COMPUTATIONAL METHODS IN SCIENCE AND TECHNOLOGY **19(1)** ??—?? (2013)

An application of graphical numerical accelerators in simulations of ion-transport through biological membranes

A. Górecki

Warsaw University of Life Sciences WULS-SGGW,
Nowoursynowska St. 159, 02-787 Warsaw, Poland
E-mail: adam_gorecki@sggw.pl

(Received: 27 October 2012; revised: 20 February 2013; accepted: 21 February 2013; published online: 28 February 2013)

Abstract: The modeling of ion-transport through biological membranes is important for understanding many life processes. The transmembrane potential and ion concentrations in the stationary state can be measured in *in-vivo* experiments. They can also be simulated within membrane models. Here we consider a basic model of ion transport that describes the time evolution of ion concentrations and potentials through a set of nonlinear ordinary differential equations.
To reduce the computation time I have developed an application for simulation of the ion-flows through a membrane starting from an ensemble of initial conditions, optimized for a Graphical Processing Unit (GPU). The application has been designed for the CUDA (Compute Unified Device Architecture) technology. It is written in CUDA C programming language and runs on NVIDIA TESLA family of numerical accelerators. The calculation speed can be increased almost 1000 times compared with a sequential program running on the Central Processing Unit (CPU) of a typical PC.

Key words: biological membranes, electrochemistry, differential equations integration, CUDA, TESLA

I. INTRODUCTION

Every living cell has to exchange energy and mass with its environment in a selective way. An animal cell is surrounded by a lipid bilayer called as a biological membrane [1]. The membrane includes special proteins (channels, transporters and pumps) which enable selective transmission of ions. The activity of these transmission devices is controlled by different factors such as ion concentration, electric fields or the presence of specific molecules.

Studies on ion-transport through biological membranes are crucial for understanding the etiology of many diseases. Abnormal ion transport is the cause of many serious health problems and is responsible for toxicity of many chemicals. For example, the malfunctioning of cystic fibrosis transmembrane conductance regulator (CFTR) channel that controls transport of chlorine and bicarbonate ions in bronchial (lungs) epithelial tissue causes *cystic fibrosis* [2]. Problems in ion transfer through nervous cells are observed in some types of epilepsy [3]. The mechanisms of many poisons or venoms such as charybdotoxin [4] or iberiotoxin [5] are based on blocking the activity of important membrane channels. A recent review of important protein channels has been presented in [6].

The time evolution of membrane potentials and ion concentrations can be modeled using differential equations [7],[8]. The variables of the models (potentials and concentrations) can also be observed in *in-vivo* experiments. The comparison between simulations and experiments allows for optimization of model parameters to make them more realistic. In this paper we present results of simulations of transmembrane potential generated by the epithelial cell monolayer. The calculations are based on a phenomenological model with a reduced number of parameters.

Epithelial cells are external cells of organs contacting with external environment. An epithelial cell membrane consists of two parts:

- *basolateral* – contacting with the internal cells of the organ, and
- *apical* – contacting with an external environment .

Both parts of the cell membrane are covered by a solvent which in a biological system is a water solution of ions at physiological concentrations. The activity of membrane proteins and their selectivity for specific ions define the stationary state of the system. The bilayer is polarized with membrane electric potential (so called resting potentials), which stops effective ion currents. This potential can be measured using special electrodes.
The most important ions contributing to the membrane transport are:

- potassium K^+
- sodium Na^+
- chlorine Cl^-
- bicarbonate HCO_3^-

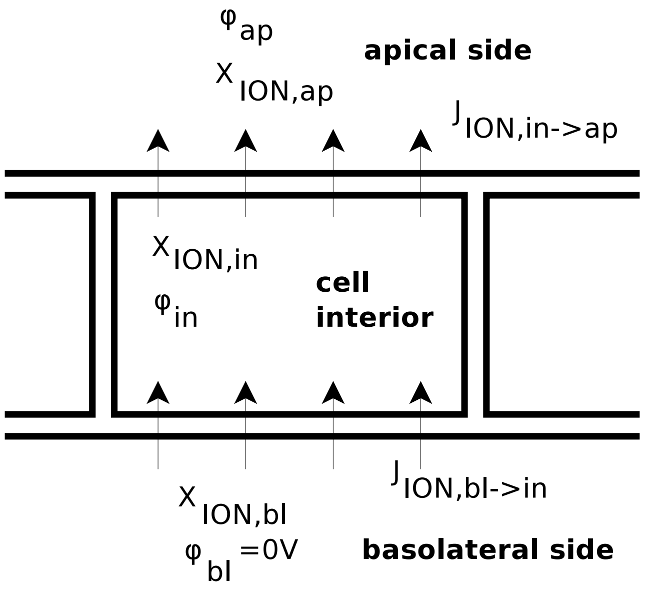

Fig. 1 The considered model of ion transport in epithelial cell monolayer. The space around a cell is divided into apical(*ap*), basolateral (*bl*) and cell interior (*in*) domains

II. THE SIMULATED SYSTEM

The model of the epithelial tissue considered below is illustrated in Figure 1. We assume that:

- the concentrations of ions in basolateral and apical regions are constant in time,
- the basolateral area has reference electric potential φ_{bl}=0 V.

Within our model the state of the system is fully represented by the following variables:

- the apical potential φ_{ap},
- the cell interior potential φ_{in},
- the ion concentrations in the cell interior $X_{ION,in}$, where *ION* = K^+, Na^+, Cl^-, HCO_3^-

The apical potential is equivalent to the transmembrane potential because we have assumed φ_{bl}=0 V.
Potentials $\varphi_{ap}, \varphi_{in}$characterize the state of two capacitors created by apical and basolateral sides of the membrane. The currents charging these 'capacitors' are related to the total flows of ions of different types. There are many models of ion flows described in literature, for example [7],[8]. These models describe time-evolution with ordinary differential equations. The equations include many parameters characterizing specific ion-channel activity. If a model contains too many parameters than usual, it is difficult to find their realistic values. We proposed our model with a reduced number of parameters:

$$\begin{aligned}
\partial_t \varphi_{ap} &= \frac{\sum_{IONS} I_{ION}(in \to ap)}{C_{ap}} \\
&\quad + \frac{\sum_{IONS} I_{ION}(bl \to in)}{C_{bl}} \\
\partial_t \varphi_{bl} &= \frac{\sum_{IONS} I_{ION}(bl \to in)}{C_{bl}} \\
\partial_t X_{ION,in} &= \frac{I_{ION}(in \to ap) - I_{ION}(bl \to in)}{F \cdot z_{ION} \cdot Vol}
\end{aligned}$$

where:

$I_{ION}(bl \to in)$	–	the electric current (positive charges flow) measured in amperes, A, corresponding to flow of ions type *ION* from the basolateral area to the cell interior,
$I_{ION}(in \to ap)$	–	the electric current (positive charges flow) measured in A, corresponding to flow of ions type *ION* from the basolateral area to the cell interior,
C_{bl}	–	the basolateral membrane capacity measured in farads, F,
C_{ap}	–	the apical membrane capacity measured in F,
z_{ION}	–	the sign of ion type ION, dimensionless, $z_{ION} = \pm 1$ corresponding to *ION* charge
Vol	–	the volume of the cell, cubic meters,
F	–	the Faraday constant, *F=96500 C/mol.*

