\Delta x^2} \right] + \phi_i^t - \left(\phi_i^t\right)^3 . \tag{7}$$

Clearing the term ϕ_i^{t+1} we get

$$\phi_i^{t+1} = r\phi_{i-1}^t + r\phi_{i+1}^t + (-2r + 1 + \Delta t)\,\phi_i^t - \Delta t \left(\phi_i^t\right)^3 , \tag{8}$$

where

$$r = \frac{\varepsilon \Delta t}{\Delta x^2} . \tag{9}$$

The eq. (8) is the iterative explicit equation to solve the AC equation in 1-D. This equation is numerically stable as long as it meets the Courant-Friedrichs-Lewy (CFL) stability criterion for the time step [11]. The CFL is defined as

$$\frac{\varepsilon \Delta t}{\Delta x^2} \leq 1. \tag{10}$$

From eq. (10), we can see the reason that justifies our explicit solution for the AC equation. Since ε weights the gradient energy, i.e. defines the interface thickness (interface area width of Fig. 1), it always has relatively small values, so we can use relatively large time steps (Δt values) that allow get faster to the solution without violating the CFL stability criterion. This advantage plus the non-necessity of storage for large matrices involved with an implicit solution make the explicit solution a good approach to solve the AC equation.

For the 1-D case, we use the Dirichlet boundary conditions

$$\phi(-1,t) = -1, \phi(1,t) = 1. \tag{11}$$

The structure of the GPU implementation of eq. (8) is shown in the algorithm 1.1.

Algorithm 1.1 CUDA FTCS pseudocode for the 1-D Allen-Cahn equation

```
main() {
  allocate memory
  copy CPU to GPU
  $u_{new}[0] = u_{old}[0] = -1$  // Boundary conditions
  $u_{new}[N-1] = u_{old}[N-1] = 1$
  kernelFTCS1D<<< $1, N-2$ >>> $(u_{old}, u_{new})$
  copy GPU to CPU
  free memory
}
__global__ kernelFTCS1D $(u_{old}, u_{new})$ {
  $dx = 2/(N-1)$
  $r_1 = \epsilon dt/dx^2$
  $r_2 = 1 - 2r_1$
  $id = threadIdx.x + 1$
  for $t < T$ do
      $u_{new}[id] = r_1 u_{old}[id-1] + r_2 u_{old}[id] + r_1 u_{old}[id+1] + dt(u_{old}[id] - (u_{old}[id])^3)$
      $\_\_syncthreads()$
      $u_{old}[id] = u_{new}[id]$
  endfor
}
```

As explained in section 3, the first step to use a GPU is copy the necessary data from the CPU memory to the GPU memory. After that, the kernel that implements the eq. (8) is implemented by N threads simultaneously. This means that we only need a loop for the time iterations, because all the threads represents the space discretization. This is the main advantage of the GPU implementation and the root for the better performance.

4.2 Allen-Cahn equation in 2-D

The scheme for the 2-D FTCS method is similar to the 1-D case, except that this time we need to discretize in space through a bidimensional mesh; the time discretization is done in the same way as in the 1-D case. We used an $N \times N$ square shaped domain to solve the eq. (4) for our 2-D experiments, where the interval for both the x and y axis is $[-1, 1]$, N is the number of points and M is the number of iterations. The initial value for all the points of the mesh except the ones at the boundaries are given by the initial condition; the values of the points at the boundaries are given by the boundary conditions through all the iterations.

The 2-D AC equation is discretized in time using the eq. (6). The space discretization is done with eq. (5), but for the 2-D case, we need to calculate $\Delta_2\phi$ with respect to the x and y axes. This means that we need to add another subindex, unlike eq. (7), so we can denote the respective derivatives of a point.

Replacing eq. (6) and eq. (5) in eq. (4) we get the explicit discrete version for the 2-D AC equation

$$\frac{\phi_{i,j}^{t+1} - \phi_{i,j}^{t}}{\Delta t} = \varepsilon \left(\left[\frac{\phi_{i-1,j}^{t} - 2\phi_{i,j}^{t} + \phi_{i+1,j}^{t}}{\Delta x^2} \right] + \left[\frac{\phi_{i,j-1}^{t} - 2\phi_{i,j}^{t} + \phi_{i,j+1}^{t}}{\Delta y^2} \right] \right) + \phi_{i,j}^{t} - \left(\phi_{i,j}^{t}\right)^3 . \tag{12}$$

Clearing the term ϕ_i^{t+1} and assuming $\Delta y^2 = \Delta x^2$ for the mesh we get

$$\phi_{i,j}^{t+1} = r\phi_{i-1,j}^{t} + r\phi_{i+1,j}^{t} + r\phi_{i,j-1}^{t} + r\phi_{i,j+1}^{t} + (-4r + 1 + \Delta t)\,\phi_{i,j}^{t} + \Delta t \left(\phi_{i,j}^{t}\right)^3 , \tag{13}$$

where r is defined in eq. (9), so the same CFL criterion can be applied in this case, meaning that the explicit solution is a good approach to solve the 2-D AC equation.

