

GPU Memory Access Optimization for 2D Electrical Wave Propagation Through Cardiac Tissue and Karma Model Using Time and Space Blocking

Christian Willian Siqueira Pires(✉), Eduardo Charles Vasconcellos(✉), and Esteban Walter Gonzalez Clua(✉)

Institute of Computing, Fluminense Federal University, Niterói, RJ, Brazil
{cwspires,esteban}@id.uff.br, charles.edu@gmail.com
http://www.uff.br

Abstract. Abnormal heart function is one of the largest causes of death in the world. Understanding and simulating its contraction and relaxation movements may reduce this mortality and improve specific treatments. Due the high processing time to compute propagation of an electrical wave through cardiac tissue, interactive and real time simulations are a challenge, making unable corrective treatments on the fly. In this paper we propose an approach to reduce the electrophysiology models simulation computational time using the finite difference method to solve the laplacian operator, required by the karma model. Our solution optimizes accesses to global memory by storing more than one instant of time in shared memory within each GPU kernel call. We present tests that validate our proposal and show an increase performance close to 20% when compared with available GPU implementations.

Keywords: Gpu computing · Cardiac electrophysiology models · Parallel cardiac dynamics simulations · Finite difference method

1 Introduction

The contraction and relaxation of the heart are results of the propagation of electrical wave through cardiac tissue. Deficiencies on this wave propagation may cause abnormal behavior of the heart, resulting in problems like ventricular tachycardia and fibrillation, which in a more severe situation can lead to death. Many mathematical models are available for simulating the electrical wave propagation through cardiac tissues. These models help to simulate heart conditions, like tachycardia and fibrillation, and the effects of different treatments.

Electrophysiology 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 or supercomputers [5,24,25,29]. More recently, some of these models have been ported to

© Springer Nature Switzerland AG 2020
O. Gervasi et al. (Eds.): ICCSA 2020, LNCS 12249, pp. 376–390, 2020.
https://doi.org/10.1007/978-3-030-58799-4_28

GPUs, which allow their execution in lower cost platforms and opening real time simulation possibilities. Although, the Laplacian operator, frequently used on electrophysiology models, makes difficult to efficiently parallelize them on GPUs [1,26,33].

In this work, we propose a novel 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 [18], and we show that our proposed approach can reduce the complete simulation time by about 30%. Karma is a two variables model that is reliable to reproduce abnormal heart conditions like spiral waves associated with cardiac arrhythmia [10,11,18]. As a base for our evaluations, we have used the classical parallel Laplacian implementation firstly proposed by Micikevicius [23].

This paper is organized as follows. Section 2 gives an overview of the modeling and simulation of cardiac dynamics, including the standard second-order Laplacian discretization. Section 3 describes the classic parallel GPU solution proposed by Micikevicius [23]. Section 4 presents our proposed parallel approach through the GPU architecture. Section 5 presents and discusses the results achieved by our solution. Finally, Sect. 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 [7]. The value of D affects the speed of electrical wave propagation in tissues [1,7].

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

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

Its exact form depends on the level of complexity of how the electrophysiology model describes heart tissue cells. For each additional variable $\mathbf{y_j}$, $\mathbf{F_j}(\mathbf{y_j}, U, t)$ is

a nonlinear function. Clayton et al. provide a good review of cardiac electrical activity modelling [7].

The system of differential equations in reaction term represents mechanisms responsible for ionic currents across the cell membranes and changes of ion concentrations inside and outside cells [7,11]. There are many mathematical models that describe cellular cardiac electrophysiology [2,3,16,18,28,32]. The main difference among them lies in the reaction term.

2.1 Karma Model

In 1993, Karma proposed one of the simplest models used to describe cardiac electrical dynamic [18]. 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 [10,11,18].

2.2 Numerical Solution

Finite difference methods (FDM) are a common numerical methods used to obtain a numerical solution for many different numerical problems [14]. 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 nth 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 approach 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 [1,12,22,23,26].

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 [1,12,22]. For each new time step, the GPU computes thousands of points in parallel, solving Eq. 5 for each point of the mesh, where the only dependency is the temporal domain.

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 [22]. This compromises GPU performance due to the greater elapsed time to access all neighbors, due the reduction of concurrency features.

The stencil pattern illustrated in Fig. 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.

