Journal of Neuroscience Methods 206 (2012) 183–194

Contents lists available at SciVerse ScienceDirect

Journal of Neuroscience Methods

journal homepage: www.elsevier.com/locate/jneumeth

Computational Neuroscience

Optimizing ion channel models using a parallel genetic algorithm on graphical processors

Roy Ben-Shalom, Amit Aviv, Benjamin Razon[1], Alon Korngreen*

The Mina and Everard Goodman Faculty of Life Sciences and the Leslie and Susan Gonda Multidisciplinary Brain Research Center, Bar-Ilan University, Ramat Gan 52900, Israel

ARTICLE INFO

Article history:
Received 6 June 2011
Received in revised form 25 February 2012
Accepted 28 February 2012

Keywords:
Voltage-gated channels
Neuron
Graphic card
CUDA
Parallel computation
GPU
GPGPU
Data fitting
Genetic algorithm

ABSTRACT

We have recently shown that we can semi-automatically constrain models of voltage-gated ion channels by combining a stochastic search algorithm with ionic currents measured using multiple voltage-clamp protocols. Although numerically successful, this approach is highly demanding computationally, with optimization on a high performance Linux cluster typically lasting several days. To solve this computational bottleneck we converted our optimization algorithm for work on a graphical processing unit (GPU) using NVIDIA's CUDA. Parallelizing the process on a Fermi graphic computing engine from NVIDIA increased the speed ~180 times over an application running on an 80 node Linux cluster, considerably reducing simulation times. This application allows users to optimize models for ion channel kinetics on a single, inexpensive, desktop "super computer," greatly reducing the time and cost of building models relevant to neuronal physiology. We also demonstrate that the point of algorithm parallelization is crucial to its performance. We substantially reduced computing time by solving the ODEs (Ordinary Differential Equations) so as to massively reduce memory transfers to and from the GPU. This approach may be applied to speed up other data intensive applications requiring iterative solutions of ODEs.

1. Introduction

Ion channels are trans-membrane proteins that open and close, leading to a change in ion flow across the membrane. They may be opened or closed by a number of factors, one of them being changes in membrane potential. Voltage-dependent ion channels display highly complex kinetics requiring intricate kinetic models in order to understand their behavior. One paradigm describing ion channel kinetics is based on the seminal experiments of Hodgkin and Huxley (Hodgkin and Huxley, 1952a,b,c,d; Hodgkin et al., 1952). Their detailed kinetic models, derived from the giant axon of *Loligo*, are still extremely useful and the model is still widely used in simulations of neuronal physiology. However, since the development of single-channel recording techniques (Hamill et al., 1981), ion channel kinetics are usually described using Markov chain models (Sakmann and Neher, 1995).

The massive increase in computing power in recent decades has facilitated the creation of efficient programs for single channel analysis (Colquhoun and Hawkes, 1995; Colquhoun and Sigworth, 1995; Qin et al., 1996, 1997). Valuable tools have been created for analyzing currents recorded from entire cells containing many channels (Qin et al., 1996, 1997; Willms et al., 1999). We recently proposed a method for analyzing whole-cell recordings of voltage-gated channels and demonstrated that the viability of models of voltage-dependent ion channels can be verified using a genetic optimization algorithm (GA) concurrently with a global fit of experimental data to the model. Our method can be applied to all types of voltage-clamp recordings from different neuron classes (Gurkiewicz and Korngreen, 2007; Gurkiewicz et al., 2011).

Due to the nature of the genetic algorithm our analysis depends on the availability of high performance Linux clusters or similar parallel supercomputing environments, making it expensive. Furthermore, even on our 160-node cluster, it may take several days to carry out a single optimization of complex models using many recorded traces as a target data set. We thus sought to improve the compute/cost ratio of our method. Recent technological developments allow the use of graphical processing units (GPUs) for general purpose computing (GPGPU). Using the inherent parallel nature of GPUs may speed up many scientific and technological calculations by several orders of magnitude (John et al., 2009; Liu et al., 2009; Manavski and Valle, 2008; Nageswaran et al., 2009; Pinto et al., 2009; Rossant et al., 2011; Stone et al., 2007; Won-Ki et al., 2010). Here we describe the migration of our analysis technique from a Linux cluster to a GPU. We detail the logic and syntax of the new application and present benchmarks of its computing speed on two different GPU units. This new application greatly reduces both the computation time and costs of our analysis algorithm.

* Corresponding author. Tel.: +972 3 5318224; fax: +972 3 5352184.
E-mail address: alon.korngreen@biu.ac.il (A. Korngreen).
[1] Current address: Department of Computer Sciences, California Institute of Technology, USA.

0165-0270/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.jneumeth.2012.02.024

2. Methods

2.1. Markov models of voltage-gated channels

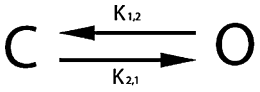
In a Markov chain model of ion channels, transition rates define the interstate dynamics. These rates can be constant or can depend on environmental variables such as the membrane potential or the concentration of a neurotransmitter. The probabilities of each state in the model are calculated by solving the differential equation.

$$\frac{d\vec{y}}{dt} = Q\vec{y} \tag{1}$$

where $\vec{y}$ is a vector of state probabilities and Q refers to the Q matrix (Colquhoun and Hawkes, 1995). The differential equation system has to be solved every time step and the outcome is the new probability.

