Contribution Title*

First Author[1][0000−1111−2222−3333], Second Author[2,3][1111−2222−3333−4444], and Third Author[3][2222−−3333−4444−5555]

[1] Princeton University, Princeton NJ 08544, USA
[2] Springer Heidelberg, Tiergartenstr. 17, 69121 Heidelberg, Germany
lncs@springer.com
http://www.springer.com/gp/computer-science/lncs
[3] ABC Institute, Rupert-Karls-University Heidelberg, Heidelberg, Germany
{abc,lncs}@uni-heidelberg.de

Abstract. The abstract should briefly summarize the contents of the paper in 150–250 words.

Keywords: First keyword · Second keyword · Another keyword.

1 Introduction

The contraction and relaxation of the heart are results of propagation of an electrical wave through cardiac tissue. Deficiencies on this wave propagation may cause abnormal behavior of the heart, resulting problems like ventricular tachycardia and fibrillation, which in a more severe situation can lead to death. Many mathematical model are available, in order to simulate the electrical wave propagation through cardiac tissues.

Eletrophysiology models for cardiac tissue have a high computational demand, mainly for three dimensional simulations. Some author address this problems by using high computational resource, such as clusters. More recently, some of these models have been ported to GPUs, which allow their execution in lower cost platforms and opening real time simulation possibilities. Although, the laplacian operator present on Eletrophysiology models makes difficult to efficiently parallelize them on GPUs [1, 12, 15].

In this work, we propose an approach to reduce computational time when solving the laplacian with the finite difference method. We overlap time steps computation, reducing the number of global memory accesses. We used a GPU implementation of the 2D Karma model for cardiac tissue [9], and we show the proposed approach can reduce hole simulation time by about 30%. As a base for our evaluations, we have used the classical parallel laplacian implementation firstly proposed by Micikevicius in 2009 [11].

This paper is organized as follows. Section 2 gives an overview of modeling and simulation of cardiac dynamics, including the standard second-order Laplacian discretization. Section 3 describes the classic parallel GPU solution proposed

* Supported by organization x.

by Micikevicius [11]. Section 4 presents our proposed parallel approach through the GPU architecture. Section 5 presents and discusses the results achieved by the proposed approach. Section 5 provides an overview of related prior work. Finally, Section 6 presents the conclusions and future work.

2 Computational simulation of electrical dynamics in cardiac tissue

The electrical wave propagation in cardiac tissue can be described by a variation in time of the cell membrane's electrical potential U, for each cardiac tissue cell. Under a continuum approximation, this process can be represented using the following reaction-diffusion equation:

$$\frac{\partial U}{\partial t} = \nabla \cdot D\nabla U - \frac{I_{ion}}{C_m} \ . \tag{1}$$

The first term at the right side represents the diffusion component, and the second term on the right side represents the reaction component, where I_{ion} corresponds to the total current across the cell membrane and C_m the constant cell membrane capacitance. The diffusion coefficient D can be a scalar or a tensor and describes how cells are coupled together; it also may contain some information about tissue structure, such as the local fiber orientation [4]. The value of D affects the speed of electrical wave propagation in tissue [4, 1].

The reaction term is modeled by a system of nonlinear ordinary differential equations, such as described in Equation 2:

$$\frac{d\mathbf{y}}{dt} = \mathbf{F}(\mathbf{y}, U(\mathbf{y}, t), t) \ ; \tag{2}$$

where

The exact formulation and the number of variables depends on the level of complexity of the electrophysiology model. For each additional variable $\mathbf{y_j}$, $\mathbf{F_j}(\mathbf{y_j}, V, t)$ corresponds to a nonlinear function. Clayton et al. provide a good review about cardiac electrical activity modelling[4].

There are many mathematical models that describe cellular cardiac electrophysiology [2, 9, 8, 14, 3, 13]. The main difference among them lies in the number of differential equations, in order to represent mechanisms responsible for ionic currents across the cell membranes and changes of ion concentrations inside and outside cells. However, all these models affect only the specification of I_{ion} and they employ the same diffusion term.

