An Adaptive Step Size GPU ODE Solver for Simulating the Electric Cardiac Activity

VM Garcia[1], A Liberos[2], AM Climent[2], A Vidal[1], J Millet[2], A González[3]

[1]Department of Computing, (DSIC) Universidad Politécnica de Valencia, Valencia, Spain
[2]BioITACA, Universidad Politécnica de Valencia, Valencia, Spain
[3]Department of Communications, Universidad Politécnica de Valencia, Valencia, Spain

Abstract

Simulation of electric cardiac activity requires the solution of a very large system of ordinary differential equations, which requires long computing times. Modern Graphic Processing Units (GPU) are powerful computing devices, which have been used to simulate electric cardiac activity. However, the numerical techniques applied were based on fixed time step. In this paper we describe an adaptive step size solver written for GPUs, and its application to simulate the behavior of a model of 300 atrial cells. Results presented in this study show that a robust adaptive step ODE solver can be implemented in a GPU. As expected, GPU implementations achieved much better performance than CPU solutions. In addition, the presented adaptive methodology achieved a computation time reduction up to a 25% versus a fix step implementation.

1. Introduction

The simulation of electric activity of cardiac cells is crucial for a correct global heart simulation. Heart electric activity is obtained solving an initial value system of ordinary differential equations. Each cell is modeled with 15-40 differential equations. An accurate 3D simulation may require thousands or even millions of cells; hence, the required computing power is huge. Accurate simulation of a few seconds of cardiac activity requires hours or even days of computations.

A possible way to shorten these simulation times is to carry out the calculations in Graphic Processing Units (GPUs). This device takes care of the Graphic User Interface and has a computing power even larger than that of the central processing Unit (CPU). There exist a new generation of GPUs, the General Purpose GPU (GP-GPU) that have been designed specifically for high performance computations. The main drawback for using these devices is that its programming for general tasks is quite complex. This problem has been alleviated to some extent with the CUDA architecture released by NVIDIA.

There are already computer codes written to simulate electric activity in GP-GPU [1-4]; however, given the difficulties in programming these devices, the mathematical algorithms chosen for the simulations were very simple and relatively inefficient: they were fixed time step algorithms, when it is well known that adaptive time step algorithms can be orders of magnitude faster than fixed time step versions.

This paper describes the programming of a GP-GPU adaptive time step algorithm to solve systems of ordinary differential equations, and its application to simulate cardiac electric activity. This algorithm has been programmed using efficient Runge-Kutta-type algorithms and a variable step formulation.

1.1. Mathematical model of heart

A 1D multicellular model (i.e. a string of 300 cells) with realistic human atrial kinetics was developed. The transmembrane potential of each atrial cell was calculated using the Courtemanche et al. model for human atrial cells [5]. Neighbouring cells were coupled by a coupling conductance g, which was fixed to a constant value adjusted to obtain a realistic conduction velocity.

The cardiac tissue was modelled by using the following partial differential equation:

$$\frac{\partial V_i}{\partial t} = -\frac{1}{C_m}\left(I_{total} + \sum_j g_{i,j}\{V_i - V_j\}\right) \qquad (1)$$

where V is the transmembrane voltage, I_{total} summarizes the contribution of all transmembrane currents, C_m is the transmembrane capacitance, and $g_{i,j}$ is the conductance between cells i and j.

1.2. Graphic processing units

GPU are devices developed to handle the graphical display in computers, such as games, computer animations, and virtually all modern applications and operating systems. GPU have many small processing units (cores) which are less powerful than a core of a CPU; however, in recent GP-GPU the number of these cores is quite high (from tens to hundreds, and soon there will be GP-GPU with thousands of cores). If all these cores work efficiently in parallel, the joint computing power can be much higher than that of the CPU. Furthermore, these devices are quite cheap compared with CPU.

Although the CUDA environment has eased the code development for GPGPU, programming these devices still remains hard. The special features of GPGPU architecture must be taken into account in order to obtain efficient codes. The most important of these features are:

1) There must be as few branches as possible in the code.

2) There are several kinds of memory that must be handled explicitly by the programmer. Fast memories such as "shared memory" and "registers" must be used as much as possible, since they are much faster than global memory. However, these "fast" memories are limited in size. If these memories are not enough (which is the most usual case) the programmer must resort to the slower "global" memory.