The currents of ions are given by equations:

$$\begin{aligned} I_{ION}(bl \to in) &= G_{ION,bl} \cdot (-\varphi_{in} \\ &\quad - \varphi_{Nernst}(T, z_{ION}, X_{ION,bl}, X_{ION,in})) \\ I_{ION}(in \to ap) &= G_{ION,ap} \cdot (\varphi_{ap} - \varphi_{in} \\ &\quad - \varphi_{Nernst}(T, z_{ION}, X_{ION,in}, X_{ION,ap})) \end{aligned}$$

where:
$\varphi_{Nernst}(T, z_{ION}, X_{ION,src}, X_{ION,trg})$ is the Nernst equilibrium potential for *ION* and flow direction of *src trg*, where *src* and *trg* denote source and target compartments of space. In the considered case both *src* and *trg* can mean apical (*ap*), cell interior (*in*), or basolateral (*bl*) areas.
$G_{ION,ap}$, $G_{ION,bl}$ are effective electric permeabilities (reciprocal resistance measured in reciprocal ohms, Ω^{-1}) of the apical and basolateral side of membrane, respectively.
The positive value of $I_{ION}(src \to trg)$ means non-zero current of positive charges from *src* to *trg*.
The Nernst resting potential [9] is defined as:

$$\varphi_{Nernst}(T, z_{ION}, X_{ION,src}, X_{ION,trg}) = \\ = -\frac{RT}{Fz_{ION}} \cdot \ln\left(\frac{X_{ION,src}}{X_{ION,trg}}\right)$$

where T is the temperature of solvent.
When the potential difference between the sides of membrane *src, trg* is equal to Nernst potential, there is no effective current of ion of type *ION*.

III. IMPLEMENTATION

The simulation program is written in CUDA C language and designed to work on NVIDIA TESLA family of graphical accelerators [10].
The general idea of our approach is to perform the same operations on different data in parallel: it is a so called Single Instruction Multiple Data approach. For our applications (the same algorithm, many data sets, small amount of required operational memory) we need many instances of simple scalar calculations. The NVIDIA GPU accelerator is perfectly suited for such a task, because we can perform separate simulations on different cores. In our application the GPU accelerator works as a computer farm executing separate instances of the same program, so we have not used advanced CUDA environment features, such as dedicated numerical libraries or texture processing. In the considered problem the maximum speedup is expected if the number of separate tasks does not exceed the maximum number of threads allowed to run in parallel. The graphical illustration of subsequent operations executed on the host PC and on the TESLA GPU accelerator is presented in Figure 2. The operations are marked with the same number that is used in the comments on the attached source code. The source code includes three files:

1. **`main.cu`** – the main program code containing the **int main(int argc,char **argv)** function called by operating system, and controlling communication between host PC and GPU accelerator
2. **`my_defs.h`** – header file with data type definitions,
3. **`kernel.cu`** – parallel code of computations executed as multiple GPU threads.

The program starts on the host PC CPU (step 1), with allocation of the input and output data arrays in the host computer memory (2). Next, copies of input and output data arrays are allocated in GPU accelerator memory (3). This is done by calling the **`cudaMalloc`** function from the CUDA library. The input arrays are filled with data on the host computer (4) and the data are transferred (5) into GPU using the **`cudaMemcpy`** function. Before GPU computation starts, the threads have to be synchronized using the **`cudaThreadSynchronize`** function (6).

The individual parallel tasks are launched on separated GPU threads using the

```
myKernel<<<dimGrid,dimBlock>>>\\
    (data_gpu_in, data_gpu_out,no_of_records);
```

instruction (7). Here **`myKernel`** is the name of the function called in parallel on the GPU accelerator in many copies. A slow data transfer between the host computer and the GPU accelerator is a bottleneck of the data processing. In order to achieve the best performance myKernel refers to the data arrays allocated on the GPU accelerator (**`data_gpu_in, data_gpu_out`**) instead of using original arrays in the host computer memory (**`data_in, data_out`**). GPU threads in our code are enumerated with 4-dimensional index:(**`blockIdx.x`** , **`blockIdx.y`**, **`threadIdx.x`**, **`threadIdx.y`**). The dimGrid describes the maximum values of **`blockIdx.x`** and **`blockIdx.y`**. The dimBlock sets the maximum values of **`threadIdx.x`** and **`threadIdx.y`**. The myKernel function maps the 4-dimensional thread index into one dimensional index of input data array using the following formula:

```
int index = (gridDim.x*blockDim.x*blockDim.y)
        *blockIdx.y + (gridDim.x*blockDim.x)
        *threadIdx.y + blockDim.x*blockIdx.x+
            threadIdx.x;
```

If the index does not exceed the size of the input array, then myKernel launches the computations for the corresponding element of data_gpu_in array. Otherwise, the thread remains idle.
The numerical integration of differential equations is calculated on GPU with a forth-order Runge-Kutta integration scheme [11] (see the code of the **`calc_evolution`** function). After completing the required number of time steps the final values of intermembrane potentials are written into the **`data_gpu_out`** array on the GPU accelerator. Thread synchronization (9) is required before collecting data obtained

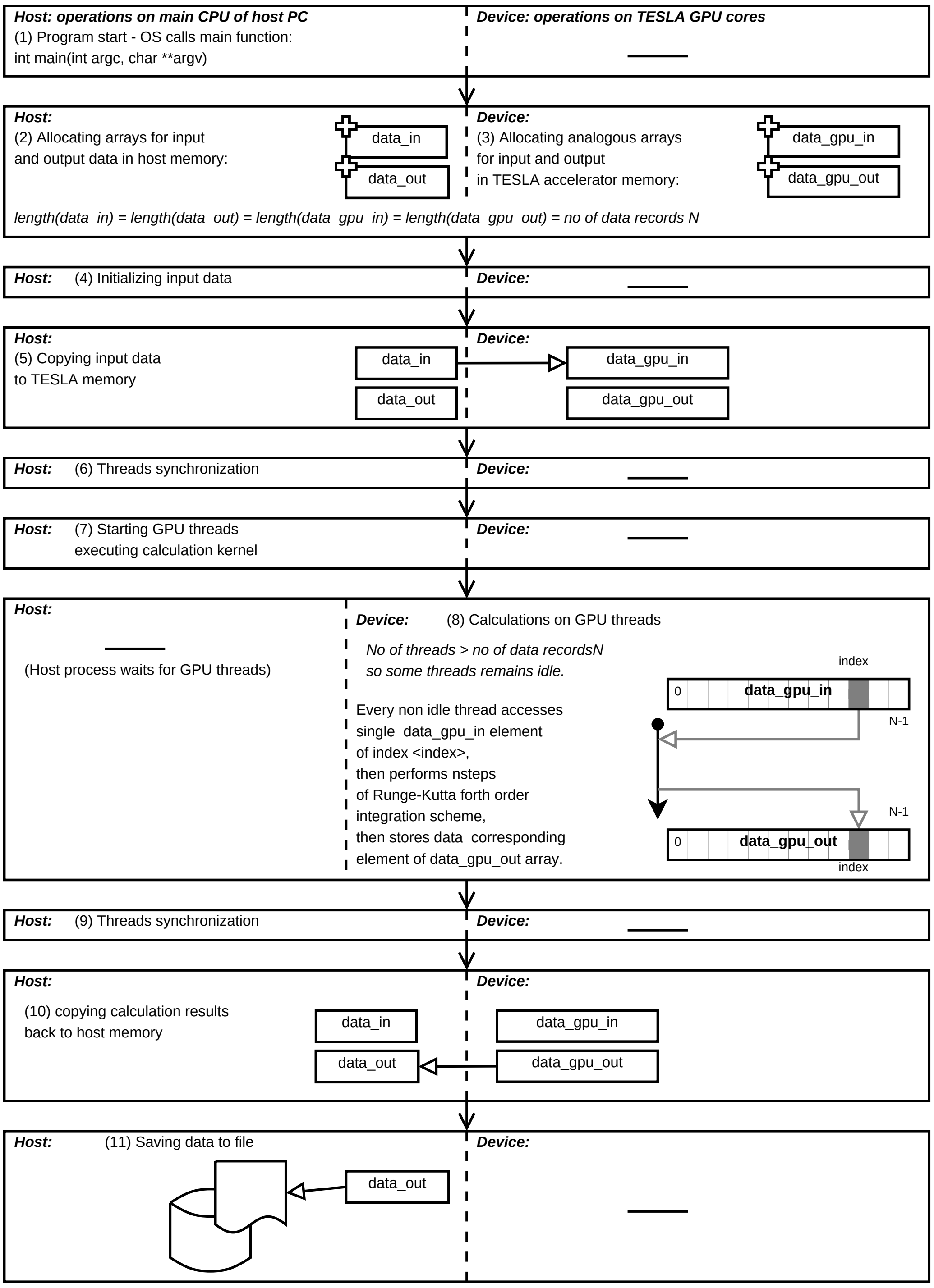
Host: operations on main CPU of host PC
(1) Program start - OS calls main function:
int main(int argc, char **argv)
Device: operations on TESLA GPU cores
Host:
(2) Allocating arrays for input and output data in host memory:
data_in
data_out
Device:
(3) Allocating analogous arrays for input and output in TESLA accelerator memory:
data_gpu_in
data_gpu_out
length(data_in) = length(data_out) = length(data_gpu_in) = length(data_gpu_out) = no of data records N
Host: (4) Initializing input data
Device:
Host:
(5) Copying input data to TESLA memory
Device:
Host: (6) Threads synchronization
Device:
Host: (7) Starting GPU threads executing calculation kernel
Device:
Host:
(Host process waits for GPU threads)
Device: (8) Calculations on GPU threads
No of threads > no of data recordsN so some threads remains idle.
Every non idle thread accesses single data_gpu_in element of index <index>, then performs nsteps of Runge-Kutta forth order integration scheme, then stores data corresponding element of data_gpu_out array.
index
0
N-1
Host: (9) Threads synchronization
Device:
Host:
(10) copying calculation results back to host memory
Device:
Host: (11) Saving data to file
Device:

