J Supercomput (2012) 59:1360–1378
DOI 10.1007/s11227-010-0540-x

Cardiac simulation on multi-GPU platform

Venkata Krishna Nimmagadda · Ali Akoglu · Salim Hariri · Talal Moukabary

Published online: 5 January 2011

Abstract The cardiac bidomain model is a popular approach to study electrical behavior of tissues and simulate interactions between the cells by solving partial differential equations. The iterative and data parallel model is an ideal match for the parallel architecture of Graphic Processing Units (GPUs). In this study, we evaluate the effectiveness of architecture-specific optimizations and fine grained parallelization strategies, completely port the model to GPU, and evaluate the performance of single-GPU and multi-GPU implementations. Simulating one action potential duration (350 msec real time) for a $256 \times 256 \times 256$ tissue takes 453 hours on a high-end general purpose processor, while it takes 664 seconds on a four-GPU based system including the communication and data transfer overhead. This drastic improvement (a factor of $2460\times$) will allow clinicians to extend the time-scale of simulations from milliseconds to seconds and minutes; and evaluate hypotheses in a shorter amount of time that was not feasible previously.

Keywords Bidomain model · Cardiac tissue · Graphics processing unit · Multi-GPU

1 Introduction

The contractions of the human heart are initiated by electric wave propagation in cardiac tissues. The electrical stimulus is generated by a region in the heart called the SinoAtrial (SA) node. Then the stimulus is propagated across the heart causing regular contractions. The instabilities in excitation propagation may cause uncoordinated contraction of cardiac muscle which leads to cardiac arrhythmias. Ventricular

V.K. Nimmagadda · A. Akoglu (✉) · S. Hariri
Department of Electrical and Computer Engineering, University of Arizona, Tucson, AZ 85721, USA
e-mail: akoglu@ece.arizona.edu

T. Moukabary
Sarver Heart Center, College of Medicine, University of Arizona, Tucson, AZ 85721, USA

fibrillation is an arrhythmia which is responsible for many cardiac deaths. 3D simulations of electric wave propagation in cardiac tissue help medical researchers to understand electrical instabilities and excitation dynamics in cardiac tissues [1]. The cardiac bidomain model is a popular multidimensional cable model [2] to study the electrical behavior of cardiac tissues. A realistic tissue simulation typically requires solving a large number of unknowns at heart cells (nodes) separated by small spatial as well as temporal steps. The computational resources required for 3D Bidomain simulation of even a $64 \times 64 \times 64$ grid of cardiac tissue cells for 1 second real time is immense and is considered as large scale simulation. Bidomain simulations define the heart as a continuous system comprising of two overlapping regions and involve solving coupled nonlinear partial differential equations. Every cell in the grid is associated with an unknown and the Bidomain model solves the nonlinear differential equations for each node and calculates the trans-membrane voltage. General technique is first to convert the continuous equations into ordinary differential equations (ODEs) and partial differential equations (PDEs) and then solve ODEs followed by PDEs to derive the current state of each cell for each time step of the cardiac simulation. This iterative and inherently data parallel nature of the software architecture makes it an ideal match for the fine grained parallel hardware architecture of the NVIDIA Graphic Processing Units (GPU).

It is well understood that GPUs are several times faster than traditional CPUs as more transistors are dedicated to computing rather than to data cache or flow control [3]. GPUs allow programmers to exploit fine grained parallelism in the application with thousands of concurrent threads which is not feasible with cluster based high performance computing systems. Partitioning the cardiac simulation with millions of cells into cell level fine grained processing modules and executing these modules as parallel threads allow exploiting the threading power of the GPU. In this study, our objective is to accelerate the simulations of the electrical dynamics in the heart tissue so that near real time visualization capability can be achieved for a wide range of clinical scenarios. For a realistic simulation of cardiac tissue on a multi-GPU platform, the implementation must maintain the excitement propagation through boundary cells. In the context of multi-GPU implementation, boundary cells refer to neighboring cells that have been separated after partitioning the tissue data among the GPUs. In order to address this challenge, we introduce a new approach for simulating cardiac tissue on multiple GPUs. We evaluate the impact of hardware specific workload partitioning and memory optimization strategies on the performance. We then compare performance of the single-GPU and multi-GPU implementations with the serial version running on a general purpose processor. Results show that as the size of the 3D grid space increases, the speedup curve reaches to saturation with up to $657\times$ and $2538\times$ on single-GPU and multi-GPU implementations, respectively, for a single time step simulation compared with the general purpose processor. This performance improvement will allow clinicians to extend the time-scale of simulations from milliseconds to seconds and minutes; and evaluate hypotheses in a shorter amount of time that was not feasible previously. In Sect. 2, we give a brief overview of the GPU architecture from the processor, memory and programming model perspectives. In Sect. 3, we present the mathematical foundations of the cardiac simulation and describe how 3D heart is modeled through ODEs and PDEs. In Sect. 4, we describe our parallelization approach on single-GPU and multi-GPU platforms and then present performance analysis with respect to serial version in Sect. 5. Finally, Sect. 6 presents our conclusion.

2 GPU architecture

2.1 Processor and memory architecture

Our target GPU system houses four NVIDIA Tesla C1060 cards. Each card consists of 30 multiprocessors. Each multiprocessor contains 8 streaming processors for a total of 240 cores per device. Each multiprocessor can run 1024 threads or 8 blocks of threads, whichever is less, at an instant of time. A hierarchy of memory architecture (Global memory, Texture Memory, Shared Memory, and Local Registers) is available on the GPU for a programmer to utilize. To optimize the memory access time, data partitioning across various memories at thread level is an important consideration. GPUs are proven to give high speedups with inherently data parallel applications if the thread level data partitioning is efficient.

2.2 Programming model

The Compute Unified Device Architecture (CUDA), a C like programming environment introduced by NVIDIA, has improved the programmability of GPUs for general purpose applications. CUDA programming environment hides complexity of managing the computational cores and allows programmer to define a workload in terms of number of blocks and number of threads per block. However, understanding the low level details of the hardware architecture and memory access mechanisms is essential to be able to harness the computation power of the GPU. CUDA programming model consists of a grid of blocks where each block contains a specified number of threads. Threads execute an identical kernel of code and access the data through the memory hierarchy. When launching the kernel on the device, the programmer specifies the blocks per grid and threads per block based on the memory footprint of the kernel. In order to gain optimal performance out of the Tesla architecture, the user must organize a program to maximize thread throughput, while managing the shared memory, registers, and global memory usage. CUDA allows sharing state between threads within the same thread-block through the shared memory on a multiprocessor. Each thread block is executed by running groups of 32 threads, known as warps. Individual threads run all of the code on the kernel, which may be as small as only a function, or may contain the entire program. The scope of sharing between threads within the same warp or different warps is the same shared memory space as long as they are in the same thread-block. If warps are located in different thread blocks, then they cannot communicate with each other through the shared memory. The global memory can be utilized to share information among all threads on all multiprocessors; however, there is no global synchronization method or function in CUDA. To ensure synchronization occurs, a kernel must finish its execution followed by the launch of a new kernel. Each thread within the multiprocessor can access the global memory directly with high latency. The other two memory subsystems, texture and constant caches, can also be utilized by all the threads within the GPU. However, shared memory is an individual and protected region in every multiprocessor that is only available to its own threads. It cannot be accessed by the threads of other multiprocessors. The threads can only execute the identical kernel of code. Once launched, the threads transfer the data stream primarily with the memory subsystems. Even though they cannot necessarily communicate with each other, they can share the same data within the shared memory on the multiprocessor.