2.1.1. Model A – Potassium Ion Channel – C–O

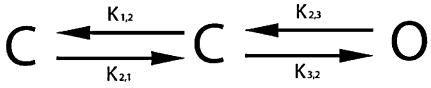
This model consists of 2 states defined by 5 free parameters; 4 parameters define the transition states and one parameter represents maximum conductance.



$$g_K = \bar{g}_K \cdot O$$
$$k_{i,j} = a_{i,j} \exp\left(z_{i,j} \cdot v_m\right)$$
$$k_{j,i} = a_{j,i} \exp\left(-z_{j,i} \cdot v_m\right)$$

where a_{ij} and z_{ij} are the free parameters used to set the transition value from state i to state j, v_m describes the voltage across the membrane, g_K is the conductance, $\bar{g}_K$ is the maximal conductance and O is the probability of the open state.

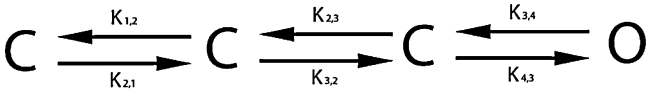
2.1.2. Model B – Potassium Ion Channel – C–C–O



$$g_K = \bar{g}_K \cdot O$$
$$k_{i,j} = a_{i,j} \exp\left(z_{i,j} \cdot v_m\right)$$
$$k_{j,i} = a_{j,i} \exp\left(-z_{j,i} \cdot v_m\right)$$

This model consists of 3 states with 9 free parameters; 8 define the transition states and 1 parameter represents maximum conductance.

2.1.3. Model C – Potassium Ion Channel – C–C–C–O



$$g_K = \bar{g}_K \cdot O$$
$$k_{i,j} = a_{i,j} \exp\left(z_{i,j} \cdot v_m\right)$$
$$k_{j,i} = a_{j,i} \exp\left(-z_{j,i} \cdot v_m\right)$$

This model consists of 4 states with 13 free parameters; 12 define the transition states and 1 parameter represents maximum conductance.

2.2. Generation of surrogate data

We have previously shown that to correctly optimize the model, the GA requires a large data set derived from several voltage-clamp protocols. All simulations presented in this study used surrogate data generated by a widely used set of electrophysiological voltage-clamp protocols (Gurkiewicz and Korngreen, 2006, 2007; Hodgkin and Huxley, 1952a,b,c,d; Hodgkin et al., 1952; Sakmann and Neher, 1995). These protocols did not change during the whole run of the genetic algorithm, only the parameters of the model varied between iterations. The kinetic parameters determined the probability of the model channel opening, resulting in ionic current.

A typical set of surrogate data generated from Model B is displayed in Fig. 1. Two standard voltage-clamp protocols were used. One set of voltage-clamp steps started from a holding potential of −100 mV followed by a step to various potentials from −80 to +40 mV. The activation of the conductance described by Model B by these steps is shown in Fig. 1A. The other set of voltage-clamp steps started from −100 mV, followed by a step to +40 mV and then by steps to various potentials starting at −110 mV. These deactivating changes to the conductance are displayed in Fig. 1B. All data presented in Fig. 1 were used for calculating the score of one individual in the population. The model's resulting current was quantitatively compared to the surrogate data using the sum-of-squares function (Eq. (2)). The surrogate data was subtracted from data generated by the same voltage-clamp protocols according to a parameter vector describing each individual in the population and the result was squared and summed.

$$\chi^2 = \frac{1}{MN}\sum_{i=1}^{M}\sum_{j=1}^{N}(I_{ij} - i_{ij}) \tag{2}$$

where N is the number of voltage traces in the protocol, M is the number of time points every trace, I_{ij} is the experimental or surrogate voltage-clamp current and i_{ij} is the simulated voltage-clamp current (Gurkiewicz and Korngreen, 2007; Gurkiewicz et al., 2011; Keren et al., 2005).

2.3. Genetic algorithm

A genetic algorithm (GA) is a search algorithm based on the mechanisms of Darwinian evolution. It uses random mutation, crossover and selection operators to breed better models or solutions (individuals) from an originally random starting population (Mitchell, 1996). Each individual in the population is described by a parameter set and the model is evaluated for each set. A search space was defined for each parameter to avoid parameter combinations causing instability to the set of differential equations while covering most of the physiological range expected for the parameters. For rate constants (k) the range was set from 0 to 2000 1/s, for voltage-dependence parameters (z) from zero to 2000 1/V, and zero to 100 pS for the conductance.

The population was sorted according to the value of the cost function of each individual (Eq. (2)) and a new generation was created using selection, crossover, and mutation as operators. Selection was achieved by a tournament (Tobias and Lothar, 1996) in which two pairs of individuals were randomly selected; the individual with the better score from each pair was transferred to the next generation. This procedure was repeated $N/2$ times (where N was the size of the population) until the new population was full. The one exception to this selection process (and later to the crossover and mutation operators) was the best individual, which was transferred unchanged to the next generation to prevent genetic drift.