Fig. 2 GPU optimization of multiple identical scalar calculations for different initial data. The most important numerations are numbered. The same enumeration is used in source code comments. The host process reads initial data and parameters, distributes these values in the GPU device memory, and launches appropriate number of GPU device threads. Each GPU thread performs simulation for a single initial data set.

on different threads. The results are copied back to the host memory with the cudaMemcpy function (10) and the results are saved to an output file (11).

IV. RESULTS

The example results illustrating potential φ_{ap}-φ_{in} between the apical side and the cell interior are shown in Figures 3-9. The calculated potential can be considered as a function of four variables – concentrations of ions K$^+$, Na$^+$, Cl$^-$, HCO$_3^-$ at apical side:
$\phi_{ap} - \phi_{in} = f(X_{K^+,ap},\ X_{Na^+,ap},\ X_{Cl^-,ap},\ X_{HCO_3^-;ap};$ *other parameters)*, where *other parameters* are fixed parameters of the system (for example ion permeabilities).

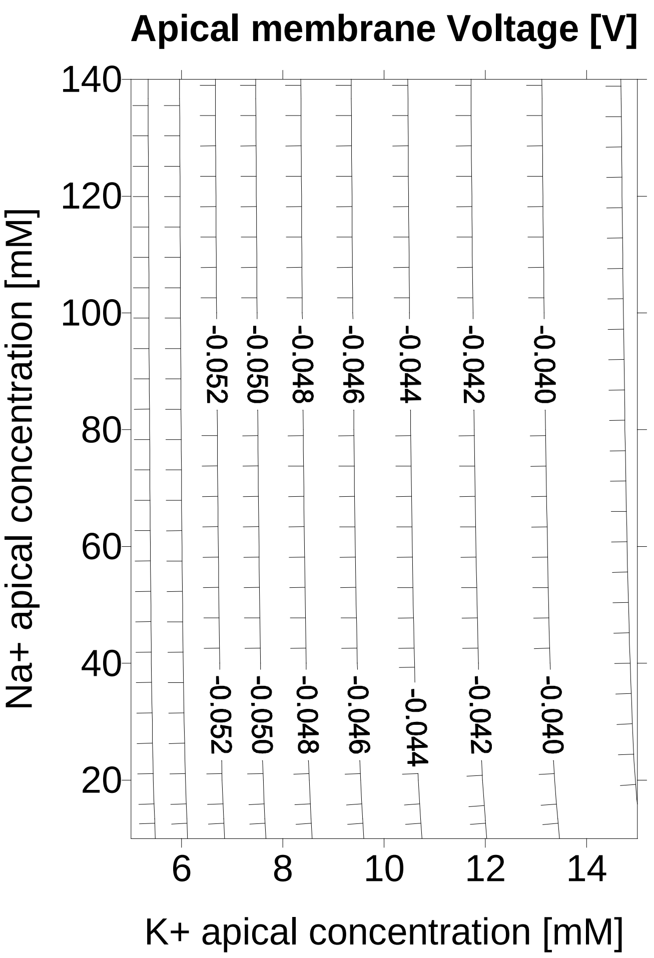

Fig. 3 The potential between cell interior and apical side as a function of concentrations of K$^+$ and Na$^+$ ions at apical side. Concentrations of Cl$^-$, HCO$_3^-$ remains constant and are equal: $X_{Cl^-,ap}$*=100 mM,* $X_{HCO_3^-,ap}$*=24 mM*.

We have calculated this potential on slices of this 4-dimensional space corresponding to a selected pair of concentration varying, with the remaining ion concentrations constant. Figures 3-8 show contour plots of potentials values of the six available slices. All the slices contain common point
$X_{K^+,ap}$*=10 mM,* $X_{Na^+,ap}$*=100 mM*
$X_{Cl^-,ap}$*=100 mM,* $X_{HCO_3^-,ap}$*=24 mM,*
with the potential value φ_{ap}-φ_{in} = *-0.048 V.*
Figure 9 shows the isosurface plot of the mentioned potential as a function of 3 variables: concentrations of K$^+$, Na$^+$, Cl$^-$ ions, with concentration of HCO$_3$treated as a constant parameter $X_{HCO_3^-,ap}$*=24 mM.*

Tab. 1 Membrane flow model parameters used to obtain results shown in Figures 3-9.

Parameter	Value	Unit
Simulated ion concentrations at apical side		
$X_{K+-,ap}$	5-15, step 1	mM
$X_{Na^+,ap}$	10-140, step 10	mM
$X_{Cl^-,ap}$	10-140, step 10	mM
$X_{HCO_3^-,ap}$	20-28, step 1	mM
Common reference point of simulated ion concentrations at apical side		
$X_{K+-,ap}$	10	mM
$X_{Na^+,ap}$	100	mM
$X_{Cl^-,ap}$	100	mM
$X_{HCO_3^-,ap}$	24	mM
Initial ion concentrations in cell interior		
$X_{K^+,in}$	100	mM
$X_{Na^+,in}$	10	mM
$X_{Cl^-,in}$	10	mM
$X_{HCO_3^-,in}$	20	mM
Temperature		
T	300	K
Apical side permeability (reciprocal resistance)		
$G_{K^+,ap}$	$5 \cdot 10^{-4}$	Ω^{-1}
$G_{Na^+,ap}$	$5 \cdot 10^{-6}$	Ω^{-1}
$G_{Cl^-,ap}$	$5 \cdot 10^{-5}$	Ω^{-1}
$G_{HCO_3^-,ap}$	$5 \cdot 10^{-5}$	Ω^{-1}
Apical side capacity		
C_{ap}	10^{-6}	F
Cell volume		
Vol	10^{-9}	m3

