Long-time Simulation of Calcium Induced Calcium Release in a Heart Cell using the Finite Element Method on a Hybrid CPU/GPU Node

Xuan Huang
Department of Mathematics and Statistics,
University of Maryland, Baltimore County
1000 Hilltop Circle, Baltimore, MD 21250
hu6@umbc.edu

Matthias K. Gobbert
Department of Mathematics and Statistics,
University of Maryland, Baltimore County
1000 Hilltop Circle, Baltimore, MD 21250
gobbert@umbc.edu

ABSTRACT

A mathematical model of Calcium Induced Calcium Release in a heart cell has been developed that consists of three coupled non-linear advection-diffusion-reaction equations. A program in C with MPI based on matrix-free Newton-Krylov method gives very good scalability, but still requires large run times for fine meshes. A programming model with CUDA and MPI that utilizes GPUs on a hybrid node can significantly reduce the wall clock time. This paper reports initial results that demonstrate speedup using a hybrid node with two GPUs over the best results on a CPU node.

Author Keywords

Calcium Induced Calcium Release, finite element method, GPU, MPI

ACM Classification Keywords

I.6.1 SIMULATION AND MODELING (e.g. Model Development). : Performance

INTRODUCTION

A mathematical model for the calcium induced calcium release (CICR) in a heart cell has been developed [5–7] that consists of three coupled non-linear reaction-diffusion equations, which describe the concentrations of calcium ions (C), fluorescent calcium indicator (F), and the endogenous calcium buffers (B). In [2] (see also [4] for details) a matrix-free Newton-Krylov method for the simulation of calcium induced calcium release in a heart cell was presented. The underlying model of calcium flow is given by a system of three coupled diffusion-reaction equations, in which the occurring source terms are highly nonlinear point sources modeled by Dirac delta distributions. The method is based on a finite element discretization and implemented in a matrix-free manner. The convergence of the finite element method in presence of measure valued source terms, as they occur in the calcium model, was rigorously shown in [11] and numerical results agree well with the theoretical predictions.

This paper extends above by implementing finite element with Newton-Krylov methods on GPUs (graphics processing units). In [2], performance studies show good scability, programmed in C with MPI. However, the run time is still huge for large mesh size, requiring excessive time if use few number of nodes. During recent years, general purpose GPUs offer an opportunity to greatly increase the throughput, and implement new ideas in parallel computing.

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model by NVIDIA. The CUDA architecture included a unified shader pipeline, allowing each and every arithmetic logic unit (ALU) on the chip to be marshaled by a program intending to perform general-purpose computations. A typical Tesla K20 GPU that we use from NVIDIA has 2496 cores, and is capable to have a theoretical double precision floating point performance of 1.17 TFLOP/s. The problem discussed in this paper is both computationally intensive and massively parallel, offering a good opportunity for better performance on hybrid nodes with CPUs and GPUs. A problem of reaction-diffusion type is solved in [8], where the authors compare different time-integtation methods with a 2D model using one GPU. A Jacobian-free Newton-Krylov method with GPU accleration is discussed in [9], where the problem size is limited by using one GPU. Our method solves a PDE system on three-dimensional domain with a degrees of freedom more than 25 million. We use performance on one state-of-the-art 16-core CPU node as baseline for speedup computation, rather than serial CPU performance on one core. Our implementation with MPI and CUDA is scalable to multiple nodes with multiple GPUs. This paper reports initial results on one node with 2 GPUs available.

BACKGROUND

The three-species application problem models the flow of calcium on the scale of one heart cell. Calcium ions enter into the cell at calcium release units (CRUs) distributed throughout the cell and then diffuse. At each CRU, the probability for calcium to be released increases along with the concentration of calcium, thus creating a feedback loop of waves re-generating themselves repeatedly. An accurate model of such waves is useful since they are part of the normal functioning of the heart, but can also trigger abnormal arrhythmias. This model requires simulations on the time scale of several repeated

HPC 2015 April 12–15, 2015, Alexandria, VA
Copyright © 2015 Society for Modeling & Simulation International (SCS)

waves and on the spatial scale of the entire cell. This requires long-time studies on spatial meshes that need to have a high resolution to resolve the positions of the calcium release units throughout the entire cell [2].