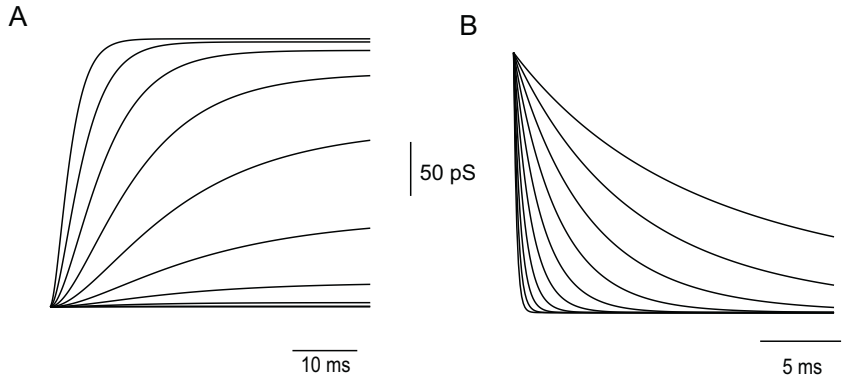


Fig. 1. A set of simulated voltage-clamp sweeps making up a data set for the optimization of a single individual. (A) Activation of the conductance of Model B simulated in response to depolarizing voltage steps from −40 to +60 mV in steps of 20 mV. (B) Deactivation of the conductance of Model B. Following a 20 ms activation to +60 mV, deactivation was simulated by lowering potential to from −120 to −20 mV in steps of 10 mV.

Each pair selected for transfer to the new population was subjected to a one-point crossover operator with a default probability of 0.1. Changes to this operator are discussed below. After the new population was created, each parameter value in the new population was subjected to mutation with a probability of 0.01. This allowed creation of double and even triple mutations in the same individual (albeit at a low frequency), thus increasing the variability in the new population. The mutation operator performed a substitution of the parameter value with a random value drawn from a flat random number distribution spanning the entire search space of the parameter. The flow of the algorithm is displayed in Fig. 2. Each individual of the population was a set of parameters for the model. The main loop of the algorithm halts when the score threshold is satisfied or when the number of iterations exceeds the maximum number of generations defined by the user.

2.4. Simulation environments

Three simulation environments were used here; two applied GPUs and one used a high performance Linux cluster. The high performance Linux cluster was the one that we used in our previous studies (Gurkiewicz and Korngreen, 2007; Gurkiewicz et al., 2011) and was used to parallelize the simulation environment NEURON (Hines and Carnevale, 2000, 1997). Briefly, NEURON was run on a Linux cluster with 160 CPUs sharing the same network file system via NFS. One of the machines functioned as a master, submitting and managing the jobs using the ParallelContext class of NEURON over an MPICH2 ring spanning the other computers (Gropp et al., 1999). Jobs were submitted via the Torque queue manager. Due to cluster workload we usually submitted jobs only to 80 CPUs.

Two computers with NVIDIA's GPUs were used here; one was a quad-core CPU with a GTX295 and the other, a dual-core CPU with a GTX480. The GTX295 is a double card containing two GTX280 units. All computations displayed in this paper used only one GTX280 unit. Both computers were installed with Windows 7 and Visual Studio 2010 (Microsoft Co.). Access to the GPU was achieved using the CUDA 4.0 extension of the C language (NVIDIA, 2011c). Note that the two GPUs have different architectures: GTX295 uses the Tesla architecture (NVIDIA, 2007) with a 1.3 compute capability (NVIDIA, 2011c) and GTX480 uses the Fermi architecture (NVIDIA, 2009) with a 2.0 compute capability (NVIDIA, 2011c). There are many advantages to using GPUs with the latter compute capability, such as larger shared memory for each block and more registers per block. Another advantage of the Fermi architecture

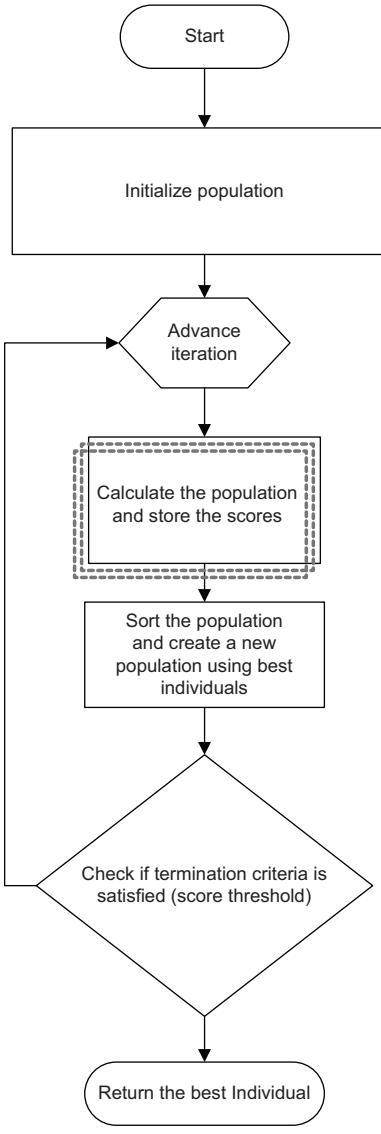


Fig. 2. Flow chart of the optimization algorithm model using a genetic algorithm (GA).

over the Tesla architecture is its ability to use better profiling, such as real time occupancy and extended debugging options. The code developed for both platforms is freely available from the authors.

The latest version of CUDA allowed us to develop the application using NVIDIA's Nsight (NVIDIA, 2011b). Nsight allows the developer to debug the code and to watch variables residing in the shared memory of the device. Nsight enables profiling the code in terms of occupancy (real time occupancy in Fermi cards), memory bandwidth, cache loads, etc. Nsight can produce a timeline of the computation, including transfer of variables between different memories and kernel launches. These help the developer see where optimizations are possible. Compiler flags can be easily controlled using the Nsight environment. These flags can greatly speed up operation by removing information on debugging and memory errors, by using faster math operations, and by optimization of host and device code.