For the 2-D case, we use Dirichlet boundary conditions at the bottom and top of the square domain, i.e.

$$\phi(x, -1, t) = -1, \phi(x, 1, t) = 1. \tag{14}$$

For the left and right sides of the domain, we use the homogeneous Neumann boundary condition, using centered finite difference with ghost points.

$$\phi_{1,i}^{t+1} = r\phi_{1,i}^{t} + 2r\phi_{2,i}^{t} - \phi_{1,i}^{t}, \tag{15}$$

$$\phi_{N,i}^{t+1} = r\phi_{N,i}^{t} - 2r\phi_{N,i}^{t} - \phi_{N-1,i}^{t}. \tag{16}$$

The physical interpretation of these boundary conditions and domain implies a container in which the bottom and top sides correspond to different phases or components of a fluid, and the right and left sides represent the walls of the container.

The structure of the GPU implementation of eq. (4.2) is shown in the algorithm 1.2.

5 Results

In this section we present the numerical results for the algorithms showed in the section 4.

The Fig. 4 shows the process of phase separation in 1-D, i.e. the numerical solution for eq. (8). The Fig. 4(a) shows the initial condition, which is made of random values -1 and 1. Intermediate steps are shown in Fig. 4(a) and (b). The final phase separation is shown in Fig. 4(d), where the final shape resembles the interface area of Fig. 1.

Moving one step forward, the Fig. 5 shows the process of phase separation in 2-D, i.e. the numerical solution for eq. (4.2). The Fig. 5(a) shows the initial condition, which is made of random values -1 and 1 and represents a non-homogeneous fluid made of two components, one represented by the value -1 and the other represented by 1. Intermediate steps are shown in Fig. 5(a) and (b), where we can see the first separation steps. The final phase separation is shown in Fig. 5(d), where the two fluids are completely separated and the interface area is represented by the middle line.

Algorithm 1.2 CUDA FTCS pseudocode for the 2-D Allen-Cahn equation

```
main() {
 allocate memory
 copy CPU to GPU
 kernelFTCS2D$<<< 1, (N, N) >>> (u_{old}, u_{new})$
 copy GPU to CPU
 free memory
}
__global__ kernelFTCS2D $(u_{old}, u_{new})$ {
 $dx = 2/(N-1)$
 $r_1 = \epsilon dt/dx^2$
 $r_2 = 1 - 2r_1$
 $i = threadIdx.x$
 $j = threadIdx.y$
 $id = j * blocksize.y + i$
 for $t = 1 : T$ do
   if $(id < 0$ AND $id < N)$ then
     $u_{new}[id] = u_{old}[id] + 2r(u_{old}[id+1] - u_{old}[id])$ // Boundary condition
   endif
   if $(id > N(N-1)$ AND $id < N^2)$ then
     $u_{new}[id] = u_{old}[id] + 2r(-u_{old}[id] + u_{old}[id-1])$ // Boundary condition
   endif
   $u_{new}[id] = r_1 u_{old}[id-1] + r_2 u_{old}[id] + r_1 u_{old}[id+1] + dt(u_{old}[id] - (u_{old}[id])^3)$
   $\_\_syncthreads()$
   $u_{old}[id] = u_{new}[id]$
 endfor
}
```

6 Conclusions and future work

6.1 Conclusions

In this work, we present an explicit parallel method to solve the AC equation in a GPU. Our approach represents an easy way to solve in a GPU a nonlinear partial differential equation that models a nonlinear phenomena, in this case, a phase field model.

Our proposed algorithm is based on the assumption that the CFL criterion is always met, due the nature of our problem. As the value of ε is always relatively small, because it represents the interface thickness, is possible to use an explicit approach and relatively big step values without violating the stability criterion.

Our approach avoid the storage of large matrices, due its explicit nature, and this characteristic allows us to solve large systems in the relatively small GPU memory.

6.2 Future work

A detailed comparison and numerical analysis between our explicit solution and an implicit one is part of our ongoing investigations. We think that this way we can establish the relation of performance and numerical accuracy between the two methods.

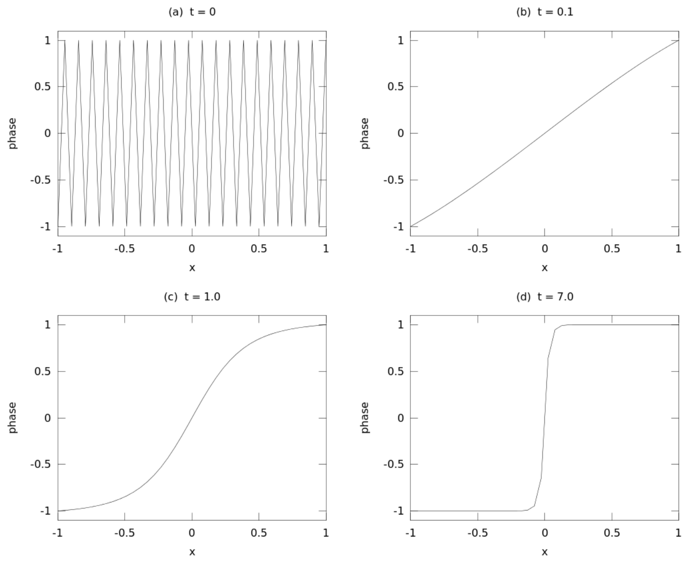

Fig. 4. FTCS scheme for 1-D Allen-Cahn.