3 Cardiac bidomain model

As the name suggests, bidomain model consists of two overlapped domains which represent the region within the cardiac cell (intracellular) and region surrounding the cardiac cells (extracellular). Capacitive membrane separates the two layers through which trans-membrane current is conducted. The trans-membrane current is made up of two components, ionic current and extracellular current. Extracellular current depends on the variation of potential gradient across different points of interest in the tissue. Ionic current depends on the difference of ion concentration (Ca, Na, and K) across the cellular membrane and the conductivity of the respective ion channels across that membrane. Finite difference equations can be applied to determine the potential value in each cell of the 3D heart model. Finite difference method converts the continuous system of cells to a manageable grid of cells in order to facilitate computer simulation. The properties of cells, e.g., conductivity, are precomputed by volume averaging across many cells.

In [4], Paulius discussed an efficient and scalable technique for 3D finite difference computation on GPUs which can be applied to solve coupled differential equations of bidomain model. In our approach, we first employ the operator splitting technique [5] and divide bidomain equations of cardiac cell into ordinary differential equations (ODEs) and partial differential equations (PDEs). ODEs are solved by using Euler forward method [6] to derive the ionic current as it is a widely accepted model in terms of its accuracy and is highly compute intensive with independent calculations for each cell. PDEs are solved by using Jacobi iterative Technique [7] to derive the capacitive current as it offers SIMD level parallelism for the data intensive cardiac tissue simulations and poses as an ideal match for multi-GPU implementation.

Cardiac tissue is considered as a grid of cells. Voltage level for each cell is a function of the voltage level of each neighboring cell calculated from the previous time step.

$$V_0 = f(V_1 + V_2 + \cdots + V_6)$$

As illustrated in Fig. 1, a voltage applied to one cell propagates through the grid as an electrical wave. The bidomain model which is equivalent to the actual electric wave propagation in heart is used to simulate such wave.

3.1 Action potential duration

When a stimulus is applied, a ventricular cell gets excited and generates action potential as shown in Fig. 2. The stimulus from SinoAtrial Node propagates across the atria

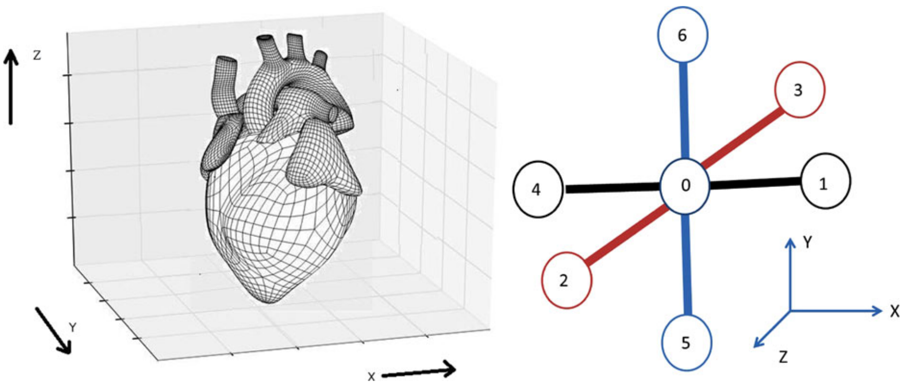
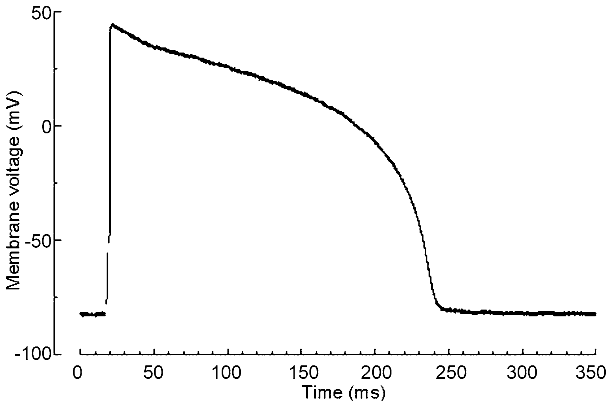


Fig. 1 On *left*—heart constructed out of grid of cells. On *right*—cell 1–6 which interact with cell 0 for excitation propagation

Fig. 2 APD Curve of a cardiac (ventricular) cell (350 ms extension)



and the atrioventricular node to reach and propagate in the ventricular region. When a ventricular cell is excited, the voltage across its cell membrane increases rapidly for a short time period and starts discharging like a capacitor. The cell membrane acts as a capacitive element which charges instantly and discharges slowly. The shape of the action potential duration (APD) curve is achieved by the combined effect of ionic currents and capacitive currents across the cell membrane. A stimulus generated by a sinoatrial cell excites its surrounding cells and starts the propagation of the excitation as wave propagation.

Excitation potential of each cell depends on the excitation potential of surrounding cells and the ionic currents of that cell. The basic procedure of calculating cell potential contributed by ionic currents is adopted from TNNP model [8] proposed by Ten Tusscher et al. The intracellular communication of excitation is implemented by calculating discretized Laplacian of a continuous system model defined by partial differential equation. To develop an efficient algorithm for heart simulation, the sequential algorithm, which simulates the propagation of signals from one cell to neighboring cells iteratively, needs to be reconstructed so that the program architecture overlaps with the GPU's SIMD model hardware architecture.

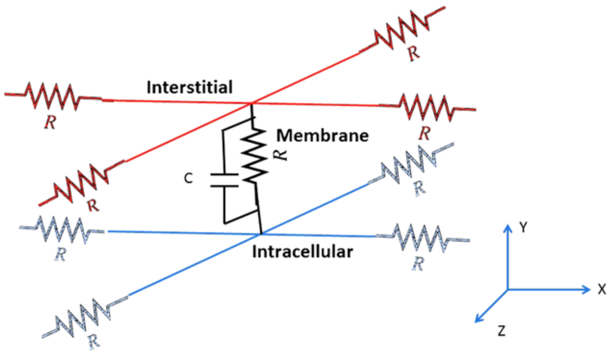


Fig. 3 The equivalent electric circuit of cardiac cell based on Bidomain model.