4 2D Space and Time Blocking on GPU

One possible approach to avoid redundant global memory access is to store all required data of a thread block in its shared memory [23]. Figures 2, 3 and 4 depicts an example on how to load data to shared memory, taking advantage of row major ordination.

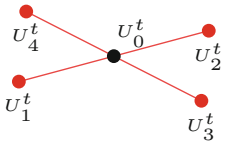

Fig. 1. 2D stencil representing data required for calculating the value at the next numerical time step $t + 1$ at each domain point U_0 using a standard second-order FDM.

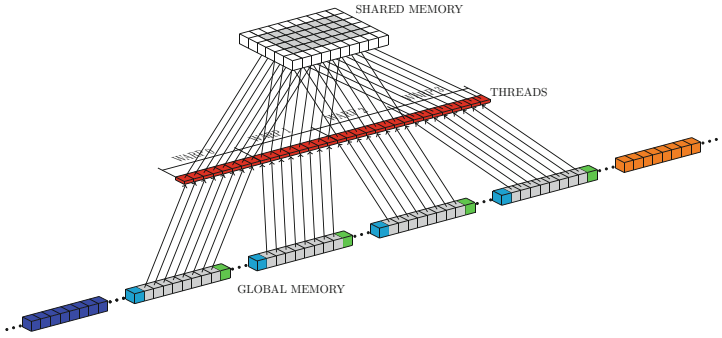

Fig. 2. Access pattern for core data on global memory.

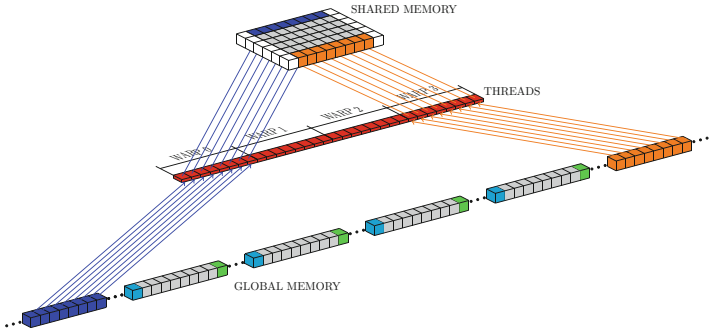

Fig. 3. Access pattern for y neighborhoods.

A possible approach to decrease computing time is balancing computation and accesses of shared memory with global memory accesses [15]. To do this, in each block, all necessary data for t_n time steps is transferred from global memory to shared memory. Doing so, some data need to be recalculated.

The use of shared memory can help avoiding to loose time with redundant global memory access. However, 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

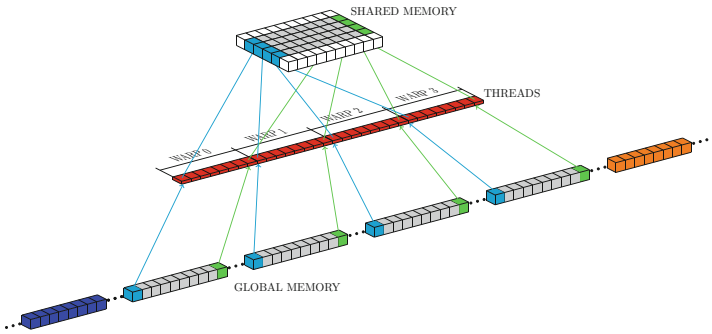

Fig. 4. Access pattern for x neighborhoods.

blocks. This means that for all time step computations it is necessary to read and write data in the global memory. We propose to reduce global memory transactions applying an approach called space-time blocking. The global memory must be updated only every t_n time steps, which is the amount of data necessary to execute n steps calculation. These data are copied from global memory to shared memory in the first time step t, and, in the following $t + t_n$, the result is written back. The time steps between t and $t + t_n$ computations are made accessing only shared memory.

The domain division through blocks, represented by yellow in Fig. 5, shows the shared memory for one time step computation, where the resulting blocks have independent shared memory. In this case, the thread or block communication is not being considered. The threads from each block have to copy the data from global memory to shared memory, execute the required computation and write the results back to the global memory.