The problem can be modeled by a system of coupled, non-linear, time-dependent advection-diffusion-reaction equations

$$u_t^{(i)} - \nabla \cdot \left(D^{(i)} \nabla u^{(i)}\right) + \beta^{(i)} \cdot \left(\nabla u^{(i)}\right) = q^{(i)} \quad (1)$$

of $i = 1, \ldots, n_s$ species with $u^{(i)} = u^{(i)}(\mathbf{x}, t)$ representing functions of space $\mathbf{x} \in \Omega \subset \mathbb{R}^3$ and time $0 \le t \le t_{\text{fin}}$. The diffusivity matrix $D^{(i)} = \text{diag}\,(D_{11}^{(i)}, D_{22}^{(i)}, D_{33}^{(i)}) \in \mathbb{R}^{3\times 3}$ consists of positive diagonal entries, which are assumed to dominate the scale of the advection velocity vectors $\beta^{(i)} \in \mathbb{R}^3$, so that the system is always of parabolic type. We consider the rectangular domain shown in Figure 1, where numerical mesh is also demonstrated in a very coarse way. The model also incorporates no-flux bound-

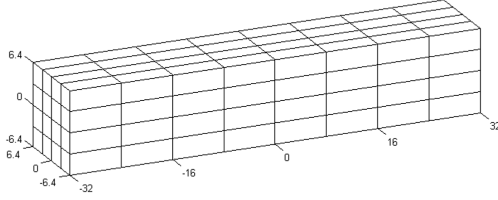

Figure 1. Rectangular domain, units in μm.

ary conditions

$$\mathbf{n} \cdot (D_i(\mathbf{x}) \nabla u^{(i)}) = 0 \qquad \text{for } x \in \partial\Omega,\ 0 < t \le t_{\text{fin}} \quad (2)$$

and a given set of initial conditions

$$u^{(i)}(\mathbf{x}, 0) = u_{\text{ini}}^{(i)}(\mathbf{x}) \quad \text{for } x \in \Omega,\ t = 0. \quad (3)$$

The right-hand side $q^{(i)}$ of each PDE in (1) is written in a way that distinguishes the different dependencies and effects as

$$q^{(i)} = r^{(i)}(u^{(1)}, \ldots, u^{(n_s)}) + s^{(i)}(u^{(i)}, \mathbf{x}, t) + f^{(i)}(\mathbf{x}, t). \quad (4)$$

Our consideration of this problem is inspired by the need to simulate calcium waves in a heart cell. The general system (1) consists of $n_s = 3$ equations corresponding to calcium ($i = 1$), an endogenous calcium buffer ($i = 2$), and a fluorescent indicator dye ($i = 3$). We describe terms on the right hand side (4) as following: The term $f^{(i)} = f^{(i)}(\mathbf{x}, t)$ incorporates a scalar linear test problem that is been discussed in [3], here we set $f^{(i)} \equiv 0$ for all i. Terms belonging only to the first equation are multiplied with the Kronecker delta function δ_{i1}, where $i = 1, \ldots, n_s$,

$$s^{(i)}(u^{(i)}, \mathbf{x}, t) = (-J_{pump}(u^{(1)}) + J_{leak} + J_{SR}(u^{(1)}, \mathbf{x}, t))\delta_{i1}. \quad (5)$$

These are the nonlinear drain term J_{pump}, the constant balance term J_{leak} and the key term of the model J_{SR}. This term houses the stochastic aspect of the model, since the calcium release unites (CRUs) which are arranged discretely on a three-dimensional lattice, each have a probability of opening depending on the concentration of calcium present at that site. This process is explained through the following equation:

$$J_{SR}(u^{(1)}, \mathbf{x}, t) = \sum_{\hat{\mathbf{x}} \in \Omega_s} g\ S_{\hat{\mathbf{x}}}(u^{(1)}, t)\delta(\mathbf{x} - \hat{\mathbf{x}}). \quad (6)$$

The equation models the superposition of calcium injection into the cell at special locations called calcium release units (CRUs), which are modeled as point sources. g controls the amount of calcium injected into the cell and Ω_s represents the set of all CRUs. $S_{\hat{\mathbf{x}}}$ is an indicator function, its value is either 1 or 0 indicating the CRU at $\hat{\mathbf{x}}$ is open or closed. When the CRU is open, it stays open for 5 millisecond, then it remains closed for 100 millisecond. The value of $S_{\hat{\mathbf{x}}}$ is determined by compare the value of a random number and the value of the following probability function

$$J_{\text{prob}}(u^{(1)}) = \frac{P_{\max}\,(u^{(1)})^{n_{\text{prob}}}}{(K_{\text{prob}})^{n_{\text{prob}}} + (u^{(1)})^{n_{\text{prob}}}}. \quad (7)$$

When the value of the probability function is higher, $S_{\hat{\mathbf{x}}} = 1$, otherwise $S_{\hat{\mathbf{x}}} = 0$. Furthermore, $\delta(\mathbf{x} - \hat{\mathbf{x}})$ denotes a Dirac delta distribution for a CRU located in $\hat{\mathbf{x}}$.