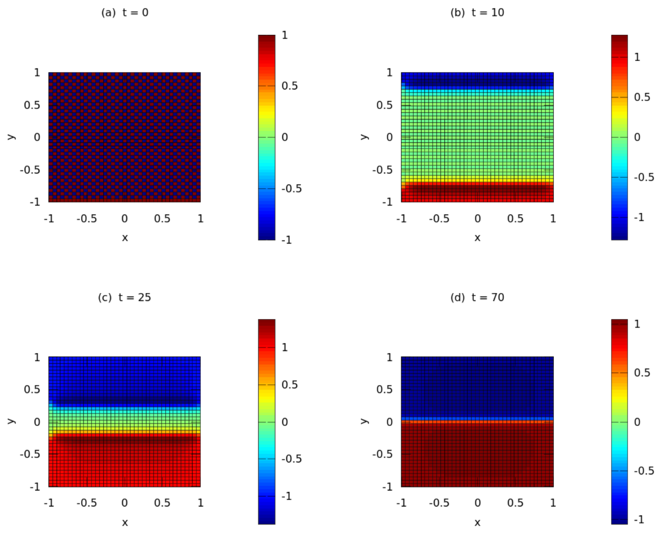

Fig. 5. FTCS scheme for 2-D Allen-Cahn.

As we stated in section 4, ϵ controls the interface thickness of the phase field. However, when we use very small ϵ values, the regular grid for the numerical solution is not longer numerically accurate. At this point, is necessary the use of an adaptive mesh refinement method in order to get a good numerical solution over the interface area. This is a future step of our investigation. In particular, we will seek for Conjugate Gradient methods as the adaptive mesh function.

A comparison between the execution time of an algorithm implemented in a CPU and in a GPU is not the best way to evaluate a CUDA application, although it is the most common way. It is not the best choice, because is not objective, fair nor qualitative. Many factors are involved such as the capacity of the processors, available resources, among others. That is why we think that an execution time comparison as well as an objective measurement of the GPU and CPU resources utilization is the best way to measure a GPU application. For this reason, the objective measurement of the kernels and C code that are used in this work is the next step of the project. To accomplish it, we need to establish adequate metrics and find tools for debugging and profiling the CUDA programs and the C programs, e.g. Compute Visual Profiler and CUDA-gdb.

Acknowledgments. This work has been partially supported by COFAA-IPN and grant IPN-SIP-20120606.

References

1. Allen S. M., Cahn J. W., A microscopic theory for antiphase boundary motion and its application to antiphase domain coarsening, Acta Metallurgica 27 (1979), pp. 1085–1095
2. Badalassi V. E., Ceniceros H. D., Banerjee S., Computation of multiphase systems with phase field models, J. Comput. Phys. 190 (2003), pp. 371–397
3. CUDA C Best Practices Guide (2011)
4. Dechevsky L., Bang B., Gundersen J., Laks A., Kristoffersen A.R., Solving nonlinear systems of equations on graphic processing units, Lect. Notes Comput. Sci. 5910 (2010), pp. 719-729
5. Farber R., CUDA Application Design and Development. Elsevier (2011)
6. Galiano V., Migallon H., Migallon V., Penadés J., GPU-based parallel algorithms for sparse nonlinear systems, J. Parallel Distrib. Comput. 72 (2012), pp. 1098-1105
7. Garcia, N., Parallel power flow solutions using a biconjugate gradient algorithm and a Newton method: A GPU-based approach, In 2010 IEEE Power and Energy Society General Meeting, pp. 1–4 (2010)
8. Huang P., Adbuwali A., A numerical method for solving Alen-Cahn equation, J. Appl. of Math. and Informatics 29 (2011), pp. 1477–1487
9. Kim J., Phase-Field Models for Multi-Component Fluid Flows, Commun. Comput. Phys. 12 (2012), pp. 613–661
10. Kirk D. B., Hwu W. W., In Praise of Programming Massively Parallel Processors: A Hands-on Approach. Elsevier (2010)
11. Lui S. H., Numerical Analysis of Partial Differential Equations. John Wiley and Sons, 2011.
12. NVIDIA CUDA C Programming Guide 5.0 (2012)
13. Wei F., Feng J., Lin H., GPU-based parallel solver via Kantorovich theorem for the nonlinear Bernstein polynomial system, Comput. Math. Appl. 62 (6) (2011), pp. 2506-2517