Bio-Medical Materials and Engineering 24 (2014) 1417–1424
DOI 10.3233/BME-130946
IOS Press

Parallel Acceleration for Modeling of Calcium Dynamics in Cardiac Myocytes

Ke Liu [a], Guangming Yao [b], and Zeyun Yu [a,*]
[a] *Department of Computer Science, University of Wisconsin-Milwaukee, WI, USA*
[b] *Department of Mathematics, Clarkson University, NY, USA*

Abstract. Spatial-temporal calcium dynamics due to calcium release, buffering, and re-uptaking plays a central role in studying excitation-contraction (E-C) coupling in both healthy and defected cardiac myocytes. In our previous work, partial differential equations (PDEs) had been used to simulate calcium dynamics with realistic geometries extracted from electron microscopic imaging data. However, the computational costs of such simulations are very high on a single processor. To alleviate this problem, we have accelerated the numerical simulations of calcium dynamics by using graphics processing units (GPUs). Computational performance and simulation accuracy are compared with those based on a single CPU and another popular parallel computing technique, OpenMP.

Keywords: calcium dynamics, PDEs, parallel, OpenMP, CUDA, GPU

1. Introduction

Heart failure has been one of the leading causes of human deaths in many countries including the United States. The prevalence of this disease is largely due to our lack of accurate understanding of excitation-contraction (E-C) coupling in cardiomyocytes [1,2,3]. For its central role in E-C coupling, modeling Ca^{2+} release and concentration change has been an active research area. In the present paper, we are interested in investigating spatial-temporal variations of intra-cellular calcium concentration at cellular and sub-cellular levels. At these scales, deterministic methods utilizing partial differential equations (PDEs) are more appropriate than stochastic methods [4,5]. The local radial basis function collocation method (LRBFCM) developed by Šarler and Vertnik [6] has been applied to solving the PDEs in our earlier work [7]. This meshless method eliminates the generation of meshes, as commonly required in finite element methods. However, the computational costs of such simulations are very high, especially when realistic geometries are considered. To this end, the main contribution of the present work is to reduce the computational time by employing modern parallel computing techniques and make comparisons between the different approaches on the specific simulation problem for numerical simulations of calcium dynamics in cardiac myocytes.

The work described was supported in part by an NIH Award (Number R15HL103497) from the National Heart, Lung, and Blood Institute (NHLBI) and by a grant from the UWM Research Growth Initiative.
*Corresponding author. E-mail: yuz@uwm.edu.

Traditional techniques on parallel computing are MPI (Message Passing Interface) and OpenMP. MPI is a distributed-memory architecture that communicates between different machines by sending/receiving messages. OpenMP, on the other hand, is a shared-memory architecture and works on a single machine with multiple cores (or CPUs). Computation on graphics processing units (GPUs) is a newer parallel methodology, which has become increasingly popular in recent years. It is a heterogeneous-memory architecture and uses graphic cards as co-processors. Modern GPUs have thousands of cores, which makes it well suited for large-scale data parallelism. There are three programming models on GPUs, namely, Open Computing Language (OpenCL), Compute Unified Device Architecture (CUDA), and DirectCompute. Among these models, CUDA is the most user-friendly and widely used, thus we decided to use CUDA in the present work. In our experiments, the computational performance and simulation accuracy of the serial version of the algorithm are compared to both OpenMP (with 4 cores) and GPU-CUDA implementations.

2. Mathematical Models and Meshless Numerical Methods

2.1. Governing Equations

To model calcium dynamics in cardiac myocytes, the following nonlinear reaction-diffusion equations, modified from [8], are considered:

$$\frac{\partial [Ca^{2+}]_i}{\partial t} = D_{Ca}\nabla^2[Ca^{2+}]_i - \sum_{m=1}^{3} R_{B_m} - R_{B_s},\ in\ \Omega \tag{1}$$

$$\frac{\partial [CaB_m]}{\partial t} = D_{CaB_m}\nabla^2[CaB_m] + R_{B_m},\ in\ \Omega,\ m = 1, 2, 3, \tag{2}$$

$$\frac{\partial [CaB_s]}{\partial t} = R_{B_s},\ in\ \Omega \tag{3}$$

$$\frac{\partial [Ca^{2+}]_i}{\partial t} = J_{Caflux},\ on\ \partial\Omega \tag{4}$$