Over 20 different values of each concentration were used to make a plot, see Table 1. As we can see, potassium ions have the biggest contribution to the mentioned potential value, as a result of big permeability of membrane ions used in our simulation. The effective permeability of membrane for K$^+$ ions was selected 100 times bigger than for Na$^+$ ions, and 10 times bigger than for Cl^- and $HCO_3{}^-$. The average time of calculations on NVIDIA TESLA C870 graphics accelerator was about **3 seconds**, compared to about **40 minutes** for a scalar computation and the program compiled by Free Pascal Compiler on AMD Athlon 2 GHz. The GPU-optimized version of the program was tested on a PC with Intel Pentium D 940 3.2 GHz 64-bit processor and the TESLA C870 accelerator installed. Other tests were done on hosts with different types of CPU. They have shown that the host CPU has no influence on the execution time of the listed program because all numerical operations were performed on the TESLA card. The host processor controlled only input/output operations. I have obtained a speed improvement of the order of 1000 compared to the scalar program**.** Such speed-up is related to the large number of independent kernels as well as better computational performance and communication with local memory of NVIDIA GPU cores compared to conventional CPU. I have performed speed scalability tests on the TESLA C1060 accelerator that allows for double precision for floating point variables. Figure 10 shows the total execution time of the GPU optimized program as a function of the number of calculated records (measured by the time command line tool). In this example the range of input Na^+ and K^+ concentrations was identical whereas the densities of probing were different. The number of time simulation steps for all records was constant and equal to 10000. As we can see, the calculation times are almost equal if the number of points does not exceed 4000. If the number of records is larger the time is growing linearly. It is the result of limit for the GPU threads running simultaneously on one TESLA C1060 accelerator.

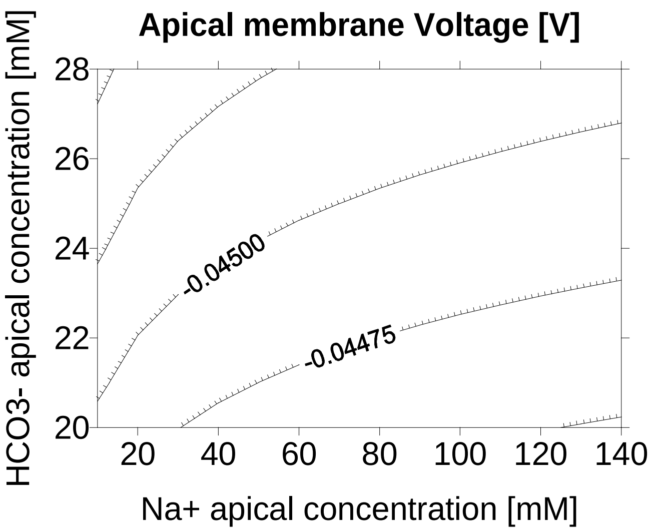

Fig. 4 The potential between cell interior and apical side as a function of concentrations of Na^+ and $HCO_3{}^-$ ions at apical side. Concentrations of K^+, Cl^- remains constant and equal $X_{K^+,ap}$=10 mM, $X_{Cl^-,ap}$=100 mM.

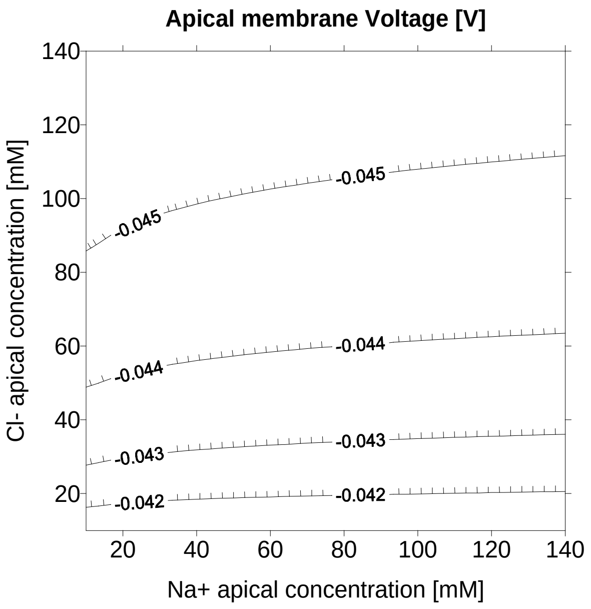

Fig. 5 The potential between cell interior and apical side as a function of concentrations of Na^+ and Cl^- ions at apical side. Concentrations of K^+, $HCO_3{}^-$ remains constant and equal $X_{K^+,ap}$=10 mM, $X_{HCO_3-,ap}$=24 mM.

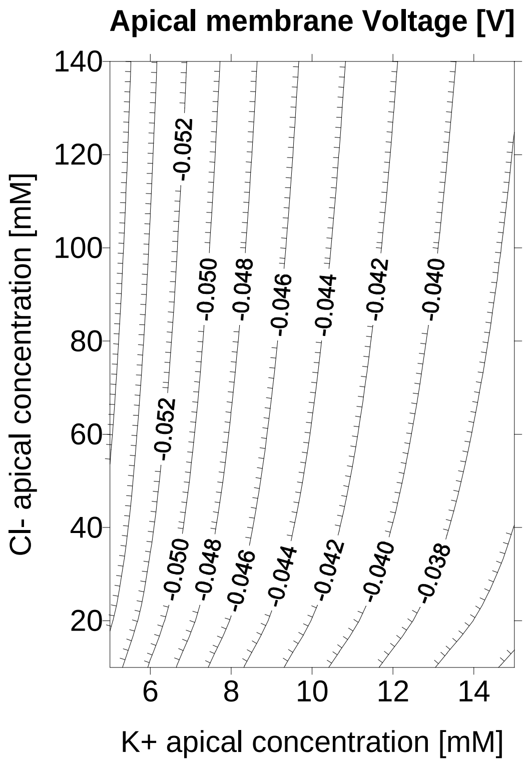

Fig. 6 The potential between cell interior and apical side as a function of concentrations of K^+ and Cl^- ions at apical side. Concentrations of Na^+, $HCO_3{}^-$ remains constant and equal $X_{Na^+,ap}$=100 mM, $X_{HCO_3-,ap}$=24 mM.

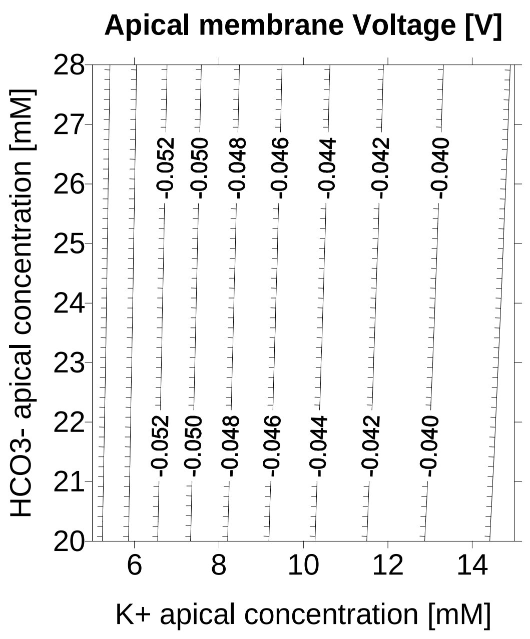

Fig. 7 The potential between cell interior and apical side as a function of concentrations of K^+ and $HCO_3{}^-$ ions at apical side. Concentrations of Na^+, Cl^- remains constant and equal $X_{Na^+,ap}$=100 mM, $X_{Cl^-,ap}$=100 mM.