The reaction terms $r^{(i)}$ shown below are nonlinear functions of the different species and couple the three equations.

$$r^{(i)} := \begin{cases} \sum_{j=2}^{n_s} R^{(j)}(u^{(1)}, u^{(j)}), & \text{for } i = 1, \\ R^{(i)}(u^{(1)}, u^{(i)}), & \text{for } i = 2, \ldots, n_s, \end{cases} \quad (8)$$

where the reaction rates are given by

$$R^{(i)} = -k_i^+ u^{(1)} u^{(i)} + k_i^- \left(\overline{u}_i - u^{(i)}\right) \qquad \text{for } i = 2, \ldots, n_s. \quad (9)$$

A complete list of the model's parameter values is given in Table 1.

NUMERICAL METHODS

In order to numerically simulate the calcium induced calcium release model, a numerical method must be designed that is very efficient in memory use. The uniform rectangular CRU lattice gives a naturally induced regular numerical mesh, see Figure 1. Additionally, the model uses constant diffusion coefficients. Using a finite element method (FEM) with these properties (constant coefficients, regular mesh) will allow for system matrices whose components can be computed by analytical formulas. Therefore routines, specifically the matrix-vector product, can be designed without an explicitly stored system matrix. A matrix-free method dramatically reduces the memory requirements of the method, thereby making useful computations feasible. This reduced memory requirements also enable us to solve the

Table 1. Table of parameters for the CICR model.

Parameter	Description	Values/Units
t	Time	ms
$\mathbf{x}$	Position	μm
$u^{(i)}$	Concentration	μM
Ω	Rectangular domain in μm	$(-6.4, 6.4) \times (-6.4, 6.4) \times (-32.0, 32.0)$
$D^{(1)}$	Calcium diffusion coefficent	diag$(0.15, 0.15, 0.30)$ μm^2/ ms
$D^{(2)}$	Mobile buffer diffusion coefficient	diag$(0.01, 0.01, 0.02)$ μm^2/ ms
$D^{(3)}$	Stationary buffer diffusion coefficient	diag$(0.00, 0.00, 0.00)$ μm^2/ ms
$u^{(1)}_{\text{ini}}$	Initial calcium concentration	0.1 μM
$u^{(2)}_{\text{ini}}$	Initial mobile buffer concentration	45.9184 μM
$u^{(3)}_{\text{ini}}$	Initial stationary buffer concentration	111.8182 μM
Δx_s	CRU spacing in x-direction	0.8 μm
Δy_s	CRU spacing in y-direction	0.8 μm
Δz_s	CRU spacing in z-direction	0.2 μm
g	Flux density distribution	110.0 μM μm^3 / ms
P_{max}	Maximum probability rate	0.3 / ms
K_{prob}	Probability sensitivity	0.2 μM
n_{prob}	Probability Hill coefficient	4.0
Δt_s	CRU time step	1.0 ms
t_{open}	CRU opening time	5.0 ms
t_{closed}	CRU refractory period	100 ms
k_2^+	Forward reaction rate	0.08 / (μM ms)
k_2^-	Backward reaction rate	0.09 / ms
$\bar{u}_2$	Total of bound and unbound indicator	50.0 μM
k_3^+	Forward reaction rate	0.10 / (μM ms)
k_3^-	Backward reaction rate	0.10 / ms
$\bar{u}_3$	Total bound and unbound buffer	123.0 μM
V_{pump}	Maximum pump strength	4.0 μM / ms
K_{pump}	Pump sensitivity	0.184 μM
n_{pump}	Pump Hill coefficient	4
J_{leak}	Leak term	0.320968365152510 μM / ms

problem on a fine mesh with relatively small GPU memory. The NVIDIA K20 GPU we use has 5 GB of memory, compared to 64 GB memory connected to CPU. The convergence of the finite element method in presence of measure valued source terms, as they occur in the model (1) and (4), was rigorously shown in [11], and numerical results agree well with the theoretical predictions [2].