3) Memory accesses must be "coalesced".

4) Transfer of data between GPU and CPU must be kept to a minimum, since these transfers are quite slow.

The programmer can split the work between the cores by writing "kernels". Many instances of these kernels are executed as "threads". Each thread is executed in a core, and typically hundreds of threads are executed in parallel, with the threads grouped in "blocks" of threads that are executed at the same time.

2. Numerical method

Consider a system of first order time-dependent ordinary differential equations (ODE) $\frac{dY}{dt} = f(t, Y)$ with initial condition (or initial state) $Y_0 = Y(t_0)$, where $Y \in \mathbb{R}^n, f: (\mathbb{R}, \mathbb{R}^n) \to \mathbb{R}^n, t \in \mathbb{R}$.

The solution of this ODE consists of finding out the value of the dependent variables, Y, in latter periods of time, say from t_0 to t_{end}. The simplest way of doing this task is to divide the period $[t_0, t_{end}]$ in k time steps of equal (fixed) length $h = \frac{(t_{end} - t_0)}{k}$ and use a first order approximation of the derivative to "advance" the solution with time, with a simple loop:

```
For j = 0  To k
        $Y_{j+1} = Y_j + h \cdot f(t_j, Y_j)$        (2)
Next j
```

This is known as the Euler explicit scheme; it has the advantage of being easy to program, but neither its accuracy nor its numerical stability are good. This method is conceptually important because virtually all ODE methods share the same structure of "stepping" forward in time. However, there are methods much more accurate and efficient than the Euler method.

Another important matter when solving ODE is the "stiffness" of the problem. An ODE system is said to be "stiff" if, for explicit methods (such as the Euler explicit method) the solution method becomes unstable [6]. In such cases, implicit methods should be used.

The use of parallel computing in this kind of algorithm is restricted to the computations needed for a single iteration of the loop (2), since, clearly, it is not possible to start the computation of an iteration without ending the computation of the previous iteration. However, if the system to be solved is very large, then the computations for a single iteration (for example, computation of $f(t_j, Y_j)$ in loop (2)) can be accelerated through parallel computing (GP-GPU in this case).

2.1. Adaptive step algorithms

The problem of keeping the step fixed in algorithms like the outlined in eq. 2 is that the error in each step depends on the step h, and decreases with h. If we desire to obtain a given accuracy per step, then we must use a small step along the whole simulation.

The key to obtain a good general accuracy without using tiny steps is to estimate the error in each step. There are several ways to estimate the error. In this paper we have used "Embedded Runge-Kutta methods". These are associated pairs of methods of different accuracy, where typically the low order method has accuracy order P and the high order method has accuracy $P + 1$. If both methods are used to take a single step, the difference between both methods can be used to estimate the error made by the lower order method.

Once a step is taken and the error is estimated, it must be decided whether the error is acceptable (if it is below a preset tolerance). If it is not, the step length must be reduced, and through simple formulae [6] a new time step is computed, and the step is recomputed. If the error is acceptable, a new step length is computed, and a new time step is taken. For a system of ODEs, the time step must be computed using the largest error produced in all the dependent variables. So, the error of all variables must be computed and the maximum error must be obtained.

This procedure adapts dynamically the step size to the problem. Typically, the steps become small in regions with strong changes in function f; in those regions where the function is smooth the steps can become very long. All commercial codes for ODEs use adaptive time step, because through this mechanism they provide accuracy and efficiency.

Another advantage is that adaptive stepsize can deal, to some extent, with the matter of the stiffness of the problem. The numerical instabilities are dealt with by reducing the time step, so that acceptable solutions can be found for stiff problems even with explicit methods, although the number of steps can become quite large.

2.2. Adaptation to the problem

The numerical features of a problem can be used to obtain a more efficient code. In this case, through numerical experiments, we found out that this problem does not need a high accuracy (so that the method to be used can be relatively simple) and the problem is slightly "stiff". We have experimented with several explicit embedded Runge-Kutta pairs; we have chosen the simplest one, composed of the explicit Euler method and the Heun method (an explicit second order method) [6] for problems that do not require high accuracy, this is an efficient pair. In this work, we decided to use explicit methods and adaptive step size to deal with the stiffness, although in the future we will experiment with methods appropriate for stiff problems.