2.1 Karma model

In 1993, Karma proposed one of the simplest models used to describe cardiac electrical dynamic ([9]). The model has two variables and consists of the following differential equations:

$$\frac{\partial U}{\partial t} = D\nabla^2 U - U + \left(\left[1 - \tanh(U-3)\right] U/2\right) \left[\gamma - \left(\frac{v}{v^*}\right)^{xm}\right] \tag{3}$$

$$\frac{dv}{dt} = \epsilon \left[\Theta\left(U-1\right) - v\right], \tag{4}$$

where U represents the electrical membrane potential and v is a recovery variable. The main purpose of Karma model is to simulate the behavior of spiral waves, such as those associated with cardiac arrhythmia.

2.2 Numerical solution

Finite difference methods (FDM) are a common numerical methods used to obtain a numerical solution for many different numerical problems [6]. This method requires a domain discretization for all variables.

In order to solve the differential equations of the Karma model on a GPU using FDM, we adopt a forward difference for time integration at time t_n and a second-order central difference approximation for the spatial derivative at position $p = (x_i, y_j, z_k)$. In addition, we assume $h = \Delta x = \Delta y = \Delta z$ and $t_n = n\Delta T$. Therefore, the finite difference approximation for space (cell tissue) and time is modeled as follows:

$$\begin{aligned} U^{n+1}_{i,j,k} = & \left(1 - \Delta T\right) U^n_{i,j,k} + \\ & + \tfrac{D\Delta T}{h} \left(U^n_{i+1,j,k} + U^n_{i-1,j,k} + U^n_{i,j+1,k} + U^n_{i,j-1,k}\right. \\ & \left. + U^n_{i,j,k+1} + U^n_{i,j,k-1} - 6U^n_{i,j,k}\right) + \\ & + \Delta T \left(0.5 * \left[1 - \tanh(U^n_{i,j,k} - 3)\right] U^n_{i,j,k}\right) \left[\gamma - \left(\tfrac{v^n_{i,j,k}}{v^{n*}_{i,j,k}}\right)^{xm}\right], \end{aligned} \tag{5}$$

where n is an index corresponding to the n^{th} time step. For the numerical experiments presented in this paper, D corresponds to a uniform field that applies the value one to all points in the domain.

3 Classic Laplacian implementation on a GPU

In this section we discuss some details of the GPU cache memory and present the shared memory approache for dealing with the spatial dependency imposed by the Laplacian discretization on this particular architecture. This approach, which is based on simple row major order [1], can works as base for some three dimensional propagation problems [11, 10, 1, 12, 5].

GPUs are specialized hardware that can be used to process data in a massively parallel way. When the computation may be performed at each memory position independently, GPUs can achieve much better performance than CPUs. The CUDA parallel programming model is a powerful tool to develop general

purpose applications for Nvidia GPUs. It is based on threads that can be addressed in one, two, or three dimensional arrays. Despite the arrangement chosen by the developer, GPU executes threads following a linear order in groups of 32 consecutive threads at same time, called warps. Additionally, GPU cache memory considers that consecutive threads access consecutive memory addresses. Thus, CUDA developers take advantage of row major order to store/load data in GPU memory. This approach seeks to minimize non-coalesced memory operations by storing and loading several consecutive data values in cache given a memory address access. The specialized GPU architecture may store or load up to 128 consecutive bytes in one cache line with a single read/write operation.

The usual parallel approach to compute time-explicit FDM on a GPU consists of addressing each point of the mesh using one CUDA thread [1, 5, 10]. For each new time step, the GPU computes thousands of points in parallel, solving Equation 5 on each point of the mesh, where the only dependency is temporal.

However, 2D or 3D domains present great challenges in minimizing the memory latency, since accessing neighboring data points in these domains may cause many non-coalesced memory operations to obtain nearest neighbors, which are required for the spatial discretization [10]. This compromises GPU performance due to the greater elapsed time to access all neighbors.