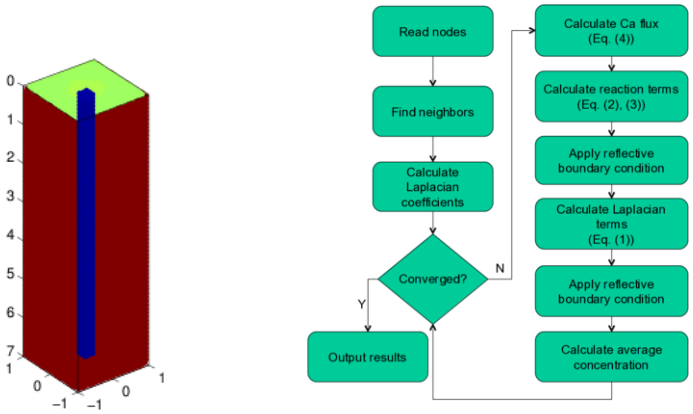
where Ω is the interior of cell and $\partial\Omega$ is the cell surface and t-tubule membrane. In [8], the calcium flux term J_{Caflux} is defined in the entire domain, although it always takes a zero value at internal nodes. In our work, however, this term is explicitly defined only on the boundary $\partial\Omega$. Therefore, instead of merging the calcium flux term in the first equation, we have an additional equation in Eq. (4).

The initial conditions (resting states) used are as follows: $[Ca^{2+}]_i = 0.10\mu M$, $[CaB_1] = 11.92\mu M, [CaB_2] = 0.97\mu M, [CaB_3] = 0.13\mu M$, $[CaB_s] = 6.36\mu M$. Note that we exam our model and methods on a portion of the cell, in which reflective boundary conditions are applied during numerical simulation on the part of $\partial\Omega$ where it is not the cell surface or t-tubule membrane. The reactions between Ca^{2+} and buffers are given by:

$$R_{B_m} = k_+^m([B_m] - [CaB_m])[Ca^{2+}]_i - k_-^m[CaB_m],\ m = 1, 2, 3, \tag{5}$$

$$R_{B_s} = k_+^s([B_s] - [CaB_s])[Ca^{2+}]_i - k_-^s[CaB_s] \tag{6}$$

In our model, three types of mobile Ca^{2+} buffers (Fluo-3, ATP, and calmodulin, denoted by B_m, $m = 1, 2, 3$) and one type of stationary Ca^{2+} buffers (troponin, denoted by B_s) are considered. Their concentrations are denoted by $[CaB_m]$, $m = 1, 2, 3$, $[CaB_s]$, respectively. At the resting (initial) state,



(a) (b)

Fig. 1. (a) The model considered in the current study, containing the t-tubule (blue), surrounding half sarcomere (red), and external cell membrane (green). (unit: μm). (b) Flow chart of the meshless algorithm for modeling of calcium dynamics.

all buffers were distributed uniformly throughout the cytosol but not on the cell membrane. The resting concentrations of mobile and stationary buffers satisfy equilibrium conditions (i.e. $R_{B_m} = R_{B_s} = 0$) [1]. The initial concentrations of buffers are calculated in equilibrium with the resting Ca^{2+} concentration, $0.1\mu M$. The total Ca^{2+} flux, J_{Caflux}, on the surface membrane is defined in [1], where Ca^{2+} influx/efflux through L-type calcium channels (LCCs) , sodium-calcium exchangers (NCXs), Ca^{2+} pumps and background leaks are included. J_{Caflux} throughout the cell surface membrane and the surface of t-tubules is defined as follows: $J_{Caflux} = J_{Ca} + J_{NCX} - J_{pCa} + J_{Cab}$, where J_{Ca} is total LCC Ca^{2+} influx; J_{NCX} is total NCX Ca^{2+} influx; $J_{pCa}$is total Ca^{2+} pump efflux; and J_{Cab} is total background Ca^{2+} leak influx.

2.2. Geometric Model Considered

According to [9,10,11], a ventricular myocytes may be simplified as repeated structural units consisting of a single t-tubule and its surrounding half sarcomeres. The surrounding half sarcomeres are modeled as a cube-shaped box with dimension of $2\mu m \times 2\mu m \times 7\mu m$, enclosing a t-tubule with dimension of $0.2\mu m \times 0.2\mu m \times 6.8\mu m$. The t-tubule is assumed to be a tiny cube located vertically in the center of sarcomeres, as shown on Figure 1(a).