The time-stepping nature of the procedure makes it necessary some synchronization, so that the computation of the maximum per-step error and of the new step does not start before all the errors have been computed. To achieve this in our CUDA implementation, we have chosen to split the computations in two kernels,

1) Computation kernel; this kernel performs the bulk of the computations; it computes the new possible solution, and computes the maximum error for all the threads in the block; the maximum errors are kept in a global vector that has as many components as blocks have been launched.

2) Reduction Kernel; once all the computations have finished, a reduction kernel executed by a single block computes the global maximum error of all the system (in parallel computing, this kind of operation is called a reduction); finally, a single thread of this block decides whether the step has been valid, and computes the new time step; if the step was valid, then the simulation time *T* is updated.

Therefore, the basic structure of the main loop is:

```
$T=T_0$
While $T<T_{end}$
 {
   Computation Kernel
   Reduction Kernel
 }
```

The fact that the last part of the reduction kernel is executed by a single core creates a bottleneck in the procedure. However there is no way to solve this problem. On the other hand, once the maximum error is computed, the operations needed to compute the new time step are very simple.

Among other details, it is important to note that present versions have been written using single precision variables, since for some GPUs the performance of GPUs is much higher using single precision than with double precision. Finally, in our codes, each thread takes care of a few cells (even only one); in our model each cell gives rise to 21 differential equations.

3. Testing

In order to evaluate the efficiency of our approach, we compared three different implementations of our simulation code: 1) a fixed step CPU code, compiled with the Intel Compiler ICPC v 11.2 and executed in an hexacore Xeon (executed with a single core), 2) A fixed step GPU code, with the same time step than code 1, run in a Fermi GPU, and 3) The adaptive step code whose main features were described above: (embedded pair Heun-Euler).

For all three implementations a cable of 300 cells was used. Consequently, the model presented 300·21 differential equations. Simulations consisted in 10 seconds with a 250ms periodical stimulation applied to the first cell. For the fixed step cases, a time step of 5us was selected based on the literature [5].

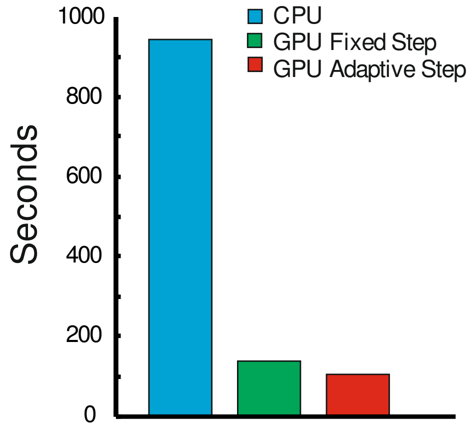

Figure 1. Computation time needed for each one of the three tested solutions.

4. Results

The CPU solution with a fixed step needed 944 seconds in a Dual-Core CPU 3GHz with RAM 4GB. Fixed step and adaptive step GPU solutions needed 137 and 104 seconds respectively (Fig. 1). Simulations were reproduced with different stimulation rates and similar results were obtained.

As visible in Fig. 2, transmembrane potentials obtained by the fixed and adaptive steps were essentially identical. For all three implementations no statistical differences were found for main transmembrane potential properties (i.e. action potential duration measured as the 90% of the repolarization, 149ms, the maximum rate of depolarization measured as the maximum slope of phase 0 (i.e. dv/dt), 112 V/s, resting membrane potential, -83mV, and maximum transmembrane potential, 28mV).

However, when action potentials from all cells during each beat were compared, the standard deviation of the maximum transmembrane potential was higher for the fixed step algorithm than for the adaptive (i.e. 9.1mV vs.

3.5 mV, p<0.01). The explanation for this difference lies in the advantage of the adaptive algorithm to reduce the time step close to points were fast variations appear and increase the accuracy of the measured solution.