3.2 Bidomain equations

The bidomain model consists of cardiac tissue and conducting surface (bath) represented as overlapped regions (Fig. 3). The space inside each cell is termed as the intracellular region and the space surrounding each cell is termed as the interstitial region. Both regions are bounded by the extracellular region.

Let ϕ_i, ϕ_e, and ϕ_o represent intracellular, extracellular, and interstitial potentials, respectively. Bidomain model is governed by following (1) through (3) where C_m is the membrane capacitance, V_m is the voltage at node of interest, ϕ_o is the interstitial potential, I_{ion} is the ionic current calculated from the Cell Model, and I_{so} is the stimulus current. Here, σ represents conductivities of the grid with the first subscript $(i,\ e,\ o)$ denoting the intracellular, extracellular, or interstitial region and the second subscript representing the direction in (x, y, z) coordinates. These differential equations are converted to finite difference equations to facilitate computer simulations.

$$\frac{\partial V_m}{\partial t} = -\frac{1}{C_m}\left[I_{so} + \left(\sigma_{ox}\frac{\partial^2\phi_o}{\partial x^2} + \sigma_{oy}\frac{\partial^2\phi_o}{\partial y^2} + \sigma_{oz}\frac{\partial^2\phi_o}{\partial z^2}\right) + I_{\text{ion}}\right] \tag{1}$$

$$\sigma_{ix}\frac{\partial^2 V_m}{\partial x^2} + \sigma_{iy}\frac{\partial^2 V_m}{\partial y^2} + \sigma_{iz}\frac{\partial^2 V_m}{\partial z^2} + (\sigma_{ix}+\sigma_{ox})\frac{\partial^2\phi_o}{\partial x^2} + (\sigma_{iy}+\sigma_{oy})\frac{\partial^2\phi_o}{\partial y^2} + (\sigma_{iz}+\sigma_{oz})\frac{\partial^2\phi_o}{\partial z^2} = -(I_{si}+I_{so}) \tag{2}$$

$$\sigma_{ex}\frac{\partial^2\phi_e}{\partial x^2} + \sigma_{ey}\frac{\partial^2\phi_e}{\partial y^2} + \sigma_{ez}\frac{\partial^2\phi_e}{\partial_z^2} = 0 \tag{3}$$

Using the operator splitting technique, (1) can be split into the following two equations:

$$C_m\frac{\partial V_m}{\partial t} = -(I_{\text{ion}} + I_{\text{stim}}) \tag{4}$$

$$\frac{\partial V_m}{\partial t} = -\frac{1}{C_m}\left[\sigma_{ox}\frac{\partial^2\phi_o}{\partial x^2} + \sigma_{oy}\frac{\partial^2\phi_o}{\partial y^2} + \sigma_{oz}\frac{\partial^2\phi_o}{\partial z^2}\right] \tag{5}$$

3.2.1 Solving ordinary differential equations

Many mathematical models [8–13] of cardiac cells have been developed to model the ionic currents in cardiac cell. The parameters of these models can be altered by medical researchers to study the cardiac cell behavior under different conditions. These models are useful for calculation of trans-membrane ionic currents in simulation. In the TNPP cell model [8] proposed by Ten Tusscher et al., action potential (or trans-membrane voltage V_m) generation in an isolated single cell is calculated with (6), where C_m represents the trans-membrane capacitance of a cell, V_m represents the action potential (trans-membrane voltage) generated at cell of interest, I_{ion} is the current contributed by various ionic currents, S represents the state of ion concentrations, and I_{stim} is the current applied as a stimulus to the cell either by surrounding cells or by injection in simulation. In this study, we are mainly interested in ventricular cells which are passive and do not generate stimulus by themselves.

$$C_m \frac{\partial V_m}{\partial t} = -\big(I_{\text{ion}}(V_m, s) + I_{\text{stim}}\big) \tag{6}$$

$$\frac{\partial s}{\partial t} = F(V_m, s) \tag{7}$$

Equation (6) is solved for intermediate value of V_m referred to as V_{mO} which is used as initial value for solving coupled PDEs.

3.2.2 Solving coupled partial differential equations

Equations (2) and (5) form a coupled system of partial differential equations which can be decomposed into a set of finite difference equations as shown in (8) and (9). Jacobi Iterative technique is used to solve these coupled partial differential equations.

$$\begin{aligned}
(\phi_o)^n_{i,j,k} &= \frac{Term1 + Term2 + (I_{si} + I_{so})^n_{i,j,k}}{2\left[\sum_{x,y,z} \frac{\sigma_{ix}+\sigma_{ox}}{\Delta x^2}\right]} \\
Term1 &= (\sigma_{ix} + \sigma_{ox}) \frac{(\phi_o)^n_{i+1,j,k} + (\phi_o)^n_{i-1,j,k}}{\Delta x^2} \\
&\quad + (\sigma_{iy} + \sigma_{oy}) \frac{(\phi_o)^n_{i,j+1,k} + (\phi_o)^n_{i,j-1,k}}{\Delta y^2} \\
&\quad + (\sigma_{iz} + \sigma_{oz}) \frac{(\phi_o)^n_{i,j,k+1} + (\phi_o)^n_{i,j,k-1}}{\Delta z^2} \\
Term2 &= \sigma_{ix} \frac{(V_m)^n_{i+1,j,k} + (V_m)^n_{i-1,j,k} - 2(V_m)^n_{i,j,k}}{\Lambda x^2} \\
&\quad + \sigma_{iy} \frac{(V_m)^n_{i,j+1,k} + (V_m)^n_{i,j-1,k} - 2(V_m)^n_{i,j,k}}{\Lambda y^2} \\
&\quad + \sigma_{iz} \frac{(V_m)^n_{i,j,k+1} + (V_m)^n_{i,j,k-1} - 2(V_m)^n_{i,j,k}}{\Lambda z^2}
\end{aligned} \tag{8}$$

$$(V_m)^{n+1} = (V_m)^n - \frac{\Delta t}{C_m}\left\{\sigma_{ox}\frac{(\phi_o)^n_{i+1,j,k} + (\phi_o)^n_{i-1,j,k} - 2(\phi_o)^n_{i,j,k}}{\Lambda x^2}\right.$$
$$+ \sigma_{oy}\frac{(\phi_o)^n_{i,j+1,k} + (\phi_o)^n_{i,j-1,k} - 2(\phi_o)^n_{i,j,k}}{\Lambda y^2}$$
$$\left. + \sigma_{oz}\frac{(\phi_o)^n_{i,j,k+1} + (\phi_o)^n_{i,j,k-1} - 2(\phi_o)^n_{i,j,k}}{\Lambda z^2}\right\} \quad (9)$$