The spatial discretization of the three-species application problem with tri-linear nodal finite elements results in a large system of ordinary differential equations (ODEs). This ODE system is solved by the family of numerical differentiation formulas (NDFk) with variable order $1 \leq k \leq 5$ and adaptively chosen time step size [12, 13]. Since these ODE solvers are fully implicit, it is necessary to solve the fully coupled non-linear system of equations at every time step. For its solution a matrix-free newton method is applied, which means that results of the Jacobian-vector products needed in the Krylov subspace method are provided directly without storing the Jacobian. In addition, the usage of the exact Jacobian should lead to quadratic convergence of the Newton method. The linear solver makes use of the iterative BiCGSTAB method with matrix-free matrix-vector multiplies. Table 2 summarizes several key parameters of the numerical method and its implementation. The first three columns show the spatial mesh resolution of $N_x \times N_y \times N_z$, the number of mesh points $N = (N_x + 1)(N_y + 1)(N_z + 1)$, and their associated numbers of unknowns $n_s N$ for the n_s species that need to be computed at every time step, commonly referred to as degrees of freedom (DOF). The following column lists the number of time steps taken by the ODE solver, which are significant and which increase with finer resolutions. The final two columns list the memory usage in GB, both predicted by counting variables in the algorithm and by observation provided in a memory log file produced from the performance run. We notice that even the finest resolution fits comfortably in the memory of one NVIDIA K20 GPU.

CUDA + MPI IMPLEMENTATION

Motivation

The motivations behind General-Purpose Computation on Graphics Processing Unit (GPGPU) are multifold. First of all, the models that we have are becoming more and more complicated. We need to solve the complicated problem with finer meshes, larger (cell) domain, longer simulation time. We also need to prepare for more species which means more PDEs coupled. Furthermore, we might ran into situations that thousands of simulations are required for certain studies [1].

The degree of freedom (DOF) as well as computational burden will increase substantially as the model increases complexity. And the current MPI approach that runs on Multi-core CPU cluster will reach limit. This approach has good scalability on the condition that you have access to large number of compute nodes with cutting edge CPUs.

Hence, offloading to accelerators such as GPUs are considered natural choices. Our problem is suitable for GPU computation because of the following reasons: The program is computationally intensive, heavy computation can be done on the GPU, with few data transfer. The program is also massively parallel, similar tasks are performed repeatedly on different data. Also, from the cost effective aspect, a state-of-the-art GPU card is much cheaper to acquire compare to a up-to-date CPU node, while providing comparable or more throughput.

Hardware Used

The hardware used in the computational studies is part of the UMBC High Performance Computing Facility (HPCF). Each hybrid node contain two eight-core 2.6 GHz Intel E5-2650v2 Ivy Bridge CPUs and 64 GB memory, see Figure 2. Each CPU is connected to one state-of-the-art NVIDIA K20 GPU, see Figure 3. The nodes are connected by a high-speed quad-data rate (QDR) InfiniBand network.

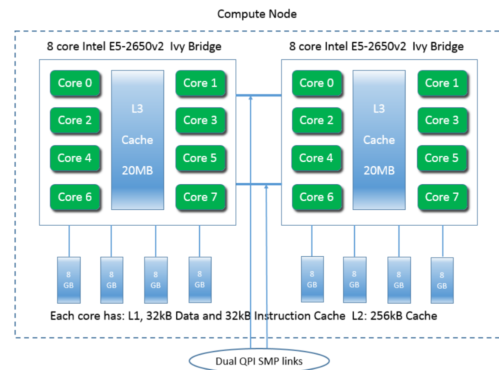

Figure 2. Schematics of CPU: Intel E5-2650v2 Ivy Bridge

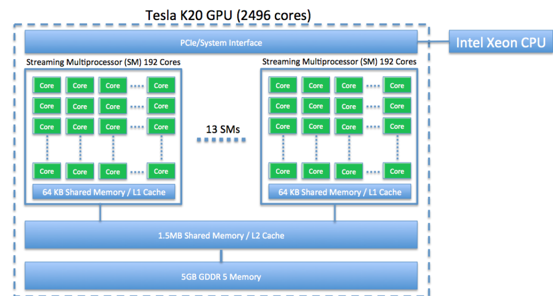

Figure 3. Schematics of NVIDIA Tesla K20 GPU.