2.3. Space Discretization (LRBFCM)

The time domain in the system of reaction-diffusion equations is discretized uniformly and explicitly. Thus, we need to approximate Laplacian term in each equation with certain spatial discretization. The LRBFCM is used to approximate Laplacian term $\nabla^2 u(\mathbf{x}, t)$ in Equations (1) and (2). Interested readers are referred to [1,12] for more details. The main idea of LRBFCM is that the collocation can be done on overlapping local domains, yielding many systems of equations with small matrices instead of a single large matrix. The size of the collocation matrices depends on the number of nodes in the local domains.

Briefly speaking, $\nabla^2 u(\mathbf{x}, t)$ at each node $\mathbf{x}_i,\ i = 1, 2, \cdots, N$, is approximated by its neighbors. The local domain Ω_i associated with $\mathbf{x}_i$ can be created using the n-nearest neighbors to $\mathbf{x}_i$ including itself,

i.e. $\left\{\mathbf{x}_k^{[i]}\right\}_{k=1}^{n} \subset \Omega_i$. In this work, the number of points in each local domain is fixed at $n = 7$. To approximate $\nabla^2 u(\mathbf{x}_i, t)$, we interpolate $u(\mathbf{x}_i, t)$ on Ω_i by using the basis function $\Phi(r)$ as follows:

$$u(\mathbf{x}_j^{[i]}, t) = \sum_{k=1}^{n} \Phi\left(\left\|\mathbf{x}_j^{[i]} - \mathbf{x}_k^{[i]}\right\|\right) \alpha_k^{[i]},\ j = 1, 2, \cdots, n, \tag{7}$$

where the coefficients $\alpha_k^{[i]}$ are unknown but can be calculated by solving an $n \times n$ linear system in the local domain Ω_i. In the current work we adopt the multiquadrics (MQ) as the basis function, although there are many other commonly used RBFs such as inverse multiquadrics (IMQ), Gaussian, thin-plate splines (TPS), polyharmonic splines (PS). The MQ RBF is defined as follows: $\Phi(r) = \sqrt{r^2 + c^2}$, where $r = \|\mathbf{x} - \mathbf{x}_c\|$ is the Euclidean norm, $\mathbf{x}_c$ is the center of Φ, and $c > 0$ is called the shape parameter. This type of RBFs has been proved to have a high-order rate of convergence. The RBF shape parameter in MQ plays a major role in improving the accuracy of numerical solutions. In general, the optimal shape parameter depends on the densities, distributions and function values at the nodes. However, it is very difficult to assign different free parameters for each local domain. Thus, choosing shape parameters has been an active topic in approximation theory [13]. In our experiments, $c = 300$ is used. We refer interested readers to [6,7,14,15,16,17].

Based on Equations (1) and (2), the calcium influx J_{Caflux}, reactions $R_{B_m}, m = 1, 2, 3, s$ and Laplacian term $\nabla^2[Ca^{2+}]$ need to be approximated. Because the points we used are fixed in the given geometric space over time, the Laplacian coefficients $\alpha_k^{[i]}$ are constants for each point and thus can be precalculated. However, the J_{Caflux}, $R_{B_m}, m = 1, 2, 3, s$ and $\nabla^2[Ca^{2+}]$ will be updated in each time step. Figure 1(b) shows the algorithm.

3. Parallel Implementations

In the current work, OpenMP and GPU are used to speed up the simulation and their performances are compared with the serial implementation. Some implementation details of OpenMP and GPU programs are given in this section. The CPU-based experiments (serial and openMP) are performed on a Dell Precision T7400 workstation with two quad-cores of 3.0 GHz Intel Xeon X5472 processors and 16 GB memory, a RedHat platform of kernel version 2.6.18-308.4.1.el5, and a GCC compiler of version 4.4.7. The serial experiment uses a single core and the OpenMP experiment uses four cores. The GPU-based experiments are performed on a Dell Precision T3500 workstation with a NVIDIA GeForce GTX 560 Ti, 2 GB device memory, and a CentOS 6.4 platform of kernel version 2.6.32-358.14.1.el6.x86_64. The NVIDIA video card driver has a version of 319.32.

