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 cause by a propagation of an electrical wave through cardiac tissue. Although, some times this wave propagation is disrupted, causing an abnormal behavior of the heart, resulting problems like ventricular tachycardia and fibrillation and maybe resulting in death. This kind of cardiac problems can be simulated using mathematical models that simulates the electrical wave propagation through cardiac tissue.

Eletrophysiology models for cardiac tissue have a high computational demand, mainly for three dimensional simulations. Some author address this problems by using super computers, but many others see GPUs as a solution at lower cost. Real time simulations, i.e., simulate a numerical time step

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:

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

* Supported by organization x.

The first term on the right side represents the diffusion component, and the second term on the right side represents the reaction component, where I_{ion} represents 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 [**?**]. The value of D affects the speed of electrical wave propagation in tissue [**?**,**?**].

The reaction term is modeled by a system of nonlinear ordinary differential equations of the form

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

the exact form depends on the level of complexity of the electrophysiology model. For each additional variable $\mathbf{y_j}$, $\mathbf{F_j}(\mathbf{y_j}, V, t)$ is a nonlinear function. Clayton et al. provide a good review about cardiac electrical activity modelling[**?**].

There are many mathematical models that describe cellular cardiac electrophysiology [**?**,**?**,**?**,**?**,**?**,**?**]. 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}; 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 ([**?**]). 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 proposes of Karma model is simulates the behavior of spiral waves, like the ones associated with cardiac arrhythmias.

2.2 Numerical solution

Finite difference methods (FDM) are a common numerical methods used to obtain a numerical solution for many different numerical problems [**?**]. 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:

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

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 [**?**], can works as base for some three dimensional propagation problems [**?**,**?**,**?**,**?**,**?**].

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 [**?**,**?**,**?**]. For each new time step, the GPU computes thousands of points in parallel, solving Equation 5 on each point in 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 needed for the spatial discretization [**?**]. This compromises GPU performance due to the greater elapsed time to access all neighbors.

The stencil in Figure **??** represents the required data to compute the value at the next time step $t + 1$ at a given point p. When we are computing float

single precision values, its neighbors in the x-direction are offset by 4 bytes to the left or right, and neighbors in the y-direction, a neighbor is 40 bytes away. Clearly, in this case the goal of accessing only nearby memory locations cannot be achieved.

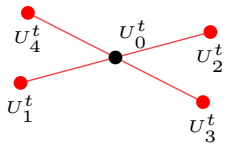

Fig. 1. 2D stencil representing data required for calculating the value at the next numerical time step $t + 1$ at each domain point U_0 using a standard second-order FDM.

4 2D space-time blocking on GPU

A simple 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 [**?**].

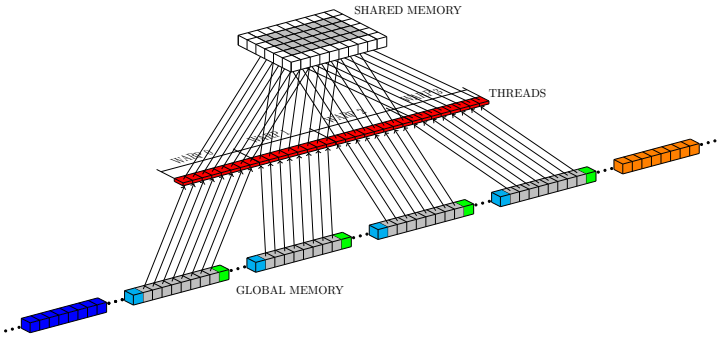

Fig. 2. Access pattern for core data on global memory.

The use of shared memory can help avoiding loose time with redundant global memory access. Although, CUDA thread blocks do not have any kind of data synchronization, so, on every time step in our FDM, we need to update global memory with the new values computed by our thread blocks. For all time step computations we need to read an write data in global memory.

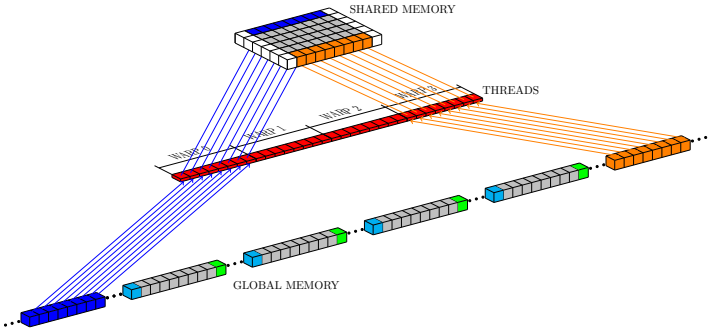

Fig. 3. Access pattern for y neighborhoods.