CUDA + MPI Workflow

In CUDA programming language, CPU and the system's memory are referred to as host, and the GPU and its memory are referred to as device. Figure 4 explains how threads are grouped into blocks, and blocks grouped into grids. Threads unite into thread blocks – one- two or three-dimensional grids of threads that interact with each other via shared memory and synchpoints. A program (kernel) is executed over a grid of thread blocks. One grid is executed at a time. Each block can also be

Table 2. Sizing study listing the mesh resolution $N_x \times N_y \times N_z$, the number of mesh points $N = (N_x+1) \times (N_y+1) \times (N_z+1)$, the number of degrees of freedom (DOF $= n_s N$), the number of time steps taken by the ODE solver, and the predicted and observed memory usage in MB for a one-process run.

$N_x \times N_y \times N_z$	N	DOF	number of time steps	memory usage (GB) predicted	memory usage (GB) observed
$32 \times 32 \times 128$	140,481	421,443	58,416	0.05	0.08
$64 \times 64 \times 256$	1,085,825	3,257,475	73,123	0.41	0.48
$128 \times 128 \times 512$	8,536,833	25,610,499	89,088	3.24	3.68

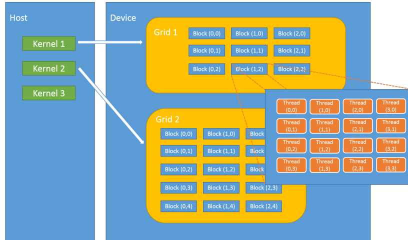

Figure 4. Schematic of Blocks and Threads

one-, two-, or three-dimensional in form. This is due to the fact that GPUs used to work on graphical data, which has 3 dimensions red, green and blue. This now gives much flexibility in launching kernels with different data structure. However, there are still limitations, such as the maximum dimension size of a thread block is $(1024, 1024, 64)$, and the maximum number of threads per block is 1024 for the K20 GPU we have.

The program used to perform the parallel computations presented in this paper is an extension of the one described in [2] and [10], which uses MPI for parallel communications. Therefore, it inherited the main structure of the C program. However, to enable efficient calculations on GPU, almost all calculations have been redesigned to take advantage of GPU parallelism. While inputs are still managed by host, C structs are shared by host and device. Large arrays are allocated directly on device memory before numerical iterations, hence to prevent frequent communications between host and device. Since host handles MPI communication and output to files, data communications between host and device occur before and after these events. The CUDA program has several levels, a detailed discussion is as follows:

The uppermost level is where computational resources are managed. First, MPI processes are setup in `main.cu`. After detect the number of CUDA capable devices (NVIDIA GPUs) on each node, the `main.cu` function then set CUDA device for each MPI process. The idea behind this setup is to allow each MPI process to have access to a unique GPU device. If more than one MPI processes are accessing the same GPU, kernels will queue up and the performance will be degraded. Since in our cluster each GPU enabled node has two GPUs, each connected to a CPU socket via PCI bus, as shown in Figure 3. This means the best approach is to request 2 processes from each node and have each MPI process utilize a unique GPU. We can also run the program in serial, with a single MPI process and one GPU. The 5 GB GPU memory can hold the memory allocated for all large arrays for our current highest mesh $128 \times 128 \times 512$. After the computational resources are correctly allocated, the `main.cu` program launches the program that does the actual computation. Lastly, the `main.cu` program records the total memory used for each CPU node.

The next level, the program selects the right solver based on a set of parameters. The code is a package that can solve many different problems with one-dimensional, two-dimensional and three-dimensional domains. In the case of the CICR model, `main.cu` call the function `run_parabolic.cu`, Within `problem_par_3d.cu`, we first read parameters from one input file, setup C structs, allow them to be shared by both C and CUDA code, and write arrays associated to the structs. Parameters in Table 1 are read in and setup in these steps. A crucial arrangement in the code is to put C struct type definitions in one h file called `struct_define.h`. After the definition of various struct data types, need to declare each struct with `extern`. Just like we declare all other functions with `extern "C"`, this is allows the mixture of C and CUDA to work properly. This h file is then included by every other `.cu` files. However, to be able to compile correctly, we also need to put regular struct declarations at the beginning of `main.cu`.