3.1. Parallelization with OpenMP

Because the calculations of Laplacian terms are most time-consuming in the algorithm (see Fig. 1(b)), OpenMP is used to parallelize this step. The Laplacian terms have to be calculated for every point in the domain. OpenMP automatically divides the calculations into multiple chunks (4 chunks in our experiments because we use 4 cores). Then it forks multiple threads and each thread calculates one chunk of work load. All threads execute the same algorithm but with different data. This is called data-parallelism. At the end of this step, all partial results are merged automatically and these working threads are synchronized implicitly. Because the overhead of thread-forking and thread-joining is larger than the good they can provide in other steps in the algorithm, we do not apply OpenMP in other steps in the algorithm.

3.2. Parallelization with GPU

CUDA is an implementation of heterogeneous programming involving CPU (called host) and GPU (called device). The functions executed on GPU are called kernel functions. The execution starts with the serial host. When a kernel function is launched, the execution switches to the device (GPU), where a large number of threads are initiated to take advantage of abundant data parallelism. All the working threads in a kernel are organized in groups called thread blocks. When the kernel finishes executing, the control returns back to the host and the serial execution continues on the host until the next kernel launches or program terminates [18]. While the GPU is computing, CPU is available to do other tasks. In out implementations, however, CPU is simply waiting for the GPU to finish. The thread-switching on GPU has almost no overhead, so CUDA is able to create a one-to-one mapping between data points and threads. If the number of threads created are more than the number of CUDA cores available, they are scheduled to execute in batches. In execution, the number of scheduled threads is determined by several factors, including the number of CUDA cores available, the number of registers available, and the size of on-chip shared-memory. To achieve the best performance, we briefly summarize below some special techniques considered in our implementations.

3.2.1. Use of shared-memory

Because CUDA has thousands of threads running in parallel, the reads and writes to the off-chip device memory (called global memory) tend to have a memory bandwidth contention, which can significantly lower the performance. Threads in the same block, however, can share data in a small on-chip memory (called shared-memory). In our experiments, the calculation of Laplacian terms needs to read Laplacian coefficients and Ca^{2+} or $CaB_m, m = 1, 2, 3$ concentrations at neighboring points. To alleviate the global memory bandwidth contention, these variables are loaded into shared-memory once at the beginning of this step instead of multiple reads when needed. If the neighbor coefficients and concentrations are in the same block as the current point, they are fetched from shared-memory. Otherwise, they are loaded from global memory. The calculation of Laplacian terms is implemented as a kernel function. Unlike the global memory, however, the contents of shared-memory is not persistable across kernels. So at the end of this step, results have to be synchronized back to global memory.

3.2.2. Reordering of data points

To increase the hit of shared-memory, we need to find a way to maximize the possibility that a point and its neighbors are all in the same block. If they are not in the same block, their coefficients and concentrations have to be loaded from global memory, which lowers the performance due to the global memory bandwidth contention. The number of thread blocks used is rounded up to $\lceil \frac{number\ of\ points\ in\ a\ dataset}{number\ of\ threads\ per\ block} \rceil$. Please refer to Section 3.2.5 for details of block dimensions. When reading points, we reorganize the points in geometry and group them into many cells (blocks). The points in a block are grouped roughly in a cubic region such that the points and their neighbors are likely to be in the same block.

3.2.3. Use of constant and texture memories

As stated in Section 3.2.1, the reads and writes of global memory have memory bandwidth contention. The constant and texture memories have built-in caches. When a program reads the same variable more than once, it actually reads it from the cache, which eliminates the bandwidth problem. Constant and texture memories are read-only and constant memory is very small (only 64 KB shared by all stream-multiprocessors). There are dozens of constant arguments (refer to [7]) in constant memory. Texture memory shows a higher performance when data are localized with each other. Therefore, the neighborhood information found by the kd-tree is saved in texture memory.