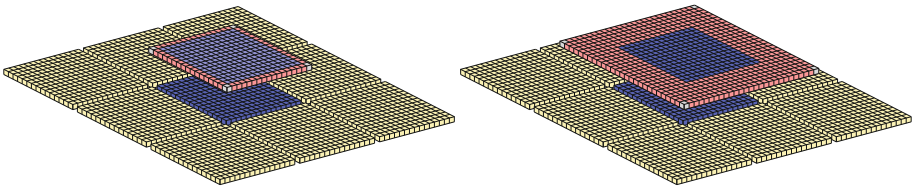

Fig. 5. Global memory blocks division in yellow grid, with tile copied to shared memory for one time step computation. The result elements to be computed is represented by dark blue, the border region in red, and unused elements in gray. (a) shows the shared memory size copies for one time step, and, (b) shows for six time steps (Color figure online)

Figure 6 shows a representation of a 2D domain tile, where the green middle element is used for the stencil computation. All the stencil neighbors computa-

tion use the green middle element as a neighbor. In this case, for a $k = 2$ stencil calculation, the reuse of an element excluding the border conditions correspond to 5 read and one write operations. When copying the data to shared memory, the global memory accesses is reduced to 2 times, corresponding only to one tile reading an the result writing on the global memory. The remaining memory operations remains only at the shared memory [23]. This memory reuse occurs in all time step computation, and the reuse is multiplied for each additional time step copied to shared memory.

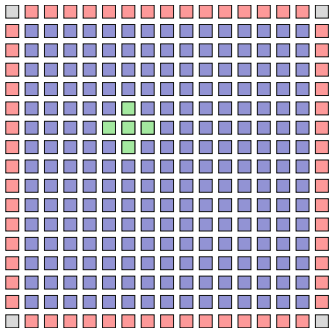

Fig. 6. Shared memory tile for $k = 2$, where green elements represent the neighbors required for the middle stencil computation, the red elements are the halo region and blue elements are the elements to compute in this block (Color figure online)

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

Equation 6 represents the memory required for n time step computations, where S is the total shared memory elements, Bx and By are the block dimension, k is the stencil size and n is the number of time steps. The number three corresponds to triplicating the amount of shared memory. This happens due the tile stencil required to store the variables described in Sect. 2.1, which are, the old electrical potential data, the new step potential data and the recovery variable v, data required to the karma model. v data is accessed two times for each time step computation, but its value is not present at the registers, due the disparity between the number of threads and the number of elements to be calculated at the middle time step computations.

The number of threads is equal to the number of elements to be calculated at the final time step computation. This is shown as dark blue elements in Figs. 5 and 7, although the size of elements to be copied is greater than the number of threads. The elements are linearly distributed across the threads, including elements that are not necessary to calculate, avoiding code complexity. The work is distributed to threads in a linearly way using a stride pattern, which means that the work is divided by the number of threads and the result is the number of strides necessary to compute. These elements are show in Fig. 6 and 5 with the gray color. In order to copy the shared memory, copying the size difference

between the number of threads and the number of elements require the same process.

The tile copy required for six time steps calculations is show in Fig. 7 as the biggest tile, where red represent border elements required the tile below computation. The light blue elements represents the data to be calculated for the tile below. All data copied at the biggest tile, over the dark blue elements represents the repeated computation that does not occur in the implementation of shared memory, but, the computations .

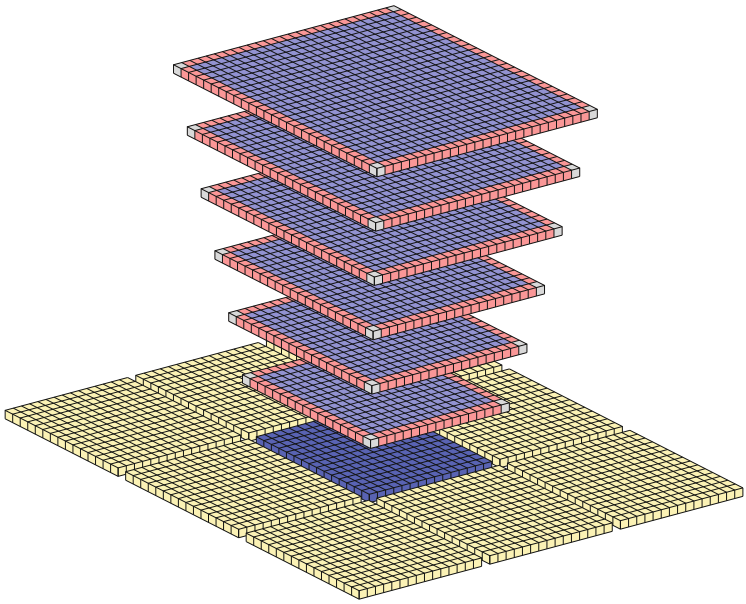

Fig. 7. Space time blocking memory representation. The yellow color shows the global memory blocks division grid. The dark blue elements is the result of the all computation. The biggest tile represents all memory needed for six time steps computation, where each tile below is the result of calculating the tile above, when the light blue elements shows the required elements for computation. The red elements is the border region considering the next step. The gray elements represents the unused data for the next time step computation. (Color figure online)