The stencil pattern illustrated in Figure 1 represents the required data to compute the value at the next time step $t+1$ at a given point p. When computing single precision values, its neighbors in the x-direction are offset by 4 bytes to the left or right, while the offset of neighbors in the y-direction depends on the domain size. Clearly, in this case the goal of accessing only nearby memory locations cannot be achieved.

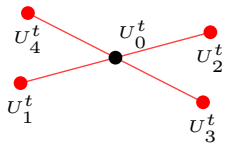

Fig. 1. 2D stencil representing data required for calculating the value at the next numerical time step $t + 1$ at each domain point U_0 using a standard second-order FDM.

4 2D space-time blocking on GPU

One possible and trivial approach to avoid redundant global memory access is to store all data needed by a thread block in its shared memory. Figures 2, 3 and 4 depicts an example on how we can load data to shared memory taking advantage of row major ordination [11].

The simulation of seconds of an electrical wave propagation through cardiac tissue requires thousands of time steps, in an acceptable time discretization.

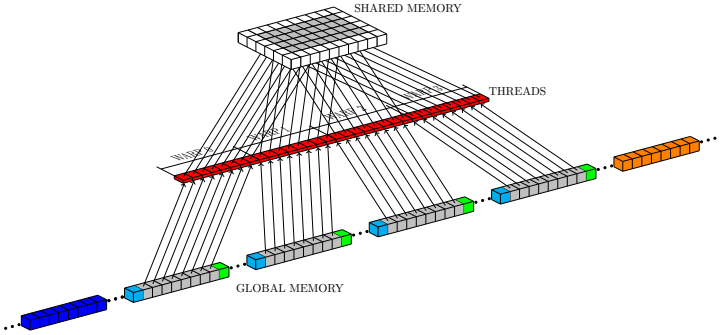

Fig. 2. Access pattern for core data on global memory.

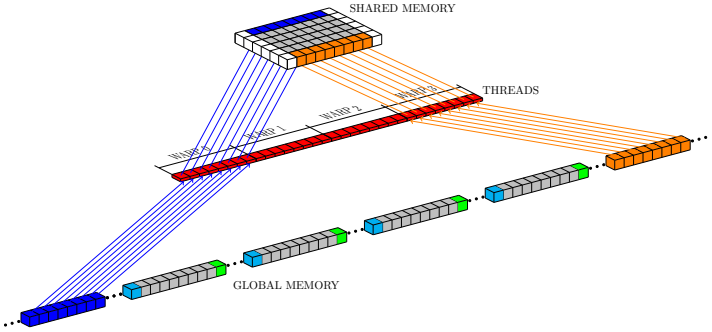

Fig. 3. Access pattern for y neighborhoods.

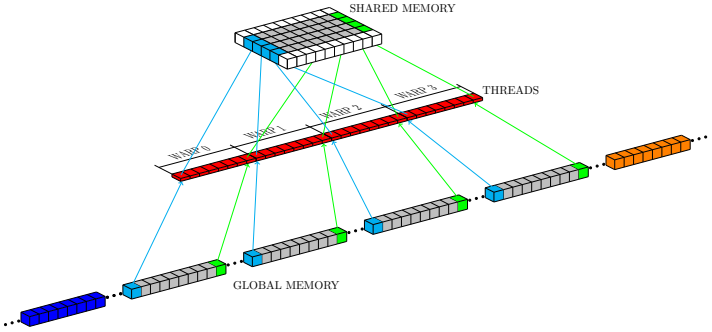

Fig. 4. Access pattern for x neighborhoods.

Taking advantage of it, an approach to decrease computing time, is negotiating computation and accesses of shared memory for global memory accesses [7]. To do this, in each block, all necessary data for t_n time steps is transferred from global memory to shared memory.