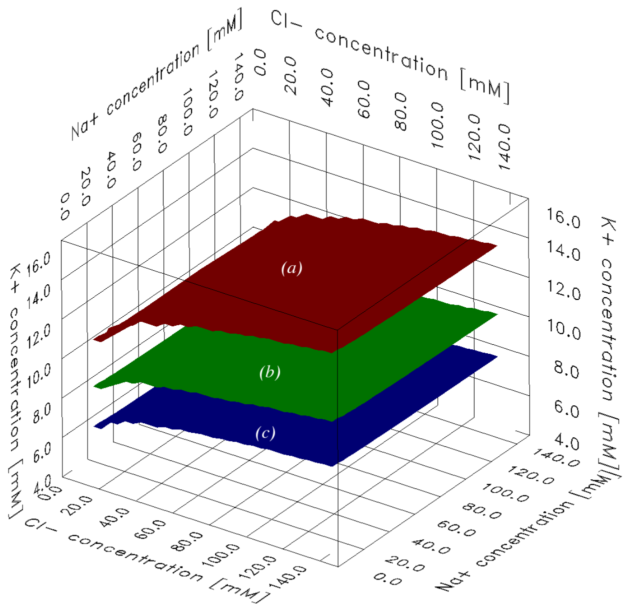

Fig. 9 the isosurface plot of potential between cell interior and apical side as a function of concentrations of k^+, $na^+$and cl^- ions at apical side. three isosurfaces of φ_{ap}-φ_{in} potential values corresponding to -0.050 v (blue, bottom), -0.045 v (green, middle) and -0.040 v (red, top) are shown. concentrations of $hco_3{}^-$ remains constant and equal $x_{hco3-,ap}$=24 mm.

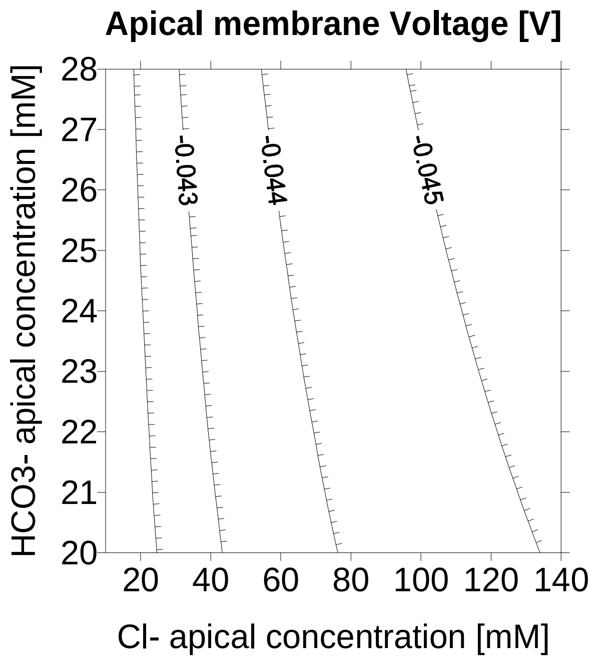

Fig. 8 The potential between cell interior and apical side as a function of concentrations of Cl^- and $HCO_3{}^-$ ions at apical side. Concentrations of K^+, Na^+ remains constant and equal $X_{Cl^-,ap}$=100 mM, $X_{HCO_3-,ap}$=24 mM.

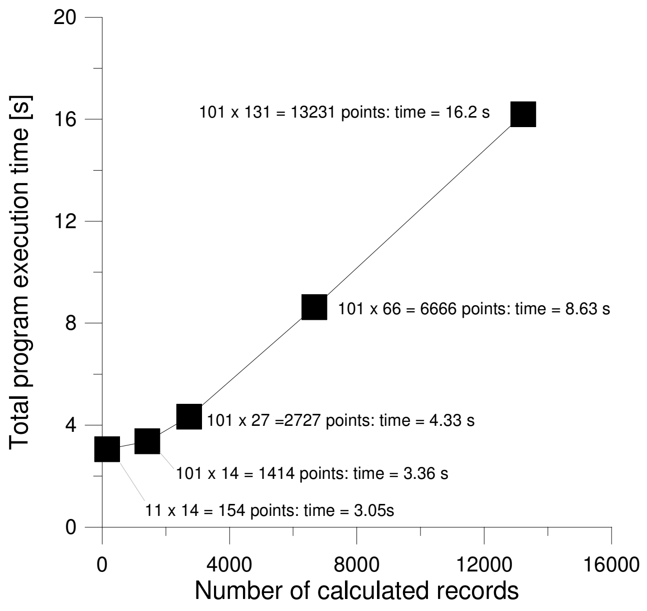

Fig. 10 Tests of scalability of calculations on the TESLA C1060 accelerator with double precision

V. CONCLUSIONS

We have developed a tool for modeling membrane ion flows working on NVIDIA graphics accelerators. The program increases the speed of calculations over 1000 times if compared with our previous approach running on a scalar CPU.

Our GPU program may be very useful for membrane model parameterization and its experimental verification. Thanks to its speed we can optimize model parameters like membrane permeabilities and capacities using a large number of experimental data. Moreover, the code can be easily modified to other forms of differentials equations.

The model reported above treats the membrane as a single entity without focusing on particular ion channels. It can be easily generalized for a specific type membrane by modification of $I_{ION}(src \rightarrow trg)$ terms. We can consider different current-voltage characteristics of specific protein channels by selecting an appropriate $I_{ION}(src \rightarrow trg)$ term form.

The speed achieved on GPU accelerators seems to be sufficient for the non-local model of membrane transport. In such models channels of different types are spatially distributed on the membranes and the equilibration process involves local currents flowing inside the cell.

Acknowledgments

The work was supported by the Ministry of Science and Higher Education Grant No 1828/B/PO1/2010/39.

References

[1] L. Stryer, *Biochemistry*. W. H. Freeman, StateplaceNew York, 1981.

[2] D. C. Gadsby, P. Vergani, L. Csanády, *The ABC protein turned chloride channel whose failure causes cystic fibrosis.* Nature 7083, 477–83 (2006).

[3] I. Scheffer, S. Berkovic, *Generalized epilepsy with febrile seizures plus. A genetic disorder with heterogeneous clinical phenotypes*. Brain 120, 479–90 (1997).

[4] S. A. Goldstein, C. Miller, *Mechanism of charybdotoxin block of a voltage gated K+ channel*. Biophysical Journal 65, 1613–1619 (1993).

[5] S. Candia, M. L. Garcia, R. Latorre, *Mode of action of iberiotoxin, a potent blocker of the large conductance Ca(2+)-activated K+ channel.*Biophysical Journal 63, 583–590 (1992).

[6] R. Toczylowska-Maminska, K. Dolowy, *Ion transporting proteins of human bronchial epithelium.* Journal of Cellular Biochemistry 113, 426-432 (2012).

[7] C. V. Falkenberg, E. Jakobsson, *A Biophysical Model for Integration of Electrical, Osmotic, and pH Regulation in the Human Bronchial Epithelium*. Biophysical Journal 98,1476–1485 (2010).

[8] Y. Sohma, M. A. Gray, Y. Imai, B. E. Argent, *$HCO_3{}^-$ Transport in a Mathematical Model of the Pancreatic Ductal Epithelium*. Journal of Membrane Biology 176,77–100 (2000).

[9] S. H. Wright, *Generation of resting membrane potential.*Advances in Physiology Education 28, 139-142 (2004).

[10] NVIDIA corporation, 2012. *CUDA C Programming Guide* Available from: `http://developer.NVIDIA.com/NVIDIA-gpu-computing-documentation` Accesed: Jul 11, 2012

[11] W. H. Press, S. A. Teukolsky, W. T. Vetterling and B. P. Flannery, chapter 16.1 in *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, 1993.