The number of time steps added to the blocks size must be multiple of the global memory bandwidth, in order to achieve data coalescence. In the computation stage, all data present at the shared memory is related to thread disparity at the final of the block execution. For solving this all step time calculations have to be multiple of 32, which corresponds to the warp size.

GPU's are limited by the number of threads per block and a size of shared memory. Since the memory access performs in different ways according to the GPU model, it may be relevant to choose the tile size according with the stencil size, and the hardware limitations, such as shared memory size. It is also

important to calculate the number of elements to be calculated according to the multiple of the number of threads in a warp. Copies and previous time steps need to be executed at a different number of elements and not necessarily as a multiple of the number of threads. It may be challenging to fit the best tile size and time step size in order to maximize computation time [15].

5 Results

To evaluate the time-space blocking, we ran several experiments in a Nvidia Geforce 1080Ti GPU. The machine host had 32 Gb of RAM and it is powered by an Intel Xeon E5-2620 CPU (8 cores). In our experiments, we variate block size, domain size and the number of time steps computed in a single kernel call, but numerical discretization and initial conditions remain the same. We used a time step of 0.05 ms and our spatial resolution was equal in both axes ($\Delta x = \Delta y$). The initial condition was given by:

$$U_{ij}^0 = 3.0 \quad \text{for } x_k \leq 0.05 * (\textit{domain size } x) \tag{7}$$

$$v_{ij}^0 = 0.5 \quad \text{for all } x_k \ . \tag{8}$$

In our experiments we simulated a total of one second ($1s$); therefore, the time domain discretization generates $20,000$ time steps.

Figure 8a depicts the amount of shared memory demanded by the different configurations we evaluate. The memory was calculated according to Eq. 6 and, as we are using single precision variables, it was multiplied by 4 bytes. The figure shows that, due to our GPU limitation of 48 Kb per thread block, not all configurations could be tested. Figure 8b shows all configurations that fits on our GPU. We discarded all experiments that require more than 48 Kb per block.

We measured execution on the GPU time using *cudaEventRecord*. We are no interested in CPU time; all our computation is performed on GPU. The U and v fields are initialized by a kernel and stored directly on GPU global memory. Figure 9 shows the performance of space-time blocking for the selected experiments. As the number of time steps computed per kernel increases, also the computation time increases. This happens due to the escalation of share memory required to store the variables. By raising the amount of share memory needed by each kernel, we reduce the number of concurrent kernels. The maximum amount of shared memory per multiprocessor, in our GPU, is 64 Kb.

To see how space-time blocking (STB) performs compared to classic shared memory implementation (CSM), we will look more carefully at our faster experiments (Fig. 10). We ran the same simulations with CSM, varying block sizes, and we chose the faster configuration to compare it with STB. The results are presented in Fig. 10. Table 1 depicts time saved by STB in comparison with CSM.

We also evaluate the coherence of our simulation outputs by simulates a spiral wave. We ran a simulation for $20,000$ time steps (1 second of cardiac activity). As in our experiments the initial condition was defined by Eqs. 7 and 8. The spiral wave was generated at time step $8,000$ by reset fields U and v to reset values in the bottom half of the domain. Figure 11 shows some frames of a spiral wave simulation with our GPU space-time blocking implementation.

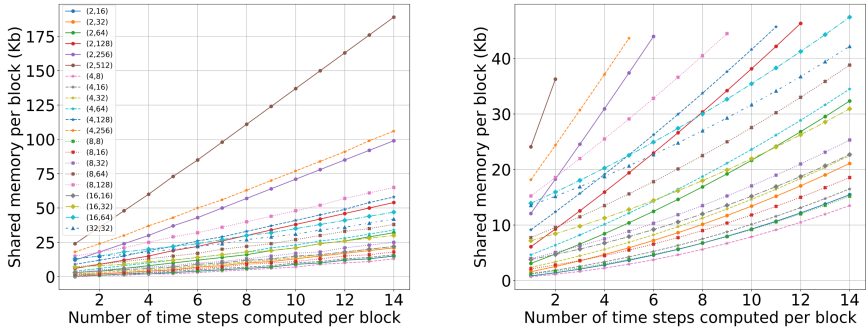

Fig. 8. The figure exhibits the amount of shared memory required by different sets of block size and number of time steps computed per kernel call. Figure (a) depicts all considered configurations, while figure (b) shows only the ones that fit the GPU's shared memory limitation.