2.5. CUDA and GPUs – thread hierarchy

GPU architecture is built around an array of multithreaded streaming multiprocessors (SMs). The GPU platform can accelerate applications if the application can be dispersed to many threads. Acceleration is achieved through using the Single Instruction Multiple Thread (SIMT) architecture that allows using many cores on a single GPU card. CUDA extends C by allowing the programmer to define kernels, these being C functions that run in parallel on the GPU N times by N different threads. Threads are ordered in blocks and blocks are ordered in grids; the size of a block and the grid are provided when a kernel is invoked from host (CPU) code.

```
model:
{
        nStates = 3;    //Total number of states in the model
        nParams = 9;    //Number of parameters describing the model's kinetics
        eRev = -100;    //The reversal potential of the channel
        nOpenStates = 1;        //Number of open (permeable) states in the model

//Name, fit range and the value used in generating surrogate data for each parameter
params = (
                {name = "a12"; min = 0.0; max = 1.0; val = 0.05;},
                {name = "z12"; min = 0.0; max = 1.0; val = 0.05;},
                {name = "a21"; min = 0.0; max = 1.0; val = 0.05;},
                {name = "z21"; min = 0.0; max = 1.0; val = 0.05;},
                {name = "a23"; min = 0.0; max = 1.0; val = 0.05;},
                {name = "z23"; min = 0.0; max = 1.0; val = 0.05;},
                {name = "a32"; min = 0.0; max = 1.0; val = 0.05;},
                {name = "z32"; min = 0.0; max = 1.0; val = 0.05;},
                {name = "gmax"; min = 0.0; max = 10.0; val = 5.0;}
        );
        //Transition state equations describing the model
        rates = [
                "k12 = a12*exp(z12*v)",
                "k21 = a21*exp(-z21*v)",
                "k23 = a23*exp(z23*v)",
                "k32 = a32*exp(-z32*v)"
        ];
        openStates = [3];       //An array describing the indices of the states.
};
```

Each thread has a specific address (threadIdx) which describes its coordinates in the block, and the block has an address (blockIdx) describing its coordinates in the grid. When the kernel is executed, the thread has a local memory, which has the lifetime of the thread (registers), and shared memory to which all the threads in the block have access.

The threads have access to additional memory types, such as global, constant and texture memory. The global memory, to which all threads have access, is much larger than the shared memory, but access to it is slower. Kernels have different signatures, either global or device. Global kernels are executed from code running on the CPU and, when they are executed, the grid is built and a vacant multiprocessor executes the block. Device kernels can be invoked from another device kernel or from a global kernel. In all cases they are executed in the thread of the global kernel originally invoked from the host code (CPU). All threads of the block are executed in parallel on the multiprocessor. When a block is terminated, new blocks are executed on a vacant multiprocessor. The SIMT architecture executes threads in groups of 32 parallel threads, called warps, with the multiprocessor managing and scheduling the execution of the threads. The number of blocks that can be concurrently executed depends on the demand for shared memory and registers by the executed (global) kernel and its device-invoked kernels. The resources of the multiprocessor (shared memory and registers) are spread among the executed warps. More detailed information is available in the CUDA programming guide (NVIDIA, 2011c).

3. Results

To simulate the kinetics of voltage-gated ion channels we previously used the simulation package NEURON implemented with MPI on a Linux cluster (Gurkiewicz and Korngreen, 2007; Gurkiewicz et al., 2011). Programming GPUs using CUDA required writing a different code in C. Thus, a preliminary step in the development of our application was to create a flexible description of voltage-gated channel mechanisms in C. This was achieved by using a simple configuration file. An example of such a file, describing Model B, is given below. The createQmat kernel defining the Q matrix (Colquhoun and Hawkes, 1995) is automatically generated using a Python script from this configuration file.

This format of model description also allows a non-Markovian description of ion channels. In principle, a Markovian chain of transitions can also describe Hodgkin–Huxley models. For example, the Hodgkin–Huxley voltage-gated potassium conductance can be described by the following scheme and script:

$$C \underset{\beta}{\overset{4\alpha}{\rightleftharpoons}} C \underset{2\beta}{\overset{3\alpha}{\rightleftharpoons}} C \underset{3\beta}{\overset{2\alpha}{\rightleftharpoons}} C \underset{4\beta}{\overset{\alpha}{\rightleftharpoons}} O$$

$$\alpha = 0.01(10 - v)/(\exp((10 - v)/10) - 1)$$
$$\beta = 0.125\exp(-v/80)$$

```
model:

{
        nStates = 5;
        nParams = 6;
        eRev = -100;
        nOpenStates = 1;

params = (
                {name = "a1"; min = 0.0; max = 1.0; val = 0.01;},
                {name = "a2"; min = 0.0; max = 40.0; val = 10;},
                {name = "a3"; min = 0.0; max = 40.0; val = 10;},
                {name = "b1"; min = 0.0; max = 1.0; val = 0.125;},
                {name = "b2"; min = 0.0; max = 200.0; val = 80;},
                {name = "gmax"; min = 0.0; max = 10.0; val = 5.0;}
        );

        rates = [
                "k12 = 4*a1*(a2-v)/(exp((a2-v)/a3))",
                "k21 = b1*exp(-v/b2)",
                "k23 = 3*k21/4",
                "k32 = 2*k21",
                "k34 = 2*k21/4",
                "k43 = 3*k21",
                "k45 = k21/4",
                "k54 = 4*k21",

        ];
        openStates = [5];
};
```

Since we used a code different from NEURON to solve the equations, it was necessary to validate our implementation. We performed the simulation of the Markov channel models described in the Methods using both MATLAB and NEURON with the same set of parameters and compared the resulting ionic currents. Both the MATLAB and NEURON implementations provided results almost identical to the CUDA implementation. The mean error in all cases tested was smaller than 0.05%.