3.2.3 Boundary conditions

At the surface of an isolated cardiac tissue, boundary conditions need to be enforced. Considering cardiac tissue as an isolated electric medium, the current leaving this surface area is forced to be zero based on the Neumann noundary condition. Equation (3) represents the boundary conditions of an isolated cardiac tissue. The corresponding finite difference equation is approximated as shown in (10).

$$(\phi_e)^n_{i,j,k} = \frac{\sigma_{ex}\frac{(\phi_e)^n_{i+1,j,k}+(\phi_e)^n_{i-1,j,k}}{\Delta x^2} + \sigma_{ey}\frac{(\phi_e)^n_{i,j+1,k}+(\phi_e)^n_{i,j-1,k}}{\Delta y^2} + \sigma_{ez}\frac{(\phi_e)^n_{i,j,k+1}+(\phi_e)^n_{i,j,k-1}}{\Delta z^2}}{2\left[\frac{\sigma_{ex}}{\Delta x^2} + \frac{\sigma_{ey}}{\Delta y^2} + \frac{\sigma_{ez}}{\Delta z^2}\right]} \quad (10)$$

3.2.4 Voltage update

In this stage, every cell is updated with the final membrane voltage (V_m) in the current time step. Equation (11) calculates the V_m based on the V_{mP} generated by the governing PDEs (4) and ionic current value generated by the governing ODEs (5).

$$V_m = V_{mP} - \frac{1}{C_m}(I_{\text{ion}} + I_{\text{stim}})\Delta t \quad (11)$$

4 Implementation details

4.1 Sequential version

The source code for the serial implementation in C (baseline) is publicly available at [14]. In this implementation, for each time step, voltage of each cell in the 3D grid space is calculated based on the governing ODEs and PDEs. This process continues until the global sum of squared residues (SSR) is less than particular value (10^{-6} in our case).

As shown in Fig. 2, APD cycle time is 350 mssec and the propagation of the excitement takes 10,000 time steps. Figure 4 shows the execution time of the bidomain model with respect to change in the tissue (grid) size for a single time step on a general purpose processor (GPP) based on Intel I7, 2.4 GHz processor. As the number of

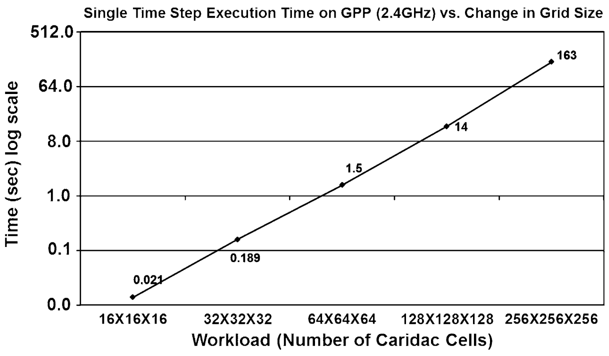


Fig. 4 Cardiac tissue simulation time on GPP based on Bidomain Model with respect to change in workload (based on logarithmic scale base 8) shows linear increase in execution time

cells increase, the execution time increases linearly. Completing the APD cycle time (350 msec) for 256 × 256 × 256 grid based tissue cells takes almost 453 hours in simulation time through serial code.

4.2 Heart simulation on a single GPU

We first approach parallelization of the heart simulation at task level and then explore fine grained parallelism at subtask level. We then evaluate the impact of memory organization on the performance. The following subsections describe our parallelization strategies for each subtask.

4.2.1 Task level parallelism

As part of the parallelization strategies, we partition the execution of the heart simulation into segments, create a kernel for each segment, and launch these kernels one after another while using the global memory as a facilitator for data dependencies between each segment. As shown in Fig. 5, heart simulation involves an initialization phase followed by the ODE (segment 1), PDE (segment 2) and boundary condition (segment 3) phases in a nested loop structure.

Figure 6 illustrates the implementation of the heart simulation with three partitioning strategies. Strategy 1 launches ODE Solver and Boundary Solver stages as a single kernel. Strategy 2 (Fig. 6(b)) launches a separate kernel for each segment of the execution flow (Fig. 5). Strategy 3 explores fine grained parallelism by partitioning the execution of the ODE into multiple kernels as illustrated in Fig. 6(c). For each strategy, Fig. 6 shows the register demand per kernel and the number of threads per multiprocessor launched by each kernel. Here, we note that PDE Solver must be executed in 2 separate Kernels as execution of (8) must be complete before the execution of (9) can start.

For example in strategy 1, threads of Kernel 1 could be grouped as 256 threads in one active block, 128 threads in 2 blocks, or 64 threads in 4 blocks and 32 threads

```
BidomainModel (){
  Initialization of Data Structures()
  for each Timestep in Simulation {
     for each cell in the grid {
     /* I. ODE Solver */
     a. Solve for $V_m$ in ODE in (Eq.6)
     b. Solve for s in ODE in (Eq.7)
    While $V_m$ not converged {
        /* II. PDE Solver */
        a. Solve PDE for $V_m$ dependent on Divergence($\phi_o$) (Eq.8)
        b. Solve PDE for $\phi_o$ dependent on Divergence($V_m$) ( Eq.9)
     } end of while
     /* III. Boundary Condition Solver */
     Solve PDE for Boundary voltage $\phi e$ (Eq.10)
     /* IV. Voltage Update */
     Calculate voltage for each cell in the timestep (Eq.11)
        } /* end of for */
  } /* end of for */
} /* end of program */
```

Fig. 5 Execution flow of the Bidomain Model in four main stages

in 8 blocks. After running each segment with various workloads, we find that having 64 threads per block yields the best performance. When using 32 threads per block, there is only 1 warp per block. According to [8], having less than 2 warps per block is not enough to hide latency associated with memory reads. When utilizing 256 threads per block, we can only have 1 active block per multiprocessor. A new block can be launched only after all threads in a block finish execution [8]. Having a slightly finer distribution of threads, in this case 64, helps in reducing this thread dependency. Having 64 threads per block allows us to achieve our best performance for this strategy as it prevents idling threads and provides the best workload distribution in terms of MP utilization of blocks. For all three strategies, we partition the threads for each multiprocessor into multiple blocks. This allows overlapping between idle blocks and working blocks, hence hiding memory latency [8]. Based on our exhaustive fine tuning efforts for identifying the most suitable number of threads per block with various workloads, we conclude that in order to get optimal performance from the GPU, the programmer must carefully balance the data partitioning among threads, form thread counts that are factors of the workload, have an even workload distribution among all MPs, and avoid idling threads whenever possible. We take the best case for each strategy and compare the performance based on a workload grid space of $256 \times 256 \times 256$ cells in Fig. 7.