SUPPLEMENTARY MATERIALS

Listing 1 Listing of source file main.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <fstream>
#include <time.h>
#include <string.h>
#include <cutil_inline.h>

#include "my_defs.h"

using namespace std;

//////////////////////
// Data configuration
```

```
//////////////////////
// DATA_N - number of independent data records
const int    DATA_N = 100000;

// Input data size in bytes
const int   DATA_SZ = DATA_N * sizeof(InputData);

// Output data size in bytes
const int RESULT_SZ = DATA_N * sizeof(OutputData);

//Ion concentrations in mM (milimoles/litre)
//K = K+, Na = Na+, Cl = Cl-, Bi = HCO_3-
// ap - apical, in - cell interior, bl -
    basolateral
const double K_in  = 140.;
const double K_bl  = 5.;

const double Na_in = 5.;
const double Na_bl = 140.;

const double Cl_ap = 110.;
const double Cl_in = 10.;
const double Cl_bl = 110.;
const double Bi_ap = 24.;
const double Bi_in = 20.;
const double Bi_bl = 24.;

// Selected ion concentration are varying in some
    range
// Here variables are K+ and Na+ on apical side
const double K_ap_begin = 5.;
const double K_ap_end = 15.;
const double K_ap_step = 0.1;

const double Na_ap_begin = 10.;
const double Na_ap_end = 140.;
const double Na_ap_step = 1.;

const double T = 300.; // K, temperature

const double dt = 1.e-3; // time step
const double nsteps = 10000; // no of steps


//////////////////////
// (1) OS calls main function: int main(int argc,
    char **argv)
//////////////////////
int main(int argc, char **argv){

    InputData *data_in, *data_gpu_in;
    OutputData *data_out, *data_gpu_out;

    int i= 0;
    // (2) Allocating arrays for input and output
        data in host memory:
    printf("...allocating␣CPU␣memory.\n");
    data_in     = (InputData *)malloc(DATA_SZ);
    data_out     = (OutputData *)malloc(RESULT_SZ)
        ;
// (3) Allocating analogous arrays for input and
    output in TESLA accelerator memory:

    printf("...allocating␣GPU␣memory.\n");
    cudaMalloc((void **)&data_gpu_in, DATA_SZ);
    cudaMalloc((void **)&data_gpu_out, RESULT_SZ);

    printf("...generating␣input␣data␣in␣CPU␣mem.\n
        ");

    for(i = 0; i < DATA_N; i++){
        data_in[i].initialized = false;
    }



    i = 0;
// (4) Initializing input data

    for(double K_ap_var = K_ap_begin;
     K_ap_var <= K_ap_end;
        K_ap_var += K_ap_step) {
      for(double Na_ap_var = Na_ap_begin;
          Na_ap_var <= Na_ap_end;
          Na_ap_var += Na_ap_step) {
         if (i >= DATA_N) break;
         data_in[i].initialized = true;

         data_in[i].K_ap  = K_ap_var;
         data_in[i].K_in  = K_in;
         data_in[i].K_bl  = K_bl;
         data_in[i].Na_ap = Na_ap_var;
         data_in[i].Na_in = Na_in;
         data_in[i].Na_bl = Na_bl;
         data_in[i].Cl_ap = Cl_ap;
         data_in[i].Cl_in = Cl_in;
         data_in[i].Cl_bl = Cl_bl;
         data_in[i].Bi_ap = Bi_ap;
         data_in[i].Bi_in = Bi_in;
         data_in[i].Bi_bl = Bi_bl;

         data_in[i].T = T;
         data_in[i].tmax = tmax;
         data_in[i].dt = dt;
         data_in[i].nsteps = nsteps;

         i++;
      }
    }

    int no_of_records = i;
//

    printf("...copying␣input␣data␣to␣GPU␣mem.\n");
    cudaMemcpy(data_gpu_in, data_in, DATA_SZ,
        cudaMemcpyHostToDevice);
    printf("Data␣init␣done.\n");

    dim3 dimBlock(8, 8);
    dim3 dimGrid(16, 16);

    printf("Executing␣GPU␣kernel...\n");
//

    cudaThreadSynchronize() ;
    //
    myKernel<<<dimGrid, dimBlock>>>(data_gpu_in,
        data_gpu_out, no_of_records);
    cudaThreadSynchronize();

    printf("Reading␣back␣GPU␣result...\n");
    //
    cudaMemcpy(data_out, data_gpu_out, RESULT_SZ,
        cudaMemcpyDeviceToHost);
    printf("Shutting␣down...\n");

    cudaFree(data_gpu_in);
    cudaFree(data_gpu_out);

    //
    ofstream outfile("results.txt");

    outfile << "#␣K_ap␣␣␣Na_ap␣␣␣V_ap-V_in␣␣␣␣V_ap
        ␣␣␣V_in" << endl;
    for(int j=0; j< DATA_N; j++) {
      if (data_in[j].initialized) {
        outfile
          << data_in[j].K_ap
          << "␣"
         << data_in[j].Na_ap
         << "␣"
         << data_out[j].V_ap - data_out[j].V_in
          << "␣"
          << data_out[j].V_ap
          << "␣"
          << data_out[j].V_in
       << endl;
      }
    }
    outfile.flush();
    outfile.close();
    free(data_in);
    free(data_out);
    cudaThreadExit();
}
```

Listing 2 Listing of source file my_defs.h

```
typedef struct {
  bool initialized;
  double K_ap,K_in,K_bl;
  double Na_ap,Na_in,Na_bl;
  double Cl_ap,Cl_in,Cl_bl;
  double Bi_ap,Bi_in,Bi_bl;
  double T;
  double tmax;
  int nsteps;
  double dt;
} InputData;

typedef struct {
  double V_ap,V_in;
  double tfin;
  bool calculated;
} OutputData;

__global__ void myKernel(InputData *data_in,
    OutputData *data_out, int elementN);
```

Listing 3 Listing of source file kernel.cu

```
#include "my_defs.h"

// number of independent variables of the model
const int NVARS = 14;
const double Faraday = 9.65e+4;  //{ C/mol,
    Faraday constant }
const double R = 8.31; //{ J/(mol*K), gas constant
    }

typedef struct {
  double T;
} Params;


// helper inline function for calculating the
    Nernst potential
// __device__ means that the function can be
    launched only from GPU
inline __device__ double V_Nernst(double T, int
    z_ion, double c_Source, double c_Target)
{
  return -R*T/Faraday/z_ion * log((c_Source/
      c_Target));
}

// declaration of function processing single data
    record
__device__ void calc_evolution(const InputData&
    data_in,  OutputData& data_out, double* y,
    double* y_fin);


// declaration of function calculating derivatives
     (right sides of equation)
inline __device__ void four_ions_flow(double t,
                    double *y,
                    Params& params,
                    double *yprime);


// declaration of the calculation kernel
// __global__ attribute means that this function
    can be launched from host PC
// in multiple instances on GPU accelerator
    threads
__global__ void myKernel(
  InputData *data_in,
  OutputData *data_out,
  int elementN)
{
  // Single GPU thread obtains single input data
      record
  // - element of data_in array of index
      determined by variable index.
  // The following formula defines unique mapping
      of multidimensional thread index
  // (blockIdx.x,blockIdx.y,threadIdx.x,threadIdx.
      y)
  // to array index:
  int index = (gridDim.x*blockDim.x*blockDim.y)*
      blockIdx.y
              + (gridDim.x*blockDim.x)*threadIdx.y
              + blockDim.x*blockIdx.x
              + threadIdx.x;
  // The results of calculations will be stored in
       element of data_out array
  // of index determined by variable index.

  double y[NVARS],y_fin[NVARS];

  data_out[index].calculated = true;

  // (8) If the number of threads is greater than
      number of data records,
  // some threads remain idle
  if (index<elementN) {
    // selected threads will process data_in[index
        ] to data_out[index]
    calc_evolution(data_in[index],  data_out[index
        ], y, y_fin);
  }