3.1. Basic CUDA implementation

Parallelization of a genetic algorithm on a multi-core cluster is straightforward. A population of parameter vectors is created on the master node. Each individual is sent to a different slave node which simulates the response of the model to different voltage-clamp protocols, generates a set of data vectors, subtracts these vectors from the surrogate data vectors and calculates χ^2 (Eq. (2)). This is then returned to the master node (Fig. 2). The master node applies the operators of selection, mutation and crossover to the population. The modified population is repeatedly sent to the slaves until the termination criteria are met.

Naively, this algorithm can be transferred as is to the GPU. It was clear that the CPU should manage the GA and that the threads running on the GPU should solve the ODEs. Thus, our initial algorithm transferred the data from the CPU to the global memory where it was available to all threads. The kernel calculated the probability of each state in the model and the ionic current, and subtracted it from the surrogate or experimental data returning χ^2 to the CPU. To perform these operations, the kernel needed access to the entire current and voltage vectors of the surrogate voltage-clamp data. When executing a device kernel from the host, all memory used by the kernel needed to be copied to the device. Since the population consisted of many individuals, the memory used by the kernel exceeded the size of the shared memory. This required the use of global memory, resulting in slow execution of code. This CUDA implementation was only eight times faster than a single CPU running NEURON, clearly not fulfilling the potential of the GPU platform (John et al., 2009; Liu et al., 2009; Manavski and Valle, 2008; Nageswaran et al., 2009; Pinto et al., 2009; Rossant et al., 2011; Stone et al., 2007; Won-Ki et al., 2010). The only modest increase in speed of code execution was due to the heavy reliance on the GPU's global memory. The faster and smaller shared memory could not hold the large amount of data generated by each individual. Even a simple voltage-clamp experiment, consisting of 10 voltage steps, sampled at 10 kHz and lasting 100 ms, requires each thread to hold 10,000 data points. This is well beyond the capacity of the shared memory.

To overcome the limitations imposed by the size of the data set, we broke down the numerical integration procedure into intrinsic steps. Advancing the numerical solution of any set of differential equations by Δt from y_n to y_{n+1}, does not require the full history of the solution. One requires only y_n, Δt, and the set of ODEs defining the model, namely the Q matrix. We used this basic property to greatly reduce the memory requirements of each thread. In each iteration the thread received a small data set from the global memory (consisting of the Q matrix, a surrogate voltage-clamp current I_{n+1} at time t_{n+1}, a simulated current i_n at time t_n, and Δt). These data were sufficiently small to store in local memory (registers). To simulate the population defined in each GA iteration, at each time step the host code (CPU) invoked a global kernel with a grid the size of the population. The kernel computed the model's state probabilities from the Q matrix, as well as the ionic current and the χ^2. When the kernel finished, it returned the Q matrix i_{n+1} and the χ^2 to the host through the global memory. The CPU then advanced the simulation to the next time step, until the end of the data vectors. This implementation was considerably faster than our initial naïve implementation. On a GTX295 it ran ~5.5 faster than a similar optimization written in NEURON, that ran on an 80 CPU Linux cluster. On a GTX480 it ran ~21 times faster than the cluster implementation.

3.2. Code optimization

This basic CUDA implementation achieved the primary goal of our research, producing a desktop application that eliminated the need to run our optimizations on an expensive Linux cluster. We now turned to optimize the code using NVIDIA's Nsight debugger and profiler. These tools allowed us to follow the execution of the