3.2.4. Unrolling for-loops

When dealing with iterations, programs on CPU use loops with specific components like programming counter, instruction decoder, and so on. However, CUDA cores are simplified to have ALU (arithmetic-logic unit) only. Thus executing loops in CUDA is slower than CPU. If one can unroll a loop by storing the involved variables into multiple registers instead of using one register and refresh that register at every iteration, the compiler can decode the instructions more efficiently and generate faster code. However, since the number of registers available is limited (GeForce GTX 560 Ti has 8 multiprocessors and each has $32,768$ registers), the growing usage of registers could decrease the number of threads scheduled to execute and thus lowers the device occupancy. Device occupancy is determined by the equation $\frac{number\ of\ threads\ scheduled}{warp\ size}$, where the warp size is 32 in current CUDA-enabled GPUs. Interested readers may refer to [18] for more details. In our implementation, unrolling loops increases the number of registers used by a single thread to 27. The device occupancy decreased from 50% to 33.3%. So, there is a balance of increased code efficiency and decreased device occupancy. Whether to use this technique is really a case-by-case scenario. We found that using this technique can improve the performance by about 10% in calculation of Laplacians.

3.2.5. Global synchronization

Unlike the CPU, CUDA threads are scheduled to execute in batches. For a point requiring concentrations at neighboring points, it is likely that the threads responsible for the neighboring points are not executed yet and thus the neighboring concentrations are not updated. To cope with this situation, threads in GPU have to be synchronized. Unfortunately in the current GPU design, a kernel function can only synchronize threads in the same block. To synchronize threads in different blocks (global synchronization), a kernel function has to be split into smaller ones. In our algorithm, every step (see the 6 boxes on the right side of Fig. 1(b)) is implemented as a kernel, yielding a total of 6 kernels. These kernels use different thread block dimensions in order to achieve maximum device occupancy. As calculating Ca^{2+} flux and reaction terms does not require neighboring concentrations, we use a relatively small block size (128 threads per block), so that each thread can use more registers and shared-memory. Ca^{2+} flux is calculated on T-Tubule and cell membrane with a 58.3% device occupancy. Reaction terms are calculated on interior points with a 66.7% device occupancy. In applying reflective boundary condition, calculating Laplacian terms and calculating average concentration, we use shared-memory as high-speed cache. A large block size (512 threads per block) is used to increase the possibility of shared-memory hit. Reflective boundary condition is applied at boundary points at a 100% device occupancy. Laplacian terms and average concentrations are calculated at all points with a 33.3% and 100% device occupancy respectively. At the end of each step, results are synchronized into global memory.

4. Result and Discussion

We consider the model shown in Figure 1(a) to simulate calcium dynamics and compare the performances of OpenMP and GPU with the serial implementation. The model is discretized in three resolutions: Set 1 contains a total of $3,969$ points (including 136 T-Tubule points and 80 cell membrane points) with a node distance $0.2\mu m$. Set 2 contains a total of $30,807$ points (including 545 T-Tubule points and 360 cell membrane points) with a node distance $0.1\mu m$. Set 3 contains a total of $234,921$ points (including $2,185$ T-Tubule points and $1,512$ cell membrane points) with a node distance $0.05\mu m$. The time steps for the three cases are $8e^{-3}$ ms, $4e^{-3}$ ms and $1e^{-3}$ ms respectively to make the PDEs converge. The total time we simulated is 400 ms.