Strategy 2 partitions Kernel 1 of Strategy 1 into two Kernels and increases the thread count1. Similarly, Strategy 3 partitions Kernel 1 of Strategy 2 into two Kernels increasing the number of threads per multiprocessor. Execution time follows the same trend resulting with the best performance based on Strategy 3.

4.2.2 Memory coalescing

Designing proper data structure is highly critical for data intensive applications. When the grid size is large, it is not feasible to fit the data (e.g., voltage and con-

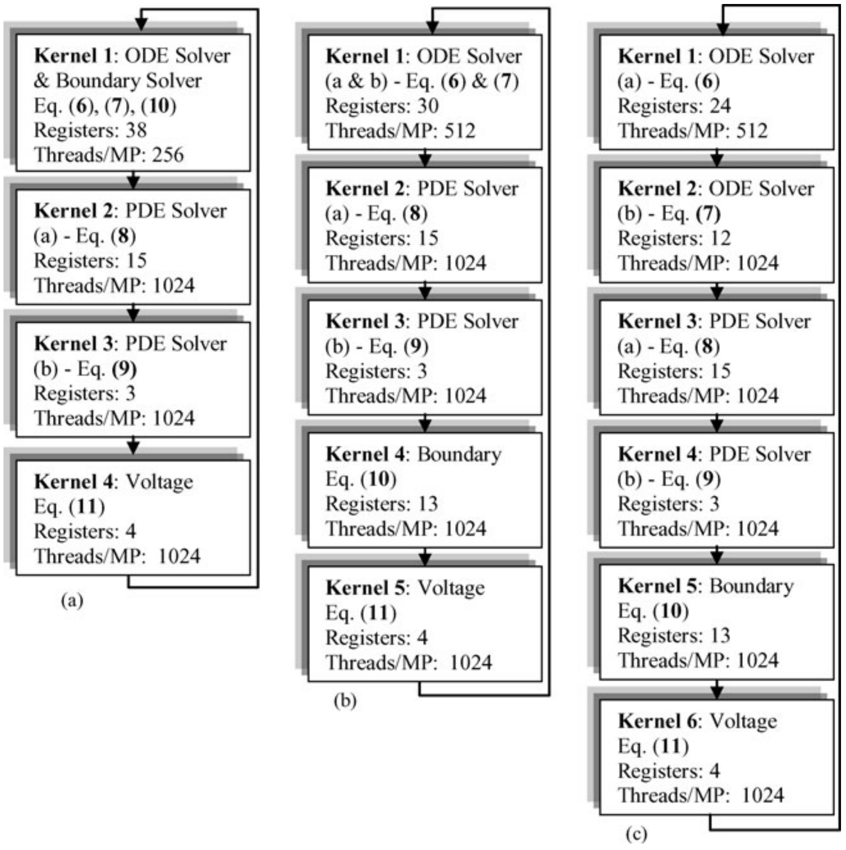

Kernel 1: ODE Solver & Boundary Solver Eq. (6), (7), (10) Registers: 38 Threads/MP: 256
Kernel 2: PDE Solver (a) - Eq. (8) Registers: 15 Threads/MP: 1024
Kernel 3: PDE Solver (b) - Eq. (9) Registers: 3 Threads/MP: 1024
Kernel 4: Voltage Eq. (11) Registers: 4 Threads/MP: 1024
(a)
Kernel 1: ODE Solver (a & b) - Eq. (6) & (7) Registers: 30 Threads/MP: 512
Kernel 2: PDE Solver (a) - Eq. (8) Registers: 15 Threads/MP: 1024
Kernel 3: PDE Solver (b) - Eq. (9) Registers: 3 Threads/MP: 1024
Kernel 4: Boundary Eq. (10) Registers: 13 Threads/MP: 1024
Kernel 5: Voltage Eq. (11) Registers: 4 Threads/MP: 1024
(b)
Kernel 1: ODE Solver (a) - Eq. (6) Registers: 24 Threads/MP: 512
Kernel 2: ODE Solver (b) - Eq. (7) Registers: 12 Threads/MP: 1024
Kernel 3: PDE Solver (a) - Eq. (8) Registers: 15 Threads/MP: 1024
Kernel 4: PDE Solver (b) - Eq. (9) Registers: 3 Threads/MP: 1024
Kernel 5: Boundary Eq. (10) Registers: 13 Threads/MP: 1024
Kernel 6: Voltage Eq. (11) Registers: 4 Threads/MP: 1024
(c)

Fig. 6 Execution flow with 3 parallelization strategies (**a**) Strategy 1 with 4-Kernels, (**b**) Strategy 2 with 5-Kernels, (**c**) Strategy 3 with 6-Kernels

ductivity vectors) into the limited shared memory space of the GPU, therefore, we utilize the global memory in our implementation. Since global memory access latency is from 400 to 600 cycles, it is crucial to hide this latency through memory coalescing. For that purpose, we designed the data structure for storing the grid space that is aligned in the memory in terms of warp size. All the threads accessing the memory are streamlined by the use of this data structure and threads in a warp are forced to access the consecutive memory locations. Figure 8 shows the performance improvement achieved through memory coalescing for the case of $256 \times 256 \times 256$ grid space based on Strategy 3.

Figure 9 shows the execution time for Strategy 3 with respect to change in grid size. Each grid size includes 8 times the amount of cells larger than its predecessor. Until the gird size of $64 \times 64 \times 64$, as the data size increase, execution time increases with less than a factor of 8. Execution time increases with a factor of $8\times$ beyond the grid size of $64 \times 64 \times 64$ which means threading power of the GPU is utilized and beyond this grid size execution of the program progresses in multiple iterations of the program flow. In the following section, we describe the multi-GPU implementation