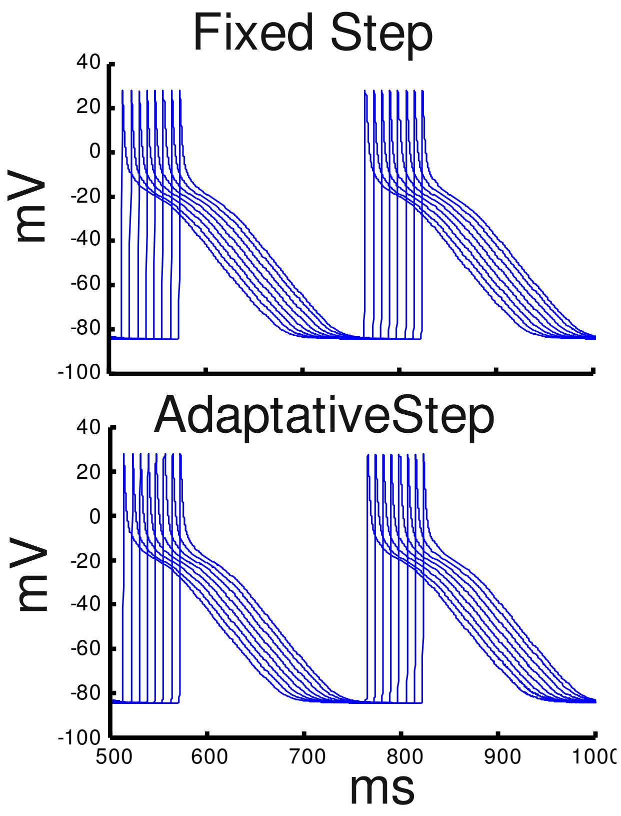

Figure 2. Illustration of the transmembrane potential measured for many cells along length of the 1D multicellular model. Top and bottom panels represent GPU fixed step and adaptive step respectively.

5. Discussion

In this study the possibility to apply adaptive step ODE solvers in General-Purpose Graphics Processing Units has been presented for the first time. Performance of this new implementation has been compared with classical fix step solutions both in CPU and GPU. As presented in previous publications [1-4], GPU implementations achieved much better performance than CPU solutions. Furthermore, the presented adaptive methodology achieved a computation time reduction up to a 25% versus a fix step implementation. In addition to this, the ability of adaptive algorithm to increase the accuracy of the calculated action potential morphology has been illustrated.

GPU structure is appropriate for the parallel solution of ODE systems. However, the solution of the partial differential equations that define the tissue propagation may be a GPU bottleneck. The extension of the present work to 2D and 3D anatomical structures will elucidate the ability of adaptive algorithms to overcome those problems.

Another important concern in programming adaptive methods by using GPUs is the nonavailability of tools and libraries for the implementation of several algorithms. That implies that the major number of applied methods needed a hand-coded implemented. However, recently different authors have developed automatic or pseudo-automatic tools to generate GPU executable code from original C code [3].

The presented methodology should be considered as a new step in the approximation to real-time cardiac simulation mathematical models. Our next step will be to introduce implicit methods, more appropriate to the "stiff" nature of the problem.

Acknowledgements

This research was supported by Spanish Ministry of Education and Science under the projects TEC2009-13939, TEC2009-13741 and TIN2008-06570-C04-02), by Generalitat Valenciana through project PROMETEO/20099/013, and by the Universitat Politecnica de Valencia through its research initiative program.

References

[1] Nimmagadda V, Akoglu A, Hariri S, Moukabary T. Cardiac simulation on multi-GPU platform. The Journal of Supercomputing 2011;1-19
[2] Vigmond EJ, Boyle PM, Leon L, Plank G. Near-real-time simulations of biolelectric activity in small mammalian hearts using graphical processing units. Conf Proc IEEE Eng Med Biol Soc 2009:3290-3.
[3] Lionetti FV, McCulloch AD, Baden SB. Source-to-source optimization of CUDA C for GPU Accelerated Cardiac Cell Modeling. Europar 2010:38-49.
[4] Sato D, Xie Y, Weiss JN, Qu Z, Garfinkel A, Sanderson AR. Acceleration of cardiac tissue simulation with graphic processing units. Med Biol Eng Comput 2009;47:1011–1015.
[5] Courtemanche M, Ramirez RJ, Nattel S. Ionic mechanisms underlying human atrial action potential properties: insights from a mathematical model. Am J Physiol. 1998;275:H301-21.
[6] Ascher UM, Petzold L. Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations SIAM, Philadelphia 1998.

Address for correspondence.

Victor M. García
DSIC, Universidad Politécnica de Valencia,
46022, Valencia, SPAIN
vmgarcia@dsic.upv.es