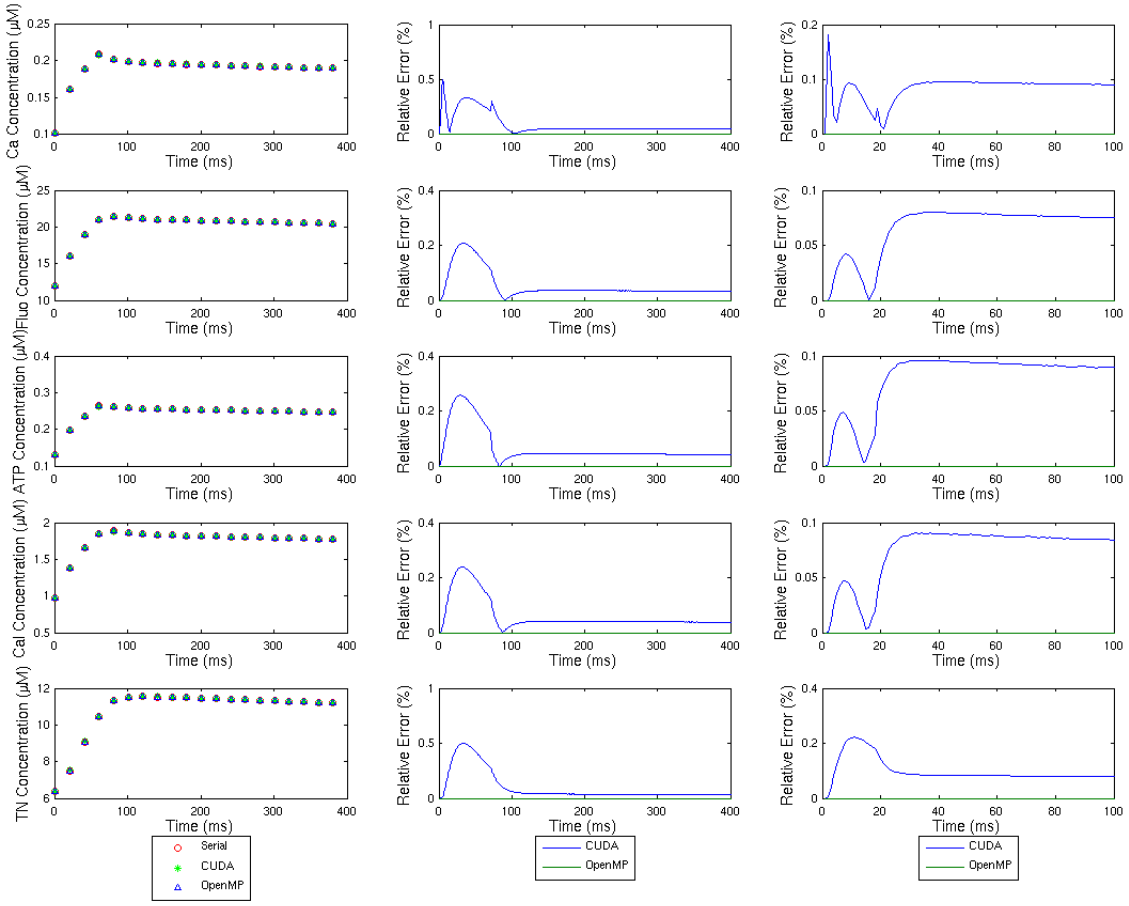


Fig. 2. Column 1 shows the average concentrations of Ca^{2+} and mobile and stationary buffers for point set 2 (30, 807 points) in three implementations, namely, serial, OpenMP and CUDA. Note that the concentrations of the three versions are almost identical. Column 2 shows the average relative errors of point set 2 (30, 807 points), as compared to the serial execution. Column 3 shows the average relative errors of point set 3 (234, 921 points), as compared to the serial execution.

Column 1 in Figure 2 shows the average concentrations of Ca^{2+}, mobile and stationary buffers in three implementations, namely, serial, OpenMP and CUDA, per the legend displayed below the figure. Column 2 and Column 3 show their average relative errors compared to the serial implementation for the second and third point sets, respectively. Because an analytical solution does not exist for such a complicated mathematical model (see Eq. (1)- Eq. (4)), we have used the numerical simulation from the serial LRBFCM approach [7] as the ground truth for comparison, which had been shown to agree with the finite element-based simulation [8] and with the available experiments as well. The errors in Column 2 and Column 3 show that OpenMP has exactly the same simulation results as the serial version. The CUDA result shows up to 0.5% relative error for the medium model (column 2) and 0.2% relative error for the large model (column 3). A close inspection discovers that the errors are partly caused by the fact that the GPU treats floating-point numbers differently from the CPU. The error curves also show that, as the number of points increases, the accuracy of CUDA implementation also increases.

Table 1 shows the running time of serial, OpenMP and CUDA on the three point sets. For a small model (set 1), OpenMP has the lowest performance because the overhead of multithreads is larger than the

Table 1
Running time of Serial, OpenMP, CUDA on 3 point sets (unit: seconds)

Number of Points	Serial	OpenMP (4 cores)	CUDA
Point set 1 (3, 969 points)	26.17	28.21	7.15
Point set 2 (30, 807 points)	504.52	419.95	49.94
Point set 3 (234, 921 points)	26893.21	17974.51	1356.85

gains. CUDA has a better performance but only 3.66X of the serial's speed. For a medium size of model (set 2), OpenMP and CUDA show 1.20X and 10.10X performance increases respectively, as compared to the serial implementation. For a large model (set 3), OpenMP shows 1.5X performance increase and CUDA shows 19.82X performance increase. Apparently the GPU shows a great performance boost and is very promising in data parallelism, thanks to a large number of cores available on GPUs.