The use of shared memory can help avoiding to loose time with redundant global memory access. Although, CUDA thread blocks do not have any kind of data synchronization. For this reason, on every time step of the FDM, it is necessary to update global memory with the new values computed by the thread blocks. This means that for all time step computations it is necessary to read an write data in the global memory. We can reduce global memory transactions applying an approach called space-time blocking. The global memory must be updated only every t time steps, which is the amount of data, necessary to execute t steps calculation. These data are copied from global memory to shared memory in the first time step, and, in the t-step, the result is written back. The other time steps computation are made in shared memory.

Observe in figure 5, that is a representation of a 2D domain tile, the green middle element is used for your stencil computation, and, all your stencil neighbors computation, uses the green middle element as a neighbor, so, for a $k = 4$ stencil calculation, the reuse of a element excluding, border conditions, is 9 times, for read operations, and, one for the result writing. Copying the data to shared memory, the global memory accesses is reduced to 2 times, only for the tile reading an the result writing on global memory, the remaining memory operations, occurs, in shared memory.

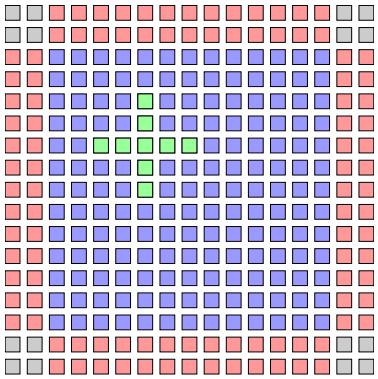

Fig. 5. Shared memory tile, the elements in green color represent the neighbors needed to the middle stencil computation, the red elements are the halo region and blue elements are the elements to compute in this block

On domain division into blocks, represented by the yellow color in the figure 6, shows the shared memory for the computation, of, one time step, the resulting blocks have independent shared memory. The thread or block communication, is out of question. For each block, yours threads, have to copy the data from global memory to shared memory, do the computation and write the results back to the global memory.

The number of threads is equal to the number of elements to be calculated, but,the size of elements to be copied, is greater than the number of threads. The

elements are distributed, linearly, across the threads, including elements that is not necessary to calculate, avoiding, code complexity. These elements are show in figure 5 and 6 with the gray color.

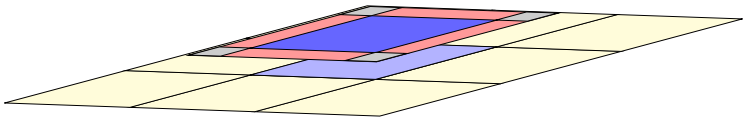

Fig. 6. Global memory blocks division in yellow grid, with tile copied to shared memory, when, the elements to be computed as show in blue color, the halo region in red, and unused elements in gray

$$S = 3 * (Bx + k * t) * (By + k * t) \tag{6}$$

The equation 6 means the memory, needed to be copied for the t time step computations, S is the total shared memory elements, Bx and By are the block dimension, k is the stencil size and t is the number of time steps. The number three, serves for, triplicating the shared memory, because, for the tile stencil computation, we have to store the old electrical potential data named read memory data, store the new computed potential data named write memory data, and, the V data that is accessed two times for each time step computation, but your value do not be in registers, because the, disparity between the number of threads and the number of elements to be calculated at the middle time step computations. The data is not really copied, but, the pointer that indicates the read memory, is, exchanged by the pointer, that indicates the write memory.

The tile copy required for three time steps calculations, are show in figure 7 as B element, where, in the red color have border elements and in blue color the elements required for the C step calculation, the same occurs between C and D elements.

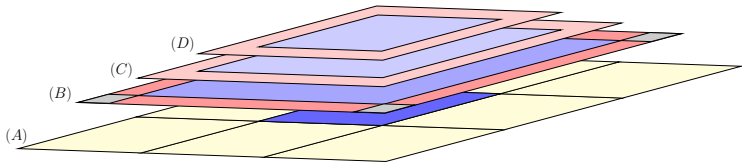

Fig. 7. Space time blocking example, of, shared memory copies in each time step, the A element shows the global memory, B is the shared memory tile required for 3 time steps, C and D represent the elements that have to be computed for the three time steps calculation.