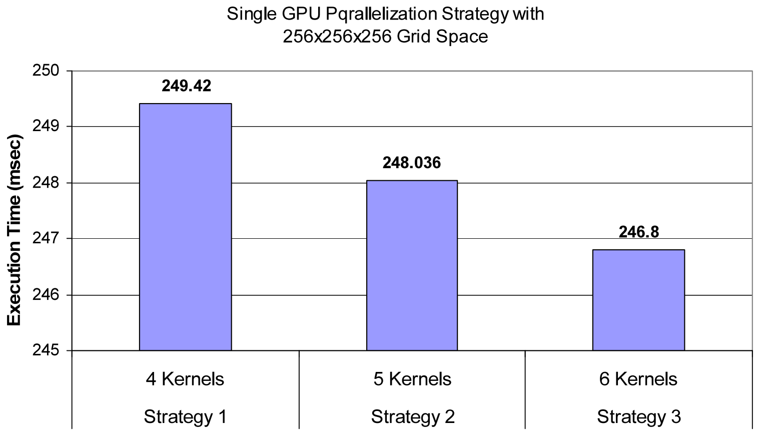


Fig. 7 Execution time for three parallelization strategies

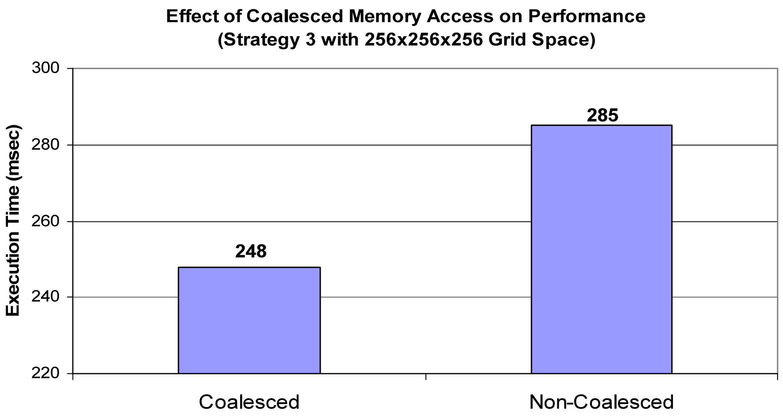


Fig. 8 Performance improvement due to memory coalescing

and then present performance comparison among single-GPU, multi-GPU, and GPP based implementations.

4.3 Heart simulation on multi GPU system

Four Tesla cards are used for experimentation. A host thread can create a context with only one GPU, therefore, we created four host threads to run the simulation on four GPUs. Synchronization of these GPUs does not force considerable penalty as workload on all GPUs are almost the same. In order to share the data produced by GPUs after each iteration, host level thread synchronization by means of barrier directive is used. We considered OpenMP and POSIX threads for creating host level threads. The amount of multithreading code required to perform simple operations made POSIX threads an inconvenient choice for using Multiple GPUs. On the other hand, OpenMP has a good abstraction of low level details which is not the case with

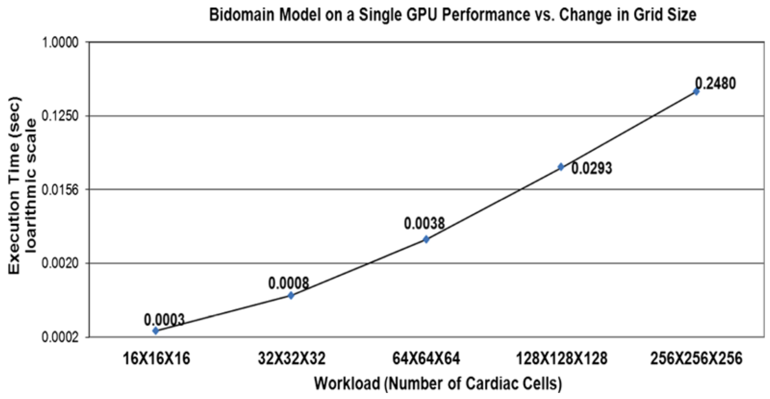


Fig. 9 Execution time for Strategy 3 with respect to change in workload (based on logarithmic scale base 8) shows linear increase in execution time beyond 64 × 64 × 64 grid size

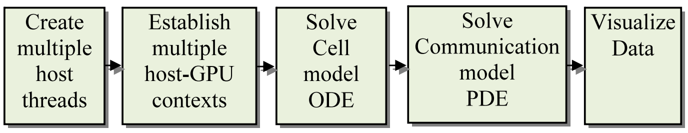

Create multiple host threads
Establish multiple host-GPU contexts
Solve Cell model ODE
Solve Communication model PDE
Visualize Data

Fig. 10 Multi GPU implementation flow

POSIX programs. Supporting the choice of OpenMP, there also exist many software tools for performance evaluation and debugging of OpenMP programs. Host threads keep the data produced by each GPU in the shared data structure. Figure 10 shows the flow of the multi-GPU implementation.

4.3.1 Jacobi iteration and solving PDE

In solving the PDE equation, new challenges arise in terms of data partitioning when multiple GPUs are used. Each cell requires data from surrounding cells after each iteration. For the cells which lie on inter GPU boundary, voltage values of the neighboring cells are received from the other GPUs. To address the data dependency, we evaluated various data sharing techniques. The naive technique is to copy the grid space back to host CPU after each iteration. Once CPU has all the data required, it can launch a separate kernel which calculates the voltage values for cells in the interface region. In this method, it is quite evident that memory transfer would take most of the time due to redundant data transfer between the CPU and the GPUs. An alternative is to copy only the boundary cells back to the CPU after each Jacobi iteration. Figure 11 illustrates our grid partitioning strategy and Fig. 12 illustrates the execution flow.

One of the major concerns in programming multi-GPUs is the nonavailability of tools and libraries. A programmer needs to take care of all levels of detail like memory hierarchy and alignment of data structure in memory. For cardiac simulations, the inter-GPU data dependency poses a major issue. Data partitioning becomes difficult

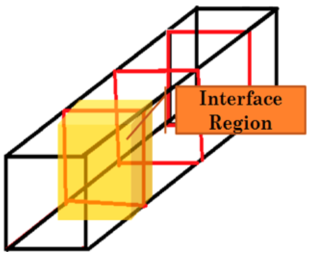


Fig. 11 The shaded region has inter GPU Data Dependency

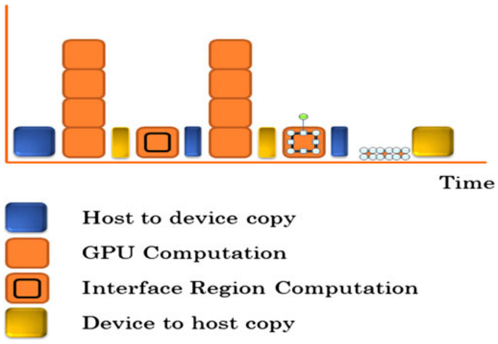


Fig. 12 Showing the process of calculating PDE

Table 1 GPU-CPU system configuration

GPU-CPU System			
Host System		GPU System	
Brand	Supermicro	GPU Model	Tesla C1060
Host CPU	Intel Xeon 2.4 GHz	Number of GPUs	4
Interface	8 GB/s (PCIe 2.0 × 16)	Core Clock (GHz)	1.3
Memory	24 GB	Global Memory	4 GB
OS	Fedora Linux	Memory Bandwidth	102 GB/s
Compiler	-g –use_fast_math -arch sm_13	Theoretical FLOP	933 Gflops/sec

if the data is not organized properly. Another challenge is finding ways to avoid irregular memory accesses by choosing proper indexing scheme in terms of block and thread IDs. If the indexing scheme is selected in such a way, that threads can access all the data in inter GPU boundary, then selective copy of boundary data is possible. This reduces the time taken for inter-GPU synchronization.

5 Results

Tables 1 and 2 show the device configurations for the GPU and GPP systems, respectively.