Table 1. Best simulation times comparison. All simulations did run 20,000 numerical time steps.

Domain size	Block size for STB	CSM elapsed elapsed time (s)	Time saved elapsed time (s)	Space-time blocking (%)
1024^2	16×32	1.074	0.906	18.543%
2048^2	16×32	4.149	3.673	12.959%
4096^2	32×32	17.379	15.655	11.012%
8192^2	32×32	70.754	64.568	9.580%

6 Related Work

Over the last decade, GPU computational power increased in such a way that they may be used in place of supercomputers for specific scientific computational problems [8,27]. Cardiac electrical dynamics simulations is no exception to this trend [6,19,31].

Several authors have tested GPU performance previously for different cardiac tissue models. They evaluated not only the performance of adapted CPU implementations for GPUs [13,20,34], but also different parallel strategies [1,9,21,26,30,36] and numerical methods [4,35].

Bartocci et al. [1] evaluated GPU implementations for cardiac tissue models with different complexities. They tried two different strategies to accelerate the PDE (Partial Differential Equation) solution through memory optimization: one using shared memory and the other using texture memory. Running on a Tesla C2070, their simulation with a 2D mesh composed of 2^{22} cells took almost $10s$ using texture memory and just over $15s$ using shared memory.

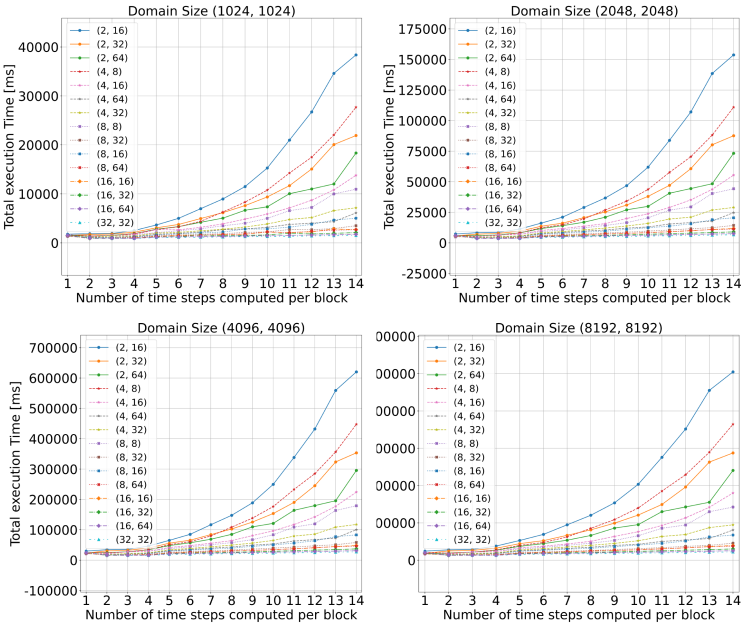

Fig. 9. Elapsed Time for different domain sizes, block sizes and number of time steps solved per kernel call. The distinct color lines means different block sizes.