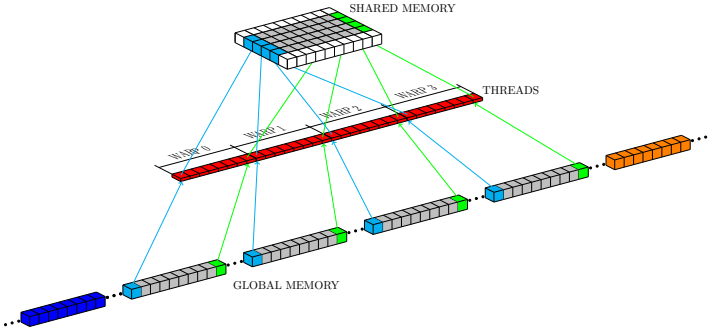

Fig. 4. Access pattern for x neighborhoods.

We can reduce global memory transactions applying an approach called space-time blocking. The global memory need to be updated, only every t time steps, the amount of data, necessary to t time steps calculation, are copied from global memory to shared memory, in the first time step, in the t-step, the result is write 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.

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

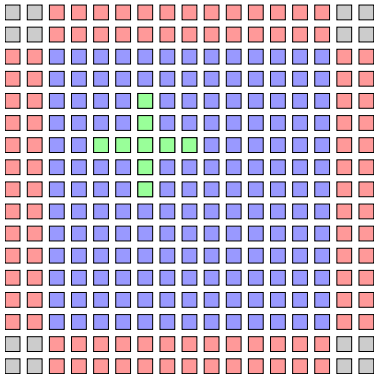

Fig. 5. Shared memory tile, the elements in green color represent the neighbors needed to the middle stencil computation, the red elements are the halo region and blue elements are the elements to compute in this block

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.

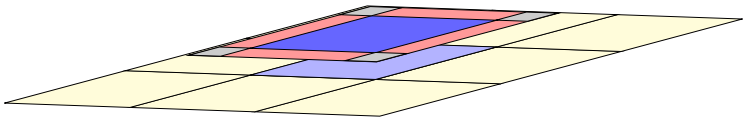

Fig. 6. Global memory blocks division in yellow grid, with tile copied to shared memory, when, the elements to be computed as show in blue color, the halo region in red, and unused elements in gray

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

Between each time step calculation all threads have to be, synchronized and the old data, have to be updated to the new data, therefore, the size of shared memory have to be duplicated from the equation 6. The new data do not be stored in registers, because, the threads have to calculate another element before the synchronization and data copy.

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.

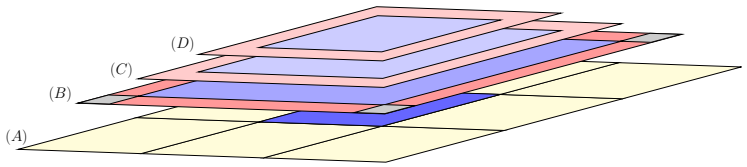

Fig. 7. Space time blocking example, of, shared memory copies in each time step, the A element shows the global memory, B is the shared memory tile required for 3 time steps, C and D represent the elements that have to be computed for the three time steps calculation.

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 memory copies, have the same problem, and 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 [**?**].

Seconds of an electrical wave propagation, through cardiac tissue, needs thousands of time steps, in a acceptable time discretization. Taking advantage of it, an approach to decrease computing time, is negotiating computation and accesses of shared memory for global memory accesses [**?**]. To do this, in each block, all necessary data for t_n time steps, is transferred from global memory to shared memory. 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.

5 Results

Three approaches of the stencil computation, is compared in two tests, for k equal to 2 and k equal to 4, in the two tests, the time step size t_n is increased and this calculation time is analyzed. Global means the simplest approach, all memory access are made in global memory. Observing the two charts in figure **??** and figure **??**, it is possible to check, that shared memory method, provides a better performance on bigger stencils. The shared memory method, copies, the actual time step tile to the shared memory, in every time step.

The shape of the curves, indicate, that global and shared memory methods have a linear behavior, because, the size of stencil have most influences to the run time. In the time space blocking implementation, 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.