On a 4-GPU system, simulating one APD cycle (350 msec real time) for 256 × 256 × 256 tissue in 10,000 time steps takes a total of 664.05 seconds including the

Table 2 General Purpose Processor (GPP) system configuration

GPP System	
Machine Model	Asus
Processor	Intel I7 920
Clock Rate	2.4 GHz
Hard Disk	640 GB
RAM	9 GB
OS	Ubuntu
Compiler Options	gcc O3 Wall

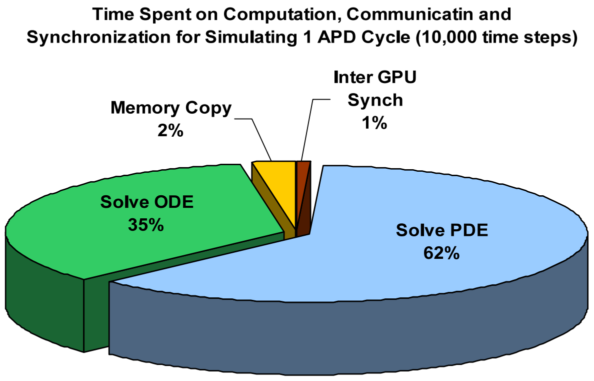


Fig. 13 Showing the time taken by various stages of the program

computation, communication, and data transfers between the host and the device. This is a drastic improvement (a factor of 2460×) compared to the serial version of the same simulation. As shown in Fig. 13, computation time (ODE and PDE solvers) for this simulation dominates the total execution time. For the multi-GPU scenario, communication time includes the overhead due to OpenMP primitives. Single-GPU and multi-GPU implementations are 657 and 2,538 times faster than the serial version respectively based on the computation time.

Table 3 illustrates the total execution time by considering the data transfer overhead for a 256 × 256 × 256 tissue after 10,000 iterations on single-GPU and multi-GPU platforms. Tissue data is transferred from the host once and the output of each iteration is accumulated in a file at the host side. To illustrate the impact of data transfer overhead on performance, in Table 4 we report the speedup values for a single time step execution based on computation time only. For both single-GPU and multi-GPU platforms, speedup values improve significantly, showing the computation efficiency of the GPU when data transfer time overhead is not considered. Table 4 presents the performance comparison between single-GPU, 4-GPU, and GPP based implementations. For small gird size (16 × 16 × 16) single GPU performs better than multi GPU due the synchronization overhead associated with the multi-GPU system. As the grids size increases, speedup for the 4-GPU version converges to the factor of 4× compared to the single-GPU version as expected. Heart simulation calculations involve 88.87 × 10^9 floating point operations for 256 × 256 × 256 grid size. There-

Table 3 Computation and Total Execution Time (seconds) for 256 × 256 × 256 grid size for 10,000 iterations on the GPU

	Computation time	Memory Copy and Synchronization Time		Total time (sec)	Speedup over GPP
		Host to Device	Device to Host		
single-GPU	0.248	0.485	0.432	6800	239
multi-GPU	0.064	0.125	0.121	1850	881

Table 4 Single Time Step Computation Time for GPU and GPP Systems with respect to change in workload

Workload	Computation time (seconds)			Speedup over GPP		Speedup
	Single GPU	Four GPUs	GPP	Single GPU	Four GPUs	Four GPUs over Single GPU
16 × 16 × 16	0.00029	0.00030	0.021	72	70	0.97
32 × 32 × 32	0.00079	0.00042	0.189	239	453	1.89
64 × 64 × 64	0.00383	0.00122	1.5	392	1232	3.14
128 × 128 × 128	0.02932	0.00765	14	478	1830	3.83
256 × 256 × 256	0.24804	0.06423	163	657	2538	3.86

fore based on the computation time reported in Table 3, the single-GPU reaches 360 GFLOPS.

5.1 Correctness of implementation

Excitation propagation in 3D cardiac tissue is visualized using OpenDX toolkit from IBM. The data generated by GPUs are stored in a file which is processed to generate offline visualization showing the propagation of wave. Apart from that, we randomly selected a cell in the grid and dumped its voltage values with respect to time in a file. The values were used to plot the APD curve shown in Fig. 14(a) and showed that the output matches the expected APD curve.

A video for the visualization of the GPU based bidomain simulation tool generating the voltage propagation across the 3D grid system is hosted at http://acl.ece.arizona.edu/cardiacsim/video.html and sample screen shots are shown in Fig. 15.

6 Related work

In [15], Vigmond et al. studied the effect of offloading ODEs to GPUs as they are slower than PDEs in CPU/cluster of CPUs. ODEs are inherently parallel and are best candidates for GPUs. In [16], Sato et al. proposed a method to solve ODEs and PDEs on GPUs. It is evident from their work that solving PDEs take more time than solving ODEs on GPUs. Whereas on CPU/cluster of CPUs, solving ODEs take more time as

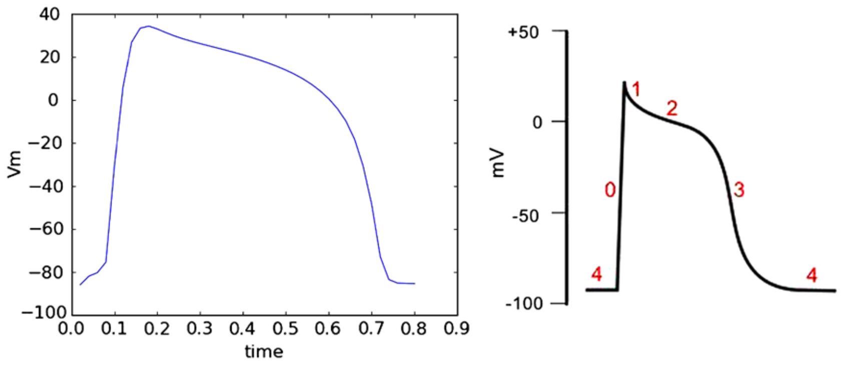


Fig. 14 (**a**) APD curve plotted using sampled values in our experiment, (**b**) Actual APD curve voltage change in grid based