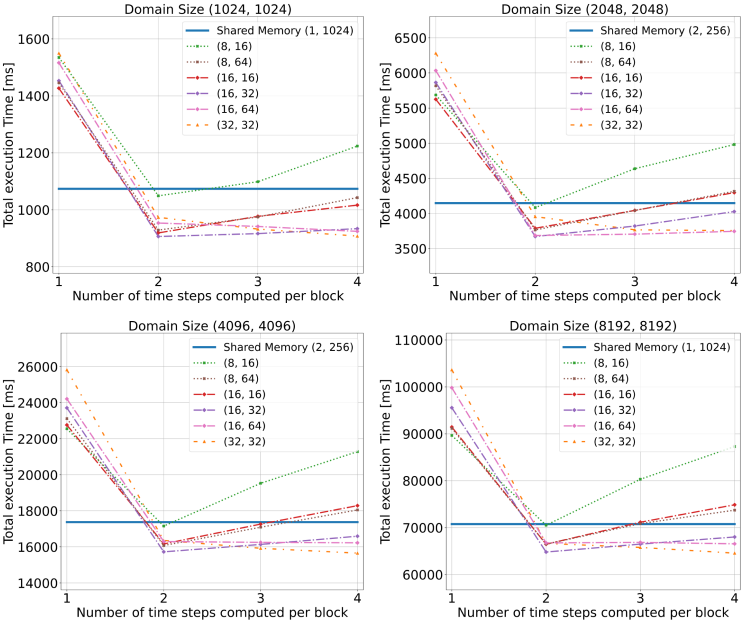

Fig. 10. Elapsed Time for different domain sizes, block sizes and number of time steps solved per kernel call. The distinct color lines means different block sizes.

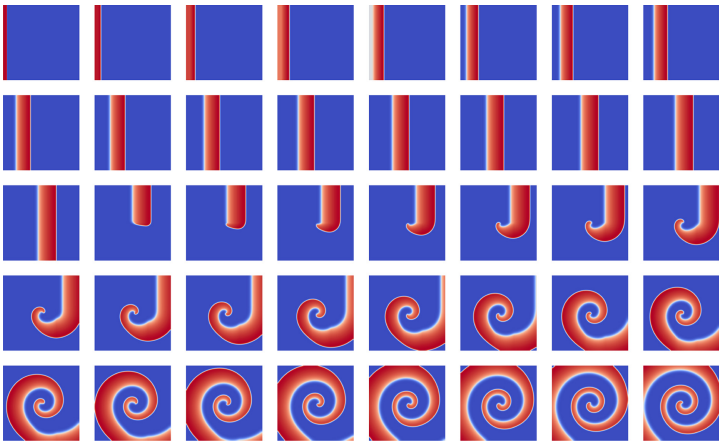

Fig. 11. One second of electrical wave propagation through cardiac tissue.

Nimmagadda [26] explored a linear data structure in GPU simulations using a 2D bidomain model of cardiac electrophysiology. The authors ordered their data to enable WARP coalesced memory access. Using a Tesla C1060, they reported a computation time around $1686s$ to calculate $10,000$ time steps in a $256 \times 256 \times 256$ 3D mesh. The authors also tried a multi-GPU approach and reported an acceleration for the PDE solution of around $14\times$ compared with their single-GPU approach.

Xia et al. [36] used linear structures to accelerate memory access when solving the diffusion term of Eq. 1. The authors' simulations performed $120,000$ time steps in a $317 \times 204 \times 109$ mesh, but only approximately 1 million are valid cell nodes (they omitted empty cells to avoid GPU performance degradation). In this case, they reported a computation time a little over $500s$ to calculate the solution using FDM and double-precision variables (this $500s$ time took into account only the PDE solution).

More recently, Kaboudian et al. (2019) [17] proposed a WebGL based approach to perform cardiac electrophysiology simulations on GPUs, using FDM and single precision. The author compares spiral waves simulated with their WebGL approach with spiral waves from experimental data. For a simulation with a 3-variable model in a 2D domain with 512^2 points, they reported that is possible to run up to $38,000$ time steps per second in a Titan X with Pascal architecture. This means they can process up to 9.96×10^9 domain points (cells) per second. While their work present a good real time visualization result, they are constrained to small domains.

Vasconcellos et al. (2020) [33] proposed a strategy to accelerate the computation of the diffusion term (Eq. 1) through a data-structure and memory access pattern designed to maximize coalescent memory transactions and minimize branch divergence. The authors achieved results approximately 1.4 times faster than the share memory approach proposed by Micikevicius (2009) [23].

They also combine this data structure with a designed communication strategy to take advantage in the case of simulations in multi-GPU platforms. We demonstrate that in the multi-GPU approach performs, simulations in 3D tissue can be just 4× slower than real time.

7 Conclusion

We have shown that with our proposed memory model approach we can reduce close to 20% of the simulation time compared to the classical performance of simulated wave propagation through cardiac tissues using GPU architectures with the karma model. The main feature of our solution is the fact that it copies and calculates multiple time steps through a single copy to the GPU shared memory. Although we have applied directly to cardiac tissue simulation, we believe that many other wave propagation simulations can also benefit.