The number of threads is equal to the number of elements to be calculated at final step show as blue, in D element, in figure 7, but, in the previous time steps,

C and B, the size of elements to be calculated, is greater than the number of threads show as blue. The elements are distributed, linearly, across the threads, including elements that is not necessary to calculate, avoiding, code complexity.

The work is distributed to threads, linearly. The technique used is named, stride, this means, the work is divided by the number of threads, the resulting is the number of strides, the threads have to compute.

The number of time steps added to the blocks size, need to be multiple of the global memory bandwidth for data coalescence. In the computation stage, all data be in shared memory and the problem, means, about thread disparity in the final of the block execution, for this, the size of tiles, of all step time calculations, have to be multiple of 32, that means the warp size.

GPU's are limited by the number of, threads per block and a size of shared memory, but, the memory acces performance difference, make sense of maximize the use of these hardware features. The choose of the tile size, is influenced by, the stencil size, and the hardware limitations, like a shared memory size. Other concerns are about performance, is desired that the number of elements to be calculated, is a multiple of the number of threads in a warp. Copies and previous time steps, need to be done at a different number of elements, not necessarily, multiple of the number of threads, and it's challenging to fit the best tile size and time step size, to maximize computation time [7].

5 Results

Two approaches of the stencil computation, is compared in two tests, for 96x96 and 768x768, 2D size domain, 19000 time steps, and the k equal to 2. In the two tests, the time step size t_n is increased and this calculation time is analyzed. Shared Memory means the simplest approach, when, the memory is copied from global memory to shared memory in all time steps. Space Time Blocking means the new approach that is present in this paper. Observing the two charts in figure 8 and figure 9 we can see, that the best choice of time steps to be copied to shared memory is close to 6 times for 96x96 size domain, for a larger domain, the result is the same, proving the scalability of the space time blocking approach.

The shape of time space blocking curves , be a exponential curve, the behavior, is explained, by the ammount of data to be calculated, in this aproach, is proportionally to a pyramid volume. Increasing the size of time steps copied to shared memory, more computations is needed but less global memory acces is needed, and this exchange rate is increased by the number of computations is comparable to a pyramid volume, when the, height means the time steps at the same time in shared memory, and the size of the resulting computations remains equal.

Spiral simulation in fig 10.

References

1. Bartocci, E., Cherry, E.M., Glimm, J., Grosu, R., Smolka, S.A., Fenton, F.H.: Toward real-time simulation of cardiac dynamics. In:

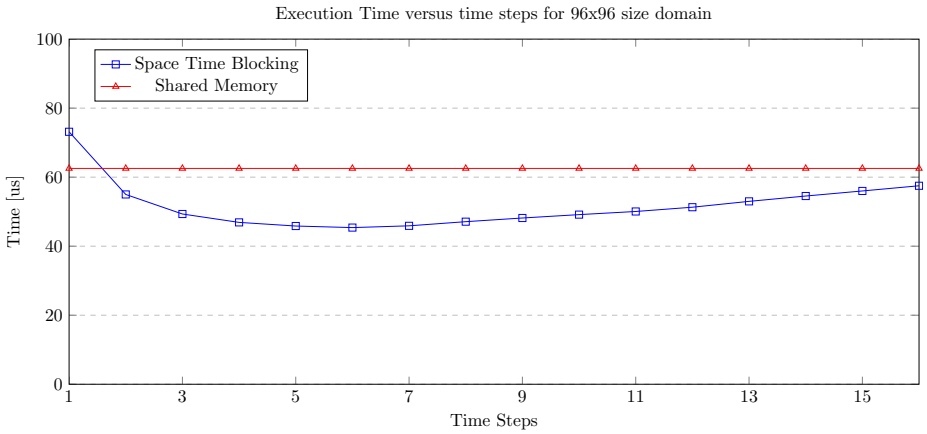

Fig. 8. A figure caption is always placed below the illustration. Please note that short captions are centered, while long ones are justified by the macro package automatically.