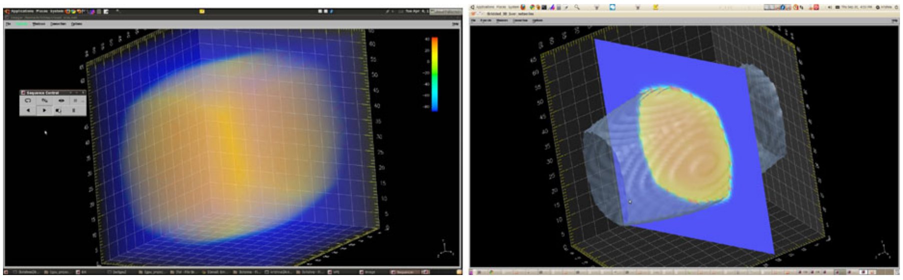


Fig. 15 The voltage values calculated at each node in the grid can be used to visualize the excitation propagation in cardiac tissue

they are computationally intensive. Both studies shows that PDEs is the bottleneck in GPU implementation. In [16], the operator splitting technique is used to break cardiac equations into PDE and ODE. We also used the operator splitting technique in our implementation. However, our approach to solving PDEs is different from the technique used in [16]. In our implementation, we employ the Jacobi iterative technique to solve PDEs which is more efficient and accurate as it iteratively calculates the result until the values in consecutive values converge to a small value. Though [16] has a 3D implementation, not enough results are reported for us to conduct a fair evaluation. For 100 × 100 cells, their 2D implementation takes 8.2 seconds for simulating 1 second of real time, whereas our implementation takes around the same time to simulate (10 × 10 × 10) cells in 3D for 1 second real time. The sequential implementation times reported in [16] is also comparable with our serial execution time. For the larger tissue of 800 × 800 cells, their implementation is about 2 times slower than our 3D implementation with 40 × 40 × 40 cells.

In [17], Unat et al. proposed the multi-GPU implementation of 2D tissue using MPI. In their configuration, multiple GPUs are connected to different host machines. Hosts synchronize one another through message passing. Values of shared cells among the GPU partitions are updated before each iteration through message passing. In our 3D implementation, all 4 GPUs are connected to the same host machine.

As each host thread can only manage one GPU context, multiple threads are created using OpenMP for the multi-GPU implementation. The inter-GPU synchronization time is nit significant in our case as the data is moved around in a single system, avoiding any uncertainties/delays caused by the network. Our inter-GPU synchronization time (which includes OpenMP barrier synchronization, exchange of shared cell values) is 1% of the total execution time per iteration. On the other hand, their implementation requires synchronization time which is 4.4% of the total execution time per iteration. It is also notable that 3D implementation is computationally heavier than 2D implementation even for same number of cells because of the complex nature of the PDEs being solved. Considering the average execution time per iteration, our implementation of 16 × 16 × 16 cells is (4849/300) 16.2× faster than their 4K cells implementation. Moreover, none of the papers that discuss the 3D Cardiac simulations have provided enough information about their parallelization strategies and how various limitations of GPUs can be overcome to provide good speedup.

7 Conclusion

3D simulations of electric wave propagation help medical researchers understand electrical instabilities and excitation dynamics in cardiac tissues. The cardiac bidomain model is a popular approach for studying the electrical behavior of cardiac tissues. The model simulates interactions between the tissue cells by solving a large number of unknowns with nonlinear differential equations and calculating the transmembrane voltage. One second of real time tissue modeling takes several days to simulate on a high end general purpose processor. The iterative and inherently data parallel nature of the bidomain model's software architecture makes it an ideal match for the fine grained parallel hardware architecture of the NVIDIA Graphic Processing Units (GPU). In this study, we evaluate the architecture specific fine grained parallelization and optimization strategies, identify the suitable threads per block configuration, and study the impact of memory organization and coalesced memory access on performance. We study the challenges for porting the single GPU implementation of an application onto a multi-PGU system. Simulating a single action potential duration (APD) cycle (350 msec real time) occurs in 10,000 time steps. In the case of simulating one time step, as the size of the 3D tissue space increases, the speedup curve reaches to saturation reaching up to 657× and 2631× with single-GPU and multi-GPU implementations, respectively, compared to general purpose processor. In the case of simulating the APD cycle for a 256 × 256 × 256 tissue, general purpose processor based system takes 453 hours to execute the program, while it takes 664 seconds on a four-GPU based system including the communication and data transfer overhead. This drastic improvement (a factor of 2460×) will allow clinicians to extend the time-scale of simulations from milliseconds to seconds and minutes; and evaluate hypotheses in shorter amount of time that was not feasible previously.

Acknowledgements This material is based upon work supported by the National Science Foundation under Grant No. IIP-0758579 for I/UCRC Center for Autonomic Computing.

References

1. Ten Tusscher KH, Panfilov AV (2006) Alternans and spiral breakup in a human ventricular tissue model. Am J Physiol 90:326–345
2. Jack JJB, Noble D, Tsien RW (1975) Electric current flow in excitable cells. Clarendon, Oxford
3. NVIDIA Corporation (2007), CUDA Programming Guide 1.0. http://www.nvidia.com
4. Micikevicius P (2009) 3d Finite difference computation on GPUs using CUDA. In: GPGPU-2: Proceedings of 2nd workshop on general purpose processing on graphics processing units. ACM, New York, pp 79–84
5. Qu Z, Garfinkel A (1999) An advanced numerical algorithm for solving partial differential equation in cardiac conduction. IEEE Trans Biomed Eng 46(9):1166–1168
6. Roth BJ (1991) Action potential propagation in a thick strand of cardiac muscle. Circ Res 68:162–173
7. Smith GD (1985) Numerical solution of partial differential equations: finite difference methods, 3rd edn. Clarendon, London
8. Ten Tusscher KHWJ, Noble D, Noble PJ, Panfilov AV (2004) A model for human ventricular tissue. Am J Physiol 286:H1573–H1589
9. Luo CH, Rudy Y (1991) A model of the ventricular cardiac action potential: depolarization, repolarization, and their interaction. Circ Res 68:1501–1526
10. Beeler GW, Reuter H (1977) Reconstruction of the action potential of ventricular myocardial fibres. J Physiol 268:177–210
11. Noble D (1962) A modification of the Hodgkin-Huxley equations applicable to Purkinje fibre action and pace-maker potentials. J Physiol 160:317–352
12. McAllister RE, Noble D, Tsien RW (1975) Reconstruction of the electrical activity of cardiac Purkinje fibres. J Physiol 251:1–59
13. Yanagihara K, Noma A, Irisawa H (1980) Reconstruction of sino-atrial node pacemaker potential based on the voltage clamp experiments. Jpn J Physiol 30:841–857
14. Cardiac modeling, http://www.cardiacmodeling.org/
15. Vigmond EJ, Boyle PM, Joshua Leon L, Plank Gernot (2009) Near-real-time simulations of biolelectric activity in small mammalian hearts using graphical processing units. In: Engineering in medicine and biology society, pp 3290–3293
16. Sato D, Xie Y, Weiss JN, Qu Z, Garfinkel A, Sanderson AR (2009) Acceleration of cardiac tissue simulation with graphic processing units. In: Medical and biological engineering and computing, pp 1011–1015
17. Unat D, Cai X, Baden S (2010) Optimizing the Aliev-Panfilov model of cardiac excitation on heterogeneous systems. University of Iceland, Reykjavik