Kaboudian et al. (2019) [17] reported 38000 time steps per second, corresponding to 9.96×10^9 domain points (cells) per second in a 512^2 2D size domain simulation. Our solution performs 22075 time steps per second, corresponding to 23.14×10^9 domain points in a 1024^2 2D size domain simulation.

We have not achieved yet real-time results, but we believe that in near future it will be possible to increase the performance through machine learning strategies capable of building multi-resolution spaces, according with the wave behavior and tissues characteristics. Improvements on the unified memory of the GPU may also be relevant for future works on the field.

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: Proceedings of the 9th International Conference on Computational Methods in Systems Biology, CMSB 2011, pp. 103–112. ACM, New York (2011). https://doi.org/10.1145/2037509.2037525
2. Beeler, G.W., Reuter, H.: Reconstruction of the action potential of ventricular myocardial fibres. J. Physiol. **268**(1), 177–210 (1977). https://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. J. Theor. Biol. **253**(3), 544–560 (2008). https://www.sciencedirect.com/science/article/pii/S0022519308001690
4. Campos, J.O., Oliveira, R.S., dos Santos, R.W., Rocha, B.M.: Lattice boltzmann method for parallel simulations of cardiac electrophysiology using GPUs. J. Comput. Appl. Math. **295**, 70–82 (2016). https://www.sciencedirect.com/science/article/pii/S0377042715000692
5. Chai, J., et al.: Towards simulation of subcellular calcium dynamics at nanometre resolution. Int. J. High Perform. Comput. Appl. **29**(1), 51–63 (2015). https://doi.org/10.1177/1094342013514465
6. Clayton, R.H., et al.: Models of cardiac tissue electrophysiology: progress, challenges and open questions. Prog. Biophys. Mol. Biol. **104**(1–3), 22–48 (2011). https://www.sciencedirect.com/science/article/pii/S0079610710000362

7. Clayton, R., Panfilov, A.: A guide to modelling cardiac electrical activity in anatomically detailed ventricles. Prog. Biophys. Mol. Biol. **96**(1–3), 19–43 (2008). https://www.sciencedirect.com/science/article/pii/S0079610707000454
8. Dematté, L., Prandi, D.: GPU computing for systems biology. Briefings Bioinf. **11**(3), 323–333 (2010). https://doi.org/10.1093/bib/bbq006
9. Esmaili, E., Akoglu, A., Ditzler, G., Hariri, S., Moukabary, T., Szep, J.: Autonomic management of 3d cardiac simulations. In: 2017 International Conference on Cloud and Autonomic Computing (ICCAC), pp. 1–9, September 2017
10. Fenton, F., Karma, A.: Vortex dynamics in three-dimensional continuous myocardium with fiber rotation: filament instability and fibrillation. Chaos **8**(1), 20–47 (1998). https://scitation.aip.org/content/aip/journal/chaos/8/1/10.1063/1.166311
11. Fenton, F.H., Cherry, E.M.: Models of cardiac cell. Scholarpedia **3**(8), 1868 (2008). https://www.scholarpedia.org/
12. 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, WHPCF 2014, pp. 1–8. IEEE Press, Piscataway (2014). https://doi.org/10.1109/WHPCF.2014.10
13. Higham, J., Aslanidi, O., Zhang, H.: Large speed increase using novel GPU based algorithms to simulate cardiac excitation waves in 3d rabbit ventricles. In: 2011 Computing in Cardiology, pp. 9–12, September 2011
14. Hoffman, J.D., Frankel, S.: Numerical Methods for Engineers and Scientists. CRC Press, New York (2001)
15. 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, ICS 2012, pp. 311–320. ACM, New York (2012). https://doi.org/10.1145/2304576.2304619
16. Iyer, V., Mazhari, R., Winslow, R.L.: A computational model of the human left-ventricular epicardial myocyte. Biophys. J. **87**(3), 1507–1525 (2004). https://www.sciencedirect.com/science/article/pii/S0006349504736346
17. Kaboudian, A., Cherry, E.M., Fenton, F.H.: Real-time interactive simulations of large-scale systems on personal computers and cell phones: toward patient-specific heart modeling and other applications. Sci. Adv. **5**(3), eaav6019 (2019). https://advances.sciencemag.org/content/5/3/eaav6019
18. Karma, A.: Spiral breakup in model equations of action potential propagation in cardiac tissue. Phys. Rev. Lett. **71**, 1103–1106 (1993). https://doi.org/10.1103/PhysRevLett.71.1103
19. Lopez-Perez, A., Sebastin, R., Ferrero, J.M.: Three-dimensional cardiac computational modelling: methods, features and applications. Biomed. Eng. OnLine **14**(1), 1–31 (2015). https://doi.org/10.1186/s12938-015-0033-5
20. Mena, A., Rodriguez, J.F.: Using graphic processor units for the study of electric propagation in realistic heart models. In: 2012 Computing in Cardiology, pp. 37–40, September 2012
21. Mena, A., Ferrero, J.M., Matas, J.F.R.: GPU accelerated solver for nonlinear reaction-diffusion systems. Application to the electrophysiology problem. Comput. Phys. Commun. **196**, 280–289 (2015). https://doi.org/10.1016/j.cpc.2015.06.018. https://www.sciencedirect.com/science/article/pii/S0010465515002635
22. Michéa, D., Komatitsch, D.: Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards. Geophys. J. Int. **182**(1), 389–402 (2010). https://doi.org/10.1111/j.1365-246X.2010.04616.x