However, GPU cores have simpler circuits compared to CPU cores and require special techniques like instruction-level optimization to get the best performance boost. Moreover, simpler circuits also restrict the applications of GPUs to simple tasks like data parallelism. The small device memory (usually 1 to 4 GB) also limits the amount of data a GPU can process simultaneously. For large input data, it is necessary to divide the data into multiple parts so that each part can be fitted into the device memory.

References

[1] Bers DM. Calcium cycling and signaling in cardiac myocytes, Annual Review of Physiology 2008; 70:23-49.
[2] Bers DM. Cardiac excitation-contraction coupling, Nature 2002; 415(6868):198-205.
[3] Michailova A, DelPrincipe F, Egger M, Niggli E. Spatiotemporal features of Ca2+ buffering and diffusion in atrial cardiac myocytes with inhibited sarcoplasmic reticulum, Biophysical J. 2002; 83(6):3134-3151.
[4] Koh X, Srinivasan B, Ching HS, Levchenko A. A 3D monte carlo analysis of the role of dyadic space geometry in spark generation, Biophys. J. 2006; 90(6):1999-2014.
[5] Izu LT, Means SA, Shadid JN, Chen-Izu Y, Balke CW. Interplay of ryanodine receptor distribution and calcium dynamics, Biophys. J. 2006; 91(1):95-112.
[6] Vertnik R, Šarler B. Solution of incompressible turbulent flow by a mesh-free method, Computer Modeling in Engineering and Sciences 2009; 44(1):65-95.
[7] G. Yao and Z. Yu, A Localized Meshless Approach for Modeling Spatial-temporal Calcium Dynamics in Ventricular Myocytes, International Journal for Numerical Methods in Biomedical Engineering, 28(2):187-204, 2012.
[8] Lu S, Michailova A, Saucerman J, Cheng Y, Yu Z, Bank R, Kaiser T, Li W, Holst M, McCammon J, Hayashi T, Arzberger P, McCulloch A, Cheng Y, Hoshijima M. Multiscale modeling in rodent ventricular myocytes, Engineering in Medicine and Biology Magazine, IEEE 2009; 28:46-57.
[9] Soeller C, Cannell MB. Examination of the transverse tubular system in living cardiac rat myocytes by 2-photon microscopy and digital image processing techniques, Circulation Research 1999; 84(3):266-275.
[10] Pasek M, Brette F, Nelson A, Pearce C, Qaiser A, Orchard C, Christe G. Quantification of t-tubule area and protein distribution in rat cardiac ventricular myocytes, Progress in Biophysics and Molecular Biology 2008; 96(1-3):244-257.
[11] Hinch R, Greenstein JL, Tanskanen AJ, Xu L, Winslow R. A simplified local control model of calciuminduced calcium release in cardiac ventricular myocytes, Biophysical Journal 2004; 87(6):3723-3736.
[12] Fasshauer GE. Meshfree Approximation Methods with MATLAB. World Scientific Press, Singapore, 2007.
[13] Wang JG, Liu GR. On the optimal shape parameters of radial basis functions used for 2-D meshless methods, Computer Methods in Applied Mechanics and Engineering 2002; 191:2611-2630.
[14] Kosec G, Šarler B. Local RBF collocation method for darcy flow, Computer Modeling in Engineering and Sciences 2008; 25(3):197-208.
[15] Šarler B, Vertnik R. Meshfree explicit local radial basis function collocation method for diffusion problems, Computers and Mathematics with Applications 2006; 51(8):1269-1282.
[16] Vertnik R, Šarler B. Meshless local radial basis function collocation method for convective-diffusive solidliquid phase change problems, International Journal of Numerical Methods for Heat and Fluid Flow 2006; 16(5):617-640.
[17] Divo E, Kassab AJ. An efficient localized RBF meshless method for fluid flow and conjugate hear transfer, ASME Journal of Heat Transfer 2007; 129:124-136.
[18] Kirk D, NVIDIA, Hwu W. Programming Massively Parallel Processors: A Hands-on Approach. Elsevier Inc, 2010