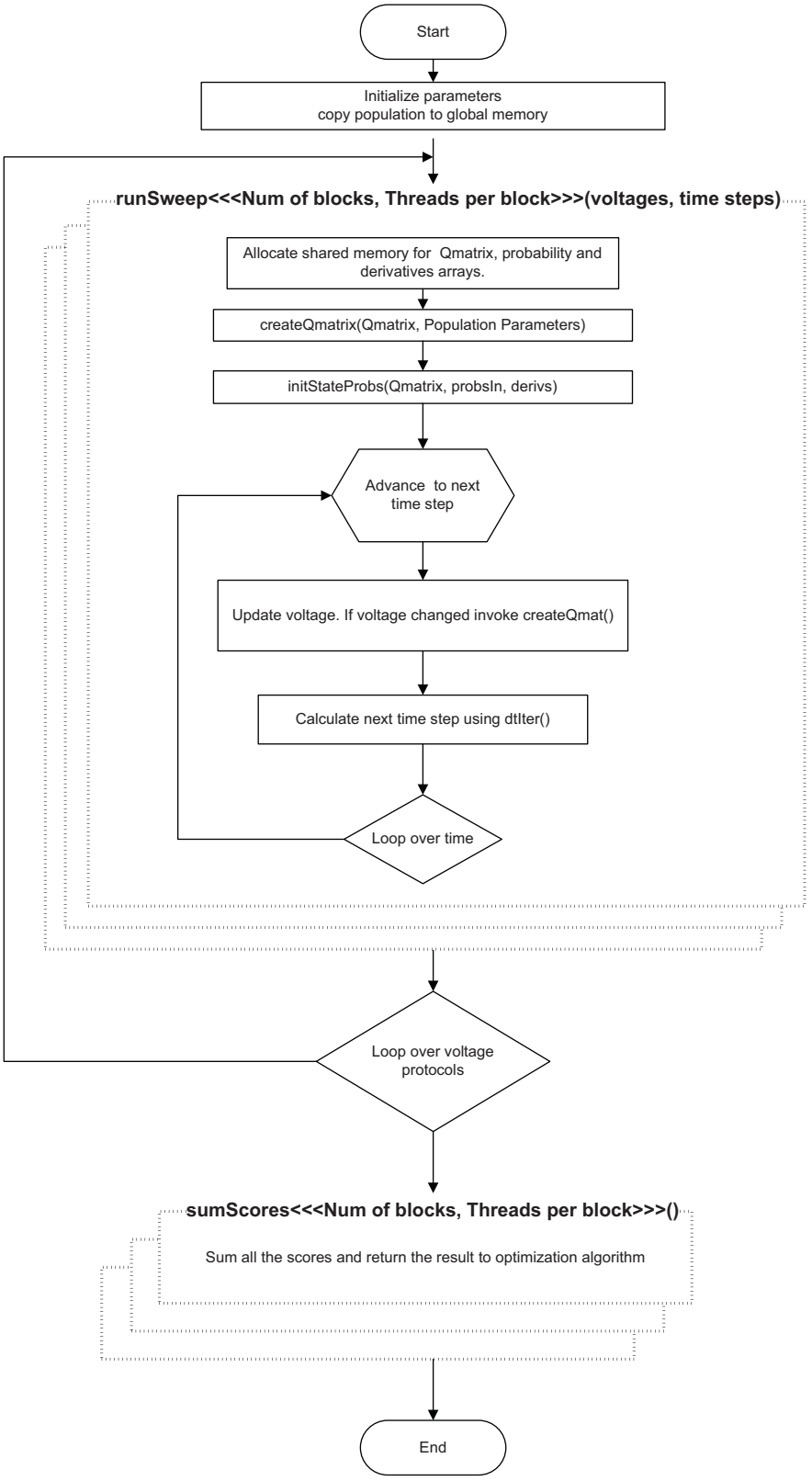


Fig. 3. Flow chart of the runGeneration function.

code and identify possibilities for code enhancement. This analysis revealed a part of the code that was a significant time consumer. Since the kernel solving the unitary ODE step was called from the CPU as a global kernel, the Q matrix, i_n, t_n and Δt were copied to the shared memory. For every kernel call the resulting i_{n+1} and model state probabilities were copied from the shared memory back to the global memory. Since the kernel was called from the CPU, the data could be directed to a different SM each time. It was therefore not possible to retain copies of the data and Q matrix in the shared memory between successive time steps. This resulted in

many time-consuming memory transfers between the global and the shared memory. To reduce the number of such memory transfers we stored variables undergoing modification by the algorithm in the shared memory whereas data vectors not modified during the calculation were stored in the constant memory of the GPU (NVIDIA, 2011a). The runSweep kernel performed most of this operation. The flow of the algorithm performed by the GPU is given in Fig. 3 and the annotated code of runSweep is presented below.

The runSweep kernel (Fig. 3) is invoked by the host runGeneration function. For every individual in the population the runSweep kernel propagates the differential equations, and for every time step in the sweep it calculates the score and ionic current. This global kernel is executed by a grid of threads. The dimensions of this grid are defined by the number of threads in a block (block size), the number of blocks in the grid (grid size), and optionally by the size

```
__global__ void runSweep(float *paramsGlobalDev, const float *IsIn, const float *voltages,
        const int *fits,  const float *dts, float totalT, float eRev, int nParams,
        int nStates, int nOpenStates, int *openStates, int popSize,
        float *chiScoreGlobal, float *outIdev, int nTrials,int nTimeSteps)
{
/* Local variables that are stored in registers */
        float currv, currInI, currdt=0, currFit, currOutI, chiSum;  // temporary  variables for voltage and current
        float *tempPtr;
        int timeStep=0;         //temporary variable for time step
        int i=0, addr = 0;      // addr holds the offset of the address in the shared memory.

/* Main mapping of shared memory to each thread. The smem variable denotes a "chunk" of shared memory that contains all
the shared memory for one block. Every thread in the block has its unique space in the shared memory for every variable. In
order to disperse smem to variables for each thread we use the threadIdx.x. For example there are T/B number of qmatrices
in smem and they appear in the start of smem, each thread uses the size (NSTATES^2) and the thread address to point the
relevant qmatrix. Then we advance addr in smem to where all the qmatrices end. This mapping facilitates rapid access of the
information from memory to the threads*/

        float* qMatrix = &smem[NSTATES*NSTATES*threadIdx.x];
        addr += NSTATES*NSTATES*blockDim.x;
        float* yIn = &smem[addr+NSTATES*threadIdx.x];
        addr += NSTATES*blockDim.x;
        float* yOut = &smem[addr+NSTATES*threadIdx.x];
        addr += NSTATES*blockDim.x;
        float* dy = &smem[b+NSTATES*threadIdx.x];

/* paramsGlobalDev and chiScoreGlobal are arguments that holds memory for all the threads in the block. Each thread needs
to use its address to point the right place in the arguments. */

        float gMax = *(&paramsGlobalDev[(((blockDim.x*blockIdx.x)+threadIdx.x)*nParams)+(nParams-1)]);
        float *chiScore = &chiScoreGlobal[((blockDim.x*blockIdx.x)+threadIdx.x)];

/*Initiating the calculation in each thread */

        currv = voltages[0];
        createQmat(currv,paramsGlobalDev,nStates,nParams,qMatrix);              // creating the Q matrix
        initStateProbs(yIn,qMatrix,yOut,nStates); //Calculating the initial probabilities of the Markov chain.

/* Main loop advancing time */

        for(i=0;i<nTimeSteps;i+=1)
        {
                currv = voltages[timeStep];

                // Recreating the Q matrix only if there was a change in the voltage
                if( timeStep!=0 && currv != voltages[timeStep-1] )
                        createQmat(currv,paramsGlobalDev,nStates,nParams,qMatrix);

        // Moving current time step information to local variables and advancing the simulation by one time step
                currdt = dts[timeStep];
                currIin = IsIn[timeStep];
                currFit = fits[timeStep];
                 dtIter(yIn, qMatrix, gMax, currIin, currv, eRev, currdt, nStates, nOpenStates, openStates,
                &chiSum, yOut, dy, &currOutI, currFit);
                timeStep +=1;

                tempPtr = yIn;          //Swapping between yIn and yOut
                yIn = yOut;
                yOut = tempPtr;
        }
        *chiScore += chiSum/(nTimeSteps*nTrials);  //updating the global χ².
}
```