23. 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)
24. Mirin, A.A., et al.: Toward real-time modeling of human heart ventricles at cellular resolution: simulation of drug-induced arrhythmias. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, pp. 2:1–2:11. IEEE Computer Society Press, Los Alamitos (2012). https://dl.acm.org/citation.cfm?id=2388996.2388999
25. Niederer, S., Mitchell, L., Smith, N., Plank, G.: Simulating human cardiac electrophysiology on clinical time-scales. Front. Physiol. **2**, 14 (2011). https://www.frontiersin.org/articles/10.3389/fphys.2011.00014/full
26. Nimmagadda, V.K., Akoglu, A., Hariri, S., Moukabrav, T.: Cardiac simulation on multi-GPU platform. J. Supercomputing **59**(3), 1360–1378 (2012). https://doi.org/10.1007/s11227-010-0540-x
27. Nobile, M.S., Cazzaniga, P., Tangherloni, A., Besozzi, D.: Graphics processing units in bioinformatics, computational biology and systems biology. Briefings Bioinf. **18**(5), 870–885 (2017). https://doi.org/10.1093/bib/bbw058
28. O'Hara, T., Virág, L., Varró, A., Rudy, Y.: Simulation of the undiseased human cardiac ventricular action potential: Model formulation and experimental validation. PLOS Comput. Biol. **7**(5), 1–29 (2011). https://doi.org/10.1371/journal.pcbi.1002061
29. Richards, D.F., et al.: Towards real-time simulation of cardiac electrophysiology in a human heart at high resolution. Comput. Methods Biomech. Biomed. Eng. **16**(7), 802–805 (2013). https://doi.org/10.1080/10255842.2013.795556
30. Rocha, B.M., et al.: Accelerating cardiac excitation spread simulations using graphics processing units. Concurrency Comput. Pract. Experience **23**(7), 708–720 (2011)
31. Szafaryn, L.G., Skadron, K., Saucerman, J.J.: Experiences accelerating matlab systems biology applications. In: Proceedings of the Workshop on Biomedicine in Computing: Systems, Architectures, and Circuits, pp. 1–4 (2009)
32. Ten Tusscher, K.H.W.J., Panfilov, A.V.: Alternans and spiral breakup in a human ventricular tissue model. Am. J. Physiol. Heart Circulatory Physiol. **291**(3), H1088–H1100 (2006). https://ajpheart.physiology.org/content/291/3/H1088
33. 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 Comput. Pract. Experience **32**(5), e5528 (2020). https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5528
34. Vigueras, G., Roy, I., Cookson, A., Lee, J., Smith, N., Nordsletten, D.: Toward GPGPU accelerated human electromechanical cardiac simulations. Int. J. Numer. Methods Biomed. Eng. **30**(1), 117–134 (2014). https://doi.org/10.1002/cnm.2593
35. Vincent, K., et al.: High-order finite element methods for cardiac monodomain simulations. Front. Physiol. **6**, 217 (2015). https://www.frontiersin.org/article/10.3389/fphys.2015.00217
36. Xia, Y., Wang, K., Zhang, H.: Parallel optimization of 3d cardiac electrophysiological model using GPU. Comput. Math. Methods Med. **2015**, 1–10 (2015)