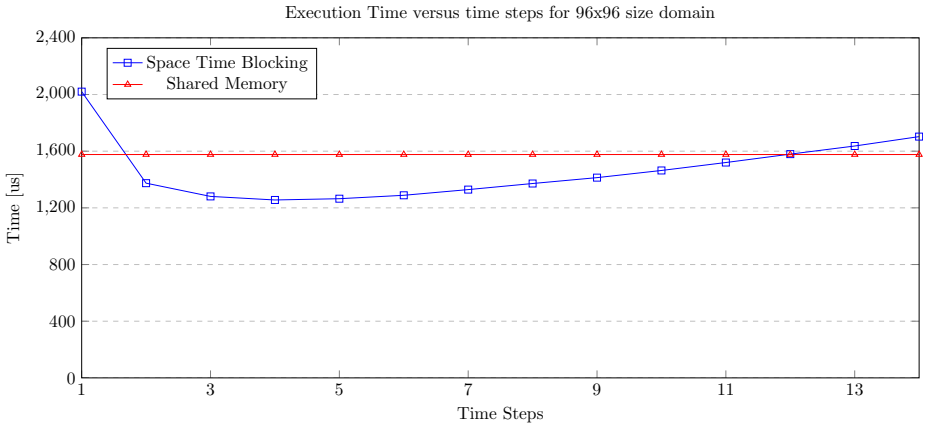

Fig. 9. A figure caption is always placed below the illustration. Please note that short captions are centered, while long ones are justified by the macro package automatically.

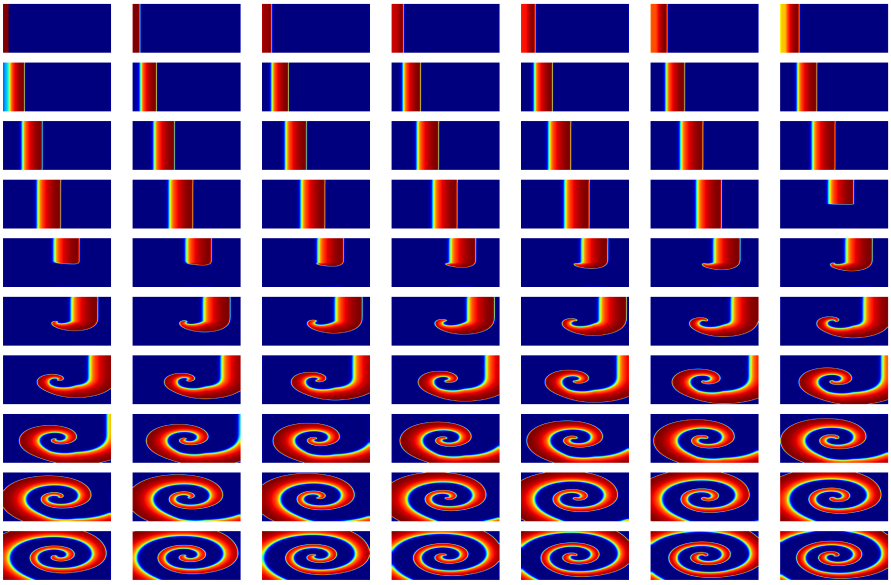

Fig. 10. Spiral simulation

Proceedings of the 9th International Conference on Computational Methods in Systems Biology. pp. 103–112. CMSB '11, ACM, New York, NY, USA (2011). https://doi.org/10.1145/2037509.2037525, http://doi.acm.org/10.1145/2037509.2037525