At level three, ODE solver is called. The solver is based on numerical differentiation formulas (NDFk) with variable order $1 \leq k \leq 5$ and adaptively chosen time step size, As described in Section , at each time step a nonlinear system is solved via a matrix-free Newton method. The matrix vector multiplication function therein have many GPU kernels that can readily take data already exist on GPU memory, hence to reduce the cost of transferring data between CPU and GPU memories. Cublas is used for dot product with double precision. While running with multiple MPI processes, data transfers between CPU and GPU memories are inevitable. Firstly, data has to be transferred back to CPU memory for output. Secondly, data needs to be transferred to CPU memory before MPI communication, and transferred back to GPU memory for calculation. But, it is expected that the second level of paralism on GPU will outperform CPU. For the matrix-vector multiplication, we split GPU kernels into two parts, one does not require computation on MPI communicated data, the other does. Since non-blocking MPI communication functions such as `MPI_Isend` and `MPI_Irecv` returns immediately, we

can have MPI communication and the executation of first kernel at the same time. We put a `MPI_Waitall` only before the launch of the second kernel, making sure the MPI communication is finished before accessing these data.

Lastly, it is vitally important to design kernels that can run with different mesh size in Table 2, and also have room for higher mesh. The most crucial design in kernels are the choice of block and grid sizes. As mentioned before, each block and grid can have one, two or three-dimensions. This gives much flexibility in launching kernels with different data structure. In the application problem here, the number of threads in one block is determined by the mesh on x-direction Nx, as `dim3 threads(Nx, 1)`. This allows Nx to be as large as the limit of 1024 threads. Moreover the number of blocks in one grid is determined by the mesh on y and z-directions Ny and $l_N z$, as `dim3 blocks(Ny, l_Nz)`. l means local to the MPI process. `dim3` can be used to define arrays of up to three-dimensions. In this setup, all utility functions and kernels that need to access large arrays on GPU can be designed to use the same block thread counts, making it much easier to program the actual kernels.

RESULTS

Figure 5 shows CRU plots generated from simulations using GPUs, which are similar to those generated by previous C programs run on CPU nodes. The plots in this figure show which CRUs are open at each time step during the simulation. We see that at $t = 100$ a few CRUs are open, the wave mostly spreads along x- and y-dimensions at this point. Later on we see that the CRUs have begun to open on both sides of the cell and spread across it. During our simulation of 1000 ms, several waves have been generated and run across the cell, with similar speed on both ways of the z-direction.

Table 3 summarizes the wall clock times for the CICR problem solved with the finite element method using $p = 1, 2$ and 16 MPI processes on a CPU node with two eight-core CPUs. All runs fit into the memory of the node, but some runs would take longer than the maximum time of 5 days allowed for a job on the system. For the cases, where the run with $p = 1$ process is possible, the parallel scalability is excellent to $p = 2$ processes, and using all 16 cores available on the node is clearly the fastest run in each case.

Table 4 summarizes the wall clock times for the CICR problem solved with the finite element method using a hybrid CPU / GPU node. For mesh resolution $32 \times 32 \times 128$, the wall clock time on one CPU core and one GPU is more than 5 times faster than a serial run on a CPU, but slower than using all 16 cores on a CPU node. For this coarse resolution, the wall clock time on one node with two MPI processes and two GPUs does not improve performance. This is due to time spent on data transfer between CPU and GPU memory dominate over time spent on calculation. For mesh resolution $64 \times 64 \times 256$, the wall clock time on one CPU core and one GPU is more than 15 times faster than a serial run on a CPU. For this resolution, the wall clock time on one node with two MPI processes and two GPUs is 1.8 times faster than using all 16 cores on a CPU node.

The amount of time needed for calculation increased rapidly due to increased mesh size. For the fine mesh resolution $128 \times 128 \times 512$, the serial run on a CPU is not available due to excessive time requirement. The wall clock time on one node with two MPI processes and two GPUs is around 3 times faster than using all 16 cores on a CPU node.

Table 3. Wall clock time for CICR problem solved with FEM using p MPI processes on a CPU node. E.T. indicates excessive time requirement (more than 5 days).