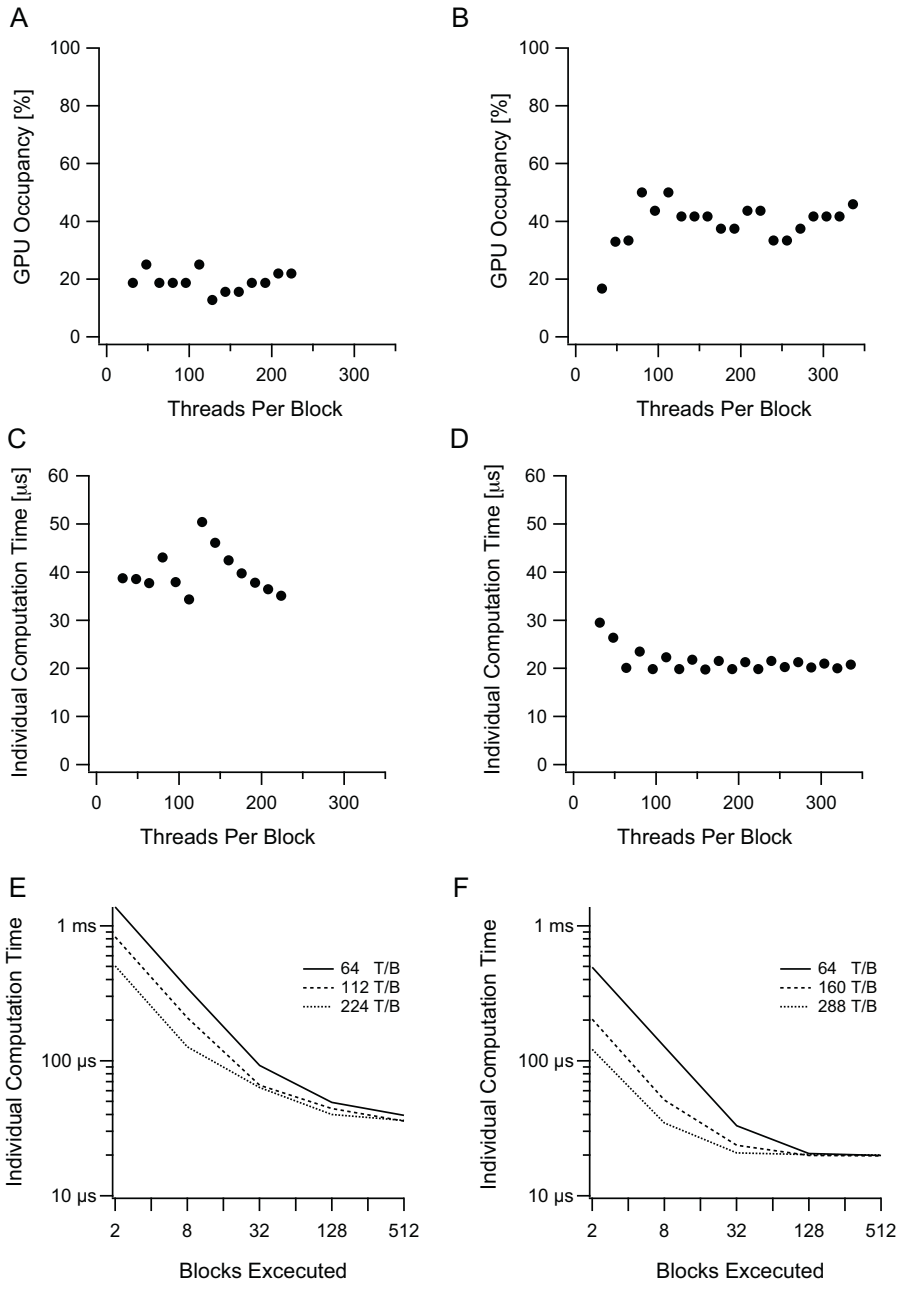


Fig. 4. Optimizing the number of threads per block (T/B) for Model B. (A) Occupancy vs. threads per block for GTX295. Maximum number of T/B was 224 due to the amount of shared memory allowed per block (16 kBytes). In all simulations we used 36 registers per thread (R/T). Since the SMs in GTX295 were calculated per half-warp, the difference between adjacent points in Fig. 4 is 16 T/B. (B) Occupancy vs. threads per block for GTX480. Maximum number of T/B was 352 due to the amount of shared memory allowed per block (48 kBytes). (C) Average time required to calculate one individual on the GTX295 card with varying T/B values. Grid size was 2048 blocks. (D) Average time required to calculate one individual on the GTX480. Grid size was 2048 blocks. (E) Average time required to calculate one individual on the GTX295, when the size of the grid was changed. (F) Average time required to calculate one individual on the GTX480 card when the size of the grid was changed.

of shared memory used by every block. The block size or number of threads per block (T/B) is a predefined parameter set by the user (see below and Fig. 4).