2. Beeler, G.W., Reuter, H.: Reconstruction of the action potential of ventricular myocardial fibres. the Journal of Physiology **268**(1), 177–210 (1977), http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1283659/
3. Bueno-Orovio, A., Cherry, E.M., Fenton, F.H.: Minimal model for human ventricular action potentials in tissue. Journal of Theoretical Biology **253**(3), 544–560 (2008). https://doi.org/10.1016/j.jtbi.2008.03.029, http://www.sciencedirect.com/science/article/pii/S0022519308001690
4. Clayton, R., Panfilov, A.: A guide to modelling cardiac electrical activity in anatomically detailed ventricles. Progress in Biophysics and Molecular Biology **96**(1-3), 19–43 (2008). https://doi.org/http://dx.doi.org/10.1016/j.pbiomolbio.2007.07.004, http://www.sciencedirect.com/science/article/pii/S0079610707000454, cardiovascular Physiome
5. Giles, M., László, E., Reguly, I., Appleyard, J., Demouth, J.: Gpu implementation of finite difference solvers. In: Proceedings of the 7th Workshop on High Performance Computational Finance. pp. 1–8. WHPCF '14, IEEE Press, Piscataway, NJ, USA (2014). https://doi.org/10.1109/WHPCF.2014.10, http://dx.doi.org/10.1109/WHPCF.2014.10
6. Hoffman, J.D., Frankel, S.: Numerical methods for engineers and scientists. CRC press (2001)

7. Holewinski, J., Pouchet, L.N., Sadayappan, P.: High-performance code generation for stencil computations on gpu architectures. In: Proceedings of the 26th ACM International Conference on Supercomputing. pp. 311–320. ICS '12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2304576.2304619, http://doi.acm.org/10.1145/2304576.2304619
8. Iyer, V., Mazhari, R., Winslow, R.L.: A computational model of the human left-ventricular epicardial myocyte. Biophysical Journal **87**(3), 1507–1525 (2004). https://doi.org/10.1529/biophysj.104.043299, http://www.sciencedirect.com/science/article/pii/S0006349504736346
9. Karma, A.: Spiral breakup in model equations of action potential propagation in cardiac tissue. Phys. Rev. Lett. **71**, 1103–1106 (Aug 1993). https://doi.org/10.1103/PhysRevLett.71.1103, http://link.aps.org/doi/10.1103/PhysRevLett.71.1103
10. Michéa, D., Komatitsch, D.: Accelerating a three-dimensional finite-difference wave propagation code using gpu graphics cards. Geophysical Journal International **182**(1), 389–402 (2010). https://doi.org/10.1111/j.1365-246X.2010.04616.x, http://dx.doi.org/10.1111/j.1365-246X.2010.04616.x
11. Micikevicius, P.: 3d finite difference computation on gpus using cuda. In: Proceedings of 2nd workshop on general purpose processing on graphics processing units. pp. 79–84. ACM (2009)
12. Nimmagadda, V.K., Akoglu, A., Hariri, S., Moukabary, T.: Cardiac simulation on multi-gpu platform. The Journal of Supercomputing **59**(3), 1360–1378 (2012). https://doi.org/10.1007/s11227-010-0540-x, http://dx.doi.org/10.1007/s11227-010-0540-x
13. O'Hara, T., Virág, L., Varró, A., Rudy, Y.: Simulation of the undiseased human cardiac ventricular action potential: Model formulation and experimental validation. PLOS Computational Biology **7**(5), 1–29 (05 2011). https://doi.org/10.1371/journal.pcbi.1002061, https://doi.org/10.1371/journal.pcbi.1002061
14. ten Tusscher, K.H.W.J., Panfilov, A.V.: Alternans and spiral breakup in a human ventricular tissue model. American Journal of Physiology - Heart and Circulatory Physiology **291**(3), H1088–H1100 (2006). https://doi.org/10.1152/ajpheart.00109.2006, http://ajpheart.physiology.org/content/291/3/H1088
15. Vasconcellos, E.C., Clua, E.W., Fenton, F.H., Zamith, M.: Accelerating simulations of cardiac electrical dynamics through a multi-gpu platform and an optimized data structure. Concurrency and Computation: Practice and Experience **32**(5), e5528 (2020). https://doi.org/10.1002/cpe.5528, https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5528, e5528 cpe.5528