$N_x \times N_y \times N_z$	$p = 1$	$p = 2$	$p = 16$
$32 \times 32 \times 128$	04:12:42	02:11:30	00:20:28
$64 \times 64 \times 256$	29:39:29	15:33:52	02:36:56
$128 \times 128 \times 512$	E.T.	E.T.	42:07:19

Table 4. CICR problem solved with FEM on a hybrid node using p MPI processes and one GPU per MPI process. Each MPI process launches kernels on a unique GPU. (a) Wall clock time, (b) speedup over $p = 16$ MPI processes on a sixteen-core CPU node.

(a) Wall clock time		
$N_x \times N_y \times N_z$	$p = 1$ 1 GPU	$p = 2$ 2 GPUs
$32 \times 32 \times 128$	00:42:33	00:43:32
$64 \times 64 \times 256$	01:58:19	01:25:32
$128 \times 128 \times 512$	25:09:06	13:46:41
(b) Speedup over $p = 16$ run on CPU Node		
$N_x \times N_y \times N_z$	$p = 1$ 1 GPU	$p = 2$ 2 GPUs
$32 \times 32 \times 128$	0.48	0.47
$64 \times 64 \times 256$	1.33	1.83
$128 \times 128 \times 512$	1.67	3.06

CONCLUSIONS

The results demonstrate that using MPI and CUDA on a hybrid node with two CPUs and two GPUs, the CICR problem can be solved much faster than using all 16 cores of two eight-core GPUs on a CPU node. The data transfer between CPU and GPU memory is inevitable, but can be improved by spliting kernels and use non-block MPI communication. In the future we will investigate possible further improvements like using cuda-aware MPI. Moreover, algorithms that can minimize the data communication without huge sacrifice of accuracy are of interest. Additionally, the performance of several hybrid nodes should be compared to using several CPU nodes.

Acknowledgments

Xuan Huang acknowledges support from the UMBC High Performance Computing Facility (HPCF). The

hardware used in the computational studies is part of HPCF. The facility is supported by the U.S. National Science Foundation through the MRI program (grant nos. CNS–0821258 and CNS–1228778) and the SCREMS program (grant no. DMS–0821311), with additional substantial support from the University of Maryland, Baltimore County (UMBC). See `www.umbc.edu/hpcf` for more information on HPCF and the projects using its resources.

REFERENCES

1. Brewster, M. W. The Influence of Stochastic Parameters on Calcium Waves in a Heart Cell. Senior thesis, Department of Mathematics and Statistics, University of Maryland, Baltimore County, 2014.

2. Gobbert, M. K. Long-time simulations on high resolution meshes to model calcium waves in a heart cell. *SIAM J. Sci. Comput. 30*, 6 (2008), 2922–2947.

3. Graf, J., and Gobbert, M. K. Parallel performance studies for a parabolic test problem on the cluster maya. Tech. Rep. HPCF–2014–7, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2014.

4. Hanhart, A. L., Gobbert, M. K., and Izu, L. T. A memory-efficient finite element method for systems of reaction-diffusion equations with non-smooth forcing. *J. Comput. Appl. Math. 169*, 2 (2004), 431–458.

5. Izu, L. T., Mauban, J. R. H., Balke, C. W., and Wier, W. G. Large currents generate cardiac Ca^{2+} sparks. *Biophys. J. 80* (2001), 88–102.

6. Izu, L. T., Wier, W. G., and Balke, C. W. Theoretical analysis of the Ca^{2+} spark amplitude distribution. *Biophys. J. 75* (1998), 1144–1162.

7. Izu, L. T., Wier, W. G., and Balke, C. W. Evolution of cardiac calcium waves from stochastic calcium sparks. *Biophys. J. 80* (2001), 103–120.

8. Marcotte, C. D., and Grigoriev, R. O. Implementation of pde models of cardiac dynamics on gpus using opencl. *arXiv preprint arXiv:1309.1720* (2013).

9. Pethiyagoda, R., McCue, S. W., Moroney, T. J., and Back, J. M. Jacobian-free newton–krylov methods with gpu acceleration for computing nonlinear ship wave patterns. *Journal of Computational Physics 269* (2014), 297–313.

10. Schäfer, J., Huang, X., Kopecz, S., Birken, P., Gobbert, M. K., and Meister, A. A memory-efficient finite volume method for advection-diffusion-reaction systems with non-smooth sources. *Numer. Methods Partial Differential Equations 31*, 1 (2015), 143–167.

11. Seidman, T. I., Gobbert, M. K., Trott, D. W., and Kružík, M. Finite element approximation for time-dependent diffusion with measure-valued source. *Numer. Math. 122*, 4 (2012), 709–723.