Grid size was calculated by dividing the size of the population by the block size. Every individual was calculated by a thread that was part of the block grid and had a unique address in that grid. The thread used this address to access the relevant shared memory location to obtain the parameter values of the specific individual. The surrogate data vectors were long vectors and accessed by all threads, in contrast to the Q matrix and probability vectors that changed from thread to thread. Each surrogate data value was accessed only once by each thread. Therefore, the const identifier was used for all surrogate data vectors instructing the compiler to place these vectors in constant, cached, memory allowing fast access to this data (NVIDIA, 2011c). This allowed fast access to data accessed by all threads without exhausting the limited size of the shared memory.

After allocating shared memory, the Q matrix was initiated using the device kernel createQmat which was invoked from the runSweep kernel. Device kernels ran in the block of threads created when a global kernel was invoked, thus with access to the same shared memory. Since createQmat was a device kernel invoked from runSweep, the Q matrix was available to the runSweep kernel after termination of the createQmat kernel. This relationship

between the device and global kernels allowed the same information to be used in the next stages of the computation when the initial state probabilities of the Markovian chain were calculated by the initStateProbs. Consequently the model was integrated for every time step using the dTIter kernel (Fig. 3). After the numerical integration of the entire sweep was terminated, the runSweep kernel added the score from the current sweep to previous sweeps and terminated. A new grid of threads was prepared by the runGeneration function calling runSweep with the next sweep in the surrogate data (Fig. 3).

3.3. Tuning block size

The speed at which an application is executed on the GPU is greatly influenced by the number of threads per block (T/B), since this determines how the streaming multiprocessors (SMs) are utilized (NVIDIA, 2011a). Each thread utilizes a certain amount of shared memory and registers, as set by the use of the global kernel (NVIDIA, 2011c). Since threads are executed in blocks on the SM, the T/B parameter sets the number of registers and shared memory used by the block. Different GPU cards place specific hardware limitations on the registers and shared memory available for a block. The limits of the GTX295 are 16 kByte for both registers and shared memory and those for the GTX480 are 32 kByte and 48 kByte respectively. If the block size exceeds the hardware limits of the shared memory, the application fails to run on the GPU. Where block size exceeds register limitations, register spilling occurs, which considerably slows the application (NVIDIA, 2011a). On the other hand, if the block does not demand all the hardware resources the same SM can run more blocks. The maximal number of blocks that can be computed by a single SM is 8 and the multiprocessor switches between the blocks. This increases occupancy and effectiveness of the code. As an SM executes in warps of 32 threads, T/B should be multiples of 32 (GTX295 runs in half-warps so T/B should be a multiple of 16).

To determine the optimal T/B we systematically changed this parameter and measured the speed of the algorithm and the occupancy of the GPU. When numerically integrating Model B runSweep stored the Q matrix, the input vector of state probabilities, the output vector of the integration and the derivatives of the model in the shared memory. The Q matrix is a square matrix, of size $NSTATES^2$, and the others are vectors of size NSTATES. Model B has 3 states occupying 72 bytes of shared memory, since we defined them as floats. Given a T/B of 32 and a shared memory of 16 kbyte on the GTX295, seven blocks were executed by one SM. On the GTX295 the occupancy of the GPU during computation was ~20% for all T/B values (Fig. 4A). Better occupancy values were obtained on the GTX480, leveling out at ~40% for most T/B values (Fig. 4B). We observed the speed of computation as a function of T/B. On the GTX295 the time required to compute one individual in the population displayed discontinuities when the computation switched from three to two blocks per SM and from two to one blocks per SM (Fig. 4C). The fastest code execution was observed at 112 T/B. Some discontinuities were also observed when we systematically changed T/B in the GTX480 (Fig. 4D). These discontinuities are mostly due to the faster execution of the code when T/B values were multiples of 32 (full warp) as opposed to multiples of 16 (half warp). For Model B a value of 160 T/B with 48 registers per thread gave the fastest code execution.

We next investigated the effect of grid size (total number of blocks) on the time required to calculate one individual. We examined configurations with 3, 2 and 1 blocks (64, 112 and 224 T/B respectively) executed simultaneously by an SM on the GTX295 (Fig. 4E). For a small number of blocks the computation was faster when an SM executed more blocks. However, this advantage decreased as the number of blocks increased. A similar trend was observed on the GTX480 where we investigated the differences between configurations using 2, 4, and 8 blocks per SM (Fig. 4F).

Having optimized the T/B values for both GTX295 and GTX480, we now quantitatively compared the application described here to our previous genetic algorithm written in NEURON (Gurkiewicz and Korngreen, 2007). All simulations using NEURON were run on an 80 CPU Linux cluster. We generated identical surrogate data sets for the NEURON and GPU applications. When the populations were small, the time required for calculating one individual decreased as the number of individuals in the population increased (Fig. 5A). As the populations increased in size, the time required for calculating one individual reached a constant value. This appeared to be qualitatively similar for both NEURON and GPU implementations. In both cases the longer time required to calculate one individual in smaller populations was due to the overhead imposed by initial memory transfer, either from the master node of the Linux cluster to the slaves or from the CPU to the GPU. For large populations the GTX295 and GTX480 were ~75 and ~180 times faster than the