}


// definition of calc_evolution function
__device__ void calc_evolution(const InputData&
    data_in,  OutputData& data_out,
                                double* y, double*
                                     y_fin)
{
  // additional validating if data record was
      initialized
  if (! data_in.initialized) return;

  Params params;
  params.T = data_in.T;
  int nsteps = data_in.nsteps;
  double dt = data_in.dt;

  double y_aux[NVARS]; // auxiliary arrays for
      Runge-Kutta algorithm
  double a[NVARS];
  double b[NVARS];
  double c[NVARS];
  double d[NVARS];

  // Model variables - initial values (time = 0 s)
  y[0] = 0.0f; // Electric potential of cell
      interior V_in
  y[1] = 0.0f; // Electric potential of apical
      side V_ap
  y[2] = data_in.K_ap; // Ion concentrations
  y[3] = data_in.K_in;
  y[4] = data_in.K_bl;
  y[5] = data_in.Na_ap;
  y[6] = data_in.Na_in;
  y[7] = data_in.Na_bl;
  y[8] = data_in.Cl_ap;
  y[9] = data_in.Cl_in;
  y[10] = data_in.Cl_bl;
  y[11] = data_in.Bi_ap;
  y[12] = data_in.Bi_in;
  y[13] = data_in.Bi_bl;


  double t=0;
  // performing nsteps of time simulations
  for(int i = 0 ; i < nsteps; i++) {
      // implementation of 4th order Runge-Kutta
           (RK4) integration
    t = t + dt;
    four_ions_flow(t, y, params, a); //{ a = fp(x,
        y) }
    for(int j=0; j<NVARS; j++) {
        a[j] *= dt;
    }

    for(int j=0; j<NVARS; j++) {
        y_aux[j] = y[j] + 0.5*a[j];
    }

    four_ions_flow(t + dt/2, y_aux, params, b); //
        { b = fp(x + dt / 2, y + a / 2);}
    for(int j=0; j<NVARS; j++) {
        b[j] *= dt;
    }

    for(int j=0; j<NVARS; j++) {
        y_aux[j] = y[j] + 0.5*b[j];
    }

    four_ions_flow(t + dt/2, y_aux, params, c); //
        { c = dt * fp(x + dt / 2, y + b / 2);}
    for(int j=0; j<NVARS; j++) {
        c[j] *= dt;
    }

    for(int j=0; j<NVARS; j++) {
        y_aux[j] = y[j] + c[j];
    }

    four_ions_flow(t + dt, y_aux, params, d); //{
        d = fp(x, y + c); }
    for(int j=0; j<NVARS; j++) {
        d[j] *= dt;
    }

    for(int j=0; j<NVARS; j++) {
        y[j] = y[j] + a[j]/6. + b[j]/3. + c[j]/3.
            + d[j]/6.;
    }

  }

  // collecting interesting variable values after
      simulation
  data_out.V_ap = y[0];
  data_out.V_in = y[1];
  data_out.tfin = t;
  data_out.calculated = true;
}

// definition of function calculating derivatives
inline __device__ void four_ions_flow(double t,
    double *y, Params& params, double *yprime) {
  // Some model parameters are hardcoded, but it
      is easy to move them to params record
  // Membrane permeabilities for selected ions
  const double G_K_ap = 5.e-6; //{ S/cm^2}
  const double G_Na_ap = 5.e-8; //{ S/cm^2}
  const double G_Cl_ap = 5.e-7; //{ S/cm^2}
  const double G_Bi_ap = 5.e-7; //{ S/cm^2}

  const double G_K_bl = 5.e-6; //{ S/cm^2}
  const double G_Na_bl = 5.e-8; //{ S/cm^2}
  const double G_Cl_bl = 5.e-7; //{ S/cm^2}
  const double G_Bi_bl = 5.e-7; //{ S/cm^2}

  // Membrane areas
  const double S_ap = 1.; //{cm^2}
  const double S_bl = 10.; //{cm^2}

#include "my_defs.h"

// number of independent variables of the model
const int NVARS = 14;
const double Faraday = 9.65e+4;  //{ C/mol,
    Faraday constant }
const double R = 8.31; //{ J/(mol*K), gas constant
    }

typedef struct {
  double T;
} Params;
```

```
// helper inline function for calculating the
    Nernst potential
// __device__ means that the function can be
    launched only from GPU
inline __device__ double V_Nernst(double T, int
    z_ion, double c_Source, double c_Target)
{
  return -R*T/Faraday/z_ion * log((c_Source/
      c_Target));
}

// declaration of function processing single data
    record
__device__ void calc_evolution(const InputData&
    data_in,  OutputData& data_out, double* y,
    double* y_fin);


// declaration of function calculating derivatives
     (right sides of equation)
inline __device__ void four_ions_flow(double t,
                    double *y,
                    Params& params,
                    double *yprime);


// declaration of the calculation kernel
// __global__ attribute means that this function
    can be launched from host PC
// in multiple instances on GPU accelerator
    threads
__global__ void myKernel(
  InputData *data_in,
  OutputData *data_out,
  int elementN)
{
  // Single GPU thread obtains single input data
      record
  // - element of data_in array of index
      determined by variable index.
  // The following formula defines unique mapping
      of multidimensional thread index
  // (blockIdx.x,blockIdx.y,threadIdx.x,threadIdx.
      y)
  // to array index:
  int index = (gridDim.x*blockDim.x*blockDim.y)*
      blockIdx.y
              + (gridDim.x*blockDim.x)*threadIdx.y
              + blockDim.x*blockIdx.x
              + threadIdx.x;
  // The results of calculations will be stored in
       element of data_out array
  // of index determined by variable index.

  double y[NVARS],y_fin[NVARS];

  data_out[index].calculated = true;

  // (8) If the number of threads is greater than
      number of data records,
  // some threads remain idle
  if (index<elementN) {
    // selected threads will process data_in[index
        ] to data_out[index]
    calc_evolution(data_in[index],  data_out[index
        ], y, y_fin);
  }

}
```

```
// definition of calc_evolution function
__device__ void calc_evolution(const InputData&
    data_in,  OutputData& data_out,
                                double* y, double*
                                    y_fin)
{
  // additional validating if data record was
      initialized
  if (! data_in.initialized) return;

  Params params;
  params.T = data_in.T;
  int nsteps = data_in.nsteps;
  double dt = data_in.dt;

  double y_aux[NVARS]; // auxiliary arrays for
      Runge-Kutta algorithm
  double a[NVARS];
  double b[NVARS];
  double c[NVARS];
  double d[NVARS];