12. Shampine, L. F. *Numerical Solution of Ordinary Differential Equations.* Chapman & Hall, 1994.

13. Shampine, L. F., and Reichelt, M. W. The MATLAB ODE suite. *SIAM J. Sci. Comput. 18*, 1 (1997), 1–22.

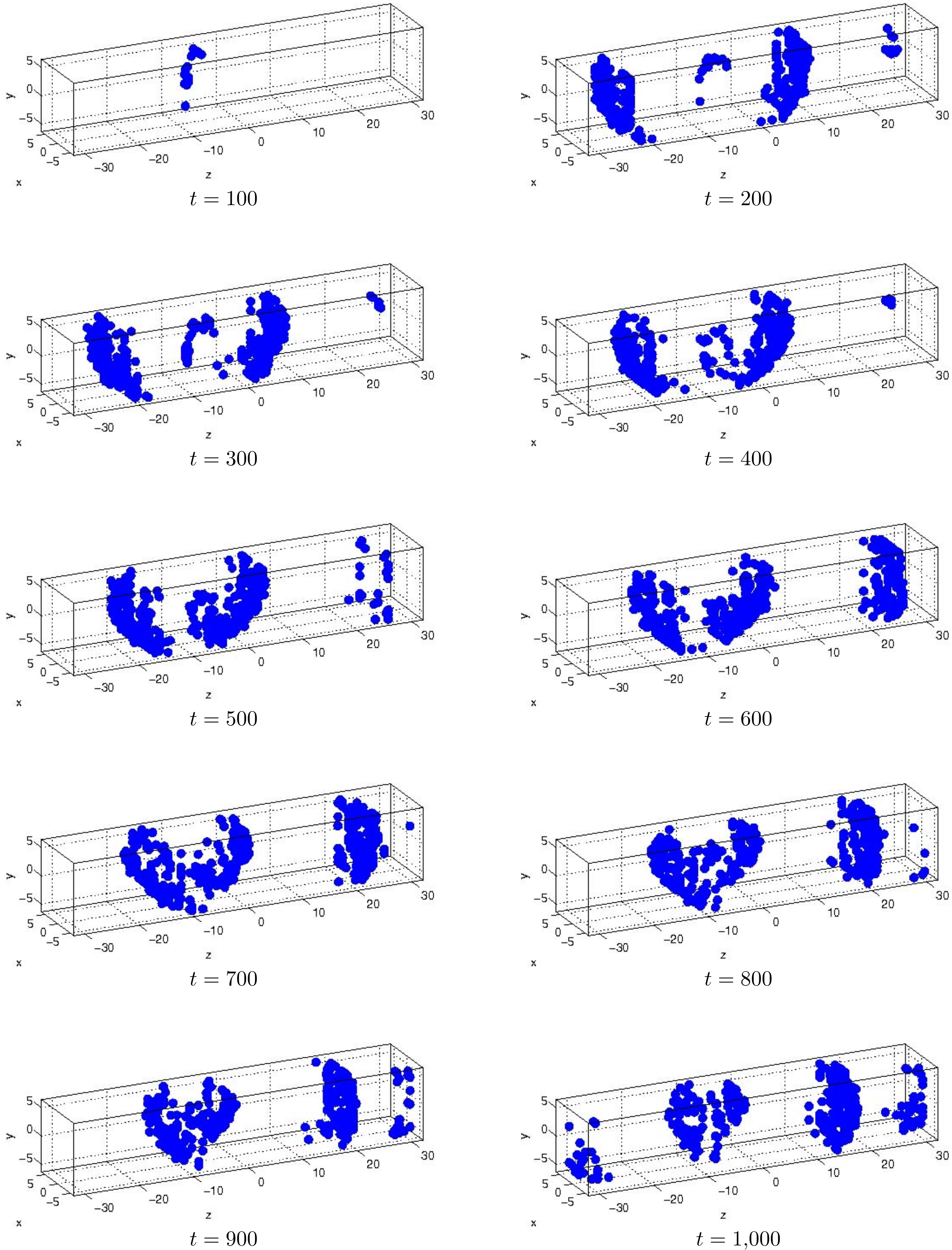

Figure 5. Open calcium release units throughout the cell using finite element method with mesh size $32 \times 32 \times 128$. Based on simulation with CUDA + MPI.