  // Model variables - initial values (time = 0 s)
  y[0] = 0.0f; // Electric potential of cell
      interior V_in
  y[1] = 0.0f; // Electric potential of apical
      side V_ap
  y[2] = data_in.K_ap; // Ion concentrations
  y[3] = data_in.K_in;
  y[4] = data_in.K_bl;
  y[5] = data_in.Na_ap;
  y[6] = data_in.Na_in;
  y[7] = data_in.Na_bl;
  y[8] = data_in.Cl_ap;
  y[9] = data_in.Cl_in;
  y[10] = data_in.Cl_bl;
  y[11] = data_in.Bi_ap;
  y[12] = data_in.Bi_in;
  y[13] = data_in.Bi_bl;


  double t=0;
  // performing nsteps of time simulations
  for(int i = 0 ; i < nsteps; i++) {
      // implementation of 4th order Runge-Kutta
           (RK4) integration
    t = t + dt;
    four_ions_flow(t, y, params, a); //{ a = fp(x,
        y) }
    for(int j=0; j<NVARS; j++) {
        a[j] *= dt;
    }

    for(int j=0; j<NVARS; j++) {
        y_aux[j] = y[j] + 0.5*a[j];
    }

    four_ions_flow(t + dt/2, y_aux, params, b); //
        { b = fp(x + dt / 2, y + a / 2);}
    for(int j=0; j<NVARS; j++) {
        b[j] *= dt;
    }

    for(int j=0; j<NVARS; j++) {
        y_aux[j] = y[j] + 0.5*b[j];
    }
    four_ions_flow(t + dt/2, y_aux, params, c); //
        { c = dt * fp(x + dt / 2, y + b / 2);}
    for(int j=0; j<NVARS; j++) {
        c[j] *= dt;
    }

    for(int j=0; j<NVARS; j++) {
        y_aux[j] = y[j] + c[j];
    }

    four_ions_flow(t + dt, y_aux, params, d); //{
        d = fp(x, y + c); }
    for(int j=0; j<NVARS; j++) {
        d[j] *= dt;
    }

    for(int j=0; j<NVARS; j++) {
        y[j] = y[j] + a[j]/6. + b[j]/3. + c[j]/3.
            + d[j]/6.;
    }

  }

  // collecting interesting variable values after
      simulation
  data_out.V_ap = y[0];
  data_out.V_in = y[1];
  data_out.tfin = t;
  data_out.calculated = true;
}


// definition of function calculating derivatives
inline __device__ void four_ions_flow(double t,
    double *y, Params& params, double *yprime) {
  // Some model parameters are hardcoded, but it
      is easy to move them to params record
  // Membrane permeabilities for selected ions
  const double G_K_ap = 5.e-6; //{ S/cm^2}
  const double G_Na_ap = 5.e-8; //{ S/cm^2}
  const double G_Cl_ap = 5.e-7; //{ S/cm^2}
  const double G_Bi_ap = 5.e-7; //{ S/cm^2}

  const double G_K_bl = 5.e-6; //{ S/cm^2}
  const double G_Na_bl = 5.e-8; //{ S/cm^2}
  const double G_Cl_bl = 5.e-7; //{ S/cm^2}
  const double G_Bi_bl = 5.e-7; //{ S/cm^2}

  // Membrane areas
  const double S_ap = 1.; //{cm^2}
  const double S_bl = 10.; //{cm^2}

  // Membrane capacities per area unit
  const double c_ap = 1.; //{uF/cm^2}
  const double c_bl = 1.; //{uF/cm^2}
  // Volume of cell layer
  const double Vol = 1.e-9; //{m^3}
  // Ion signs (charges in elementary charge e
      units)
  const int z_K = 1;
  const int z_Na = 1;
  const int z_Cl = -1;
  const int z_Bi = -1;

  double Temp, phi_ap, phi_in ;

  double K_ap,K_in,K_bl,Na_ap,Na_in,Na_bl;
  double Cl_ap,Cl_in,Cl_bl,Bi_ap,Bi_in,Bi_bl;
  double J_K_in2ap,J_Na_in2ap,J_Cl_in2ap,
      J_Bi_in2ap;
  double J_K_bl2in,J_Na_bl2in,J_Cl_bl2in,
      J_Bi_bl2in;
  double delta_phi_in2ap,delta_phi_bl2in;

  Temp = params.T;

  phi_ap = y[0];
  phi_in = y[1];

  K_ap  = y[2];
  K_in  = y[3];
  K_bl  = y[4];
  Na_ap = y[5];
  Na_in = y[6];
  Na_bl = y[7];
  Cl_ap = y[8];
  Cl_in = y[9];
  Cl_bl = y[10];
  Bi_ap = y[11];
  Bi_in = y[12];
  Bi_bl = y[13];

  //Calculation of ion current from cell interior
      to apical side, In -> Ap}
  J_K_in2ap  = S_ap*G_K_ap *(phi_in-phi_ap -
      V_Nernst(Temp,z_K, K_in,K_ap));
  J_Na_in2ap = S_ap*G_Na_ap*(phi_in-phi_ap -
      V_Nernst(Temp,z_Na, Na_in,Na_ap));
  J_Cl_in2ap = S_ap*G_Cl_ap*(phi_in-phi_ap -
      V_Nernst(Temp,z_Cl, Cl_in,Cl_ap));
  J_Bi_in2ap = S_ap*G_Bi_ap*(phi_in-phi_ap -
      V_Nernst(Temp,z_Bi, Bi_in,Bi_ap));

  //Calculation of ion current from basolateral
      side to cell interior , Bl -> In}
  J_K_bl2in  = S_bl*G_K_bl *(-phi_in - V_Nernst(
      Temp,z_K, K_bl,K_in));
  J_Na_bl2in = S_bl*G_Na_bl*(-phi_in - V_Nernst(
      Temp,z_Na, Na_bl,Na_in));
  J_Cl_bl2in = S_bl*G_Cl_bl*(-phi_in - V_Nernst(
      Temp,z_Cl, Cl_bl,Cl_in));
  J_Bi_bl2in = S_bl*G_Bi_bl*(-phi_in - V_Nernst(
      Temp,z_Bi, Bi_bl,Bi_in));

  // Ion flow causes change of charging states of
      membrane
  // (charging of capacitors); 1.e6 coefficient
      due to c_ap in microfarads
  delta_phi_in2ap = 1.e6*(J_K_in2ap+J_Na_in2ap+
      J_Cl_in2ap+J_Bi_in2ap)/S_ap/c_ap;
  delta_phi_bl2in = 1.e6*(J_K_bl2in+J_Na_bl2in+
      J_Cl_bl2in+J_Bi_bl2in)/S_bl/c_bl;

  // Vbl = 0 V (reference potential)
  // Vap = potential on basolateral membrane +
      potential on apical membrane
  yprime[0] = delta_phi_in2ap + delta_phi_bl2in;
  // Vin = potential on basolateral membrane
  yprime[1] = delta_phi_bl2in;

  // Ion concentrations speed - only concentration
       in cell interior are changing
  // (we are assuming infinite volume of apical
      and basolateral side)
  // K+
  yprime[2] = 0.; // Ap, apical side
  yprime[3] = (J_K_bl2in - J_K_in2ap)/Faraday/z_K/
      Vol; // In, cell Interior
  yprime[4] = 0.; // Bl, basolateral side

  // Na+
  yprime[5] = 0.; // Ap, apical side
  yprime[6] = (J_Na_bl2in - J_Na_in2ap)/Faraday/
      z_Na/Vol; // In, cell Interior
  yprime[7] = 0.; // Bl, basolateral side

  // Cl-
  yprime[8]  = 0.; // Ap, apical side
  yprime[9] = (J_Cl_bl2in - J_Cl_in2ap)/Faraday/
      z_Cl/Vol; // In, cell Interior
  yprime[10] = 0.; // Bl, basolateral side

  // HCO_3-
  yprime[11] = 0.; // Ap, apical side
  yprime[12] = (J_Bi_bl2in - J_Bi_in2ap)/Faraday/
      z_Bi/Vol; // In, cell Interior
  yprime[13] = 0.; // Bl, basolateral side
}
```

Adam Górecki, Ph.D., was born in Warsaw, Poland in 1978. He studied physics at the Faculty of Physics, Warsaw University, where he received his M.Sc. degree in theoretical physics in 2002. In the years 2002-2008 he was a PhD student at the Division of Biophysics, Institute for Experimental Physics, Warsaw University and received the degree in computational biophysics in 2008. He is an assistant professor at the Department of Physics, Faculty of Wood Technology, Warsaw University for Life Sciences -SGGW (http://kf.sggw.pl) since October 2010. He is interested in applications of computational methods of biophysics. His present fields of research are numerical models of biological membranes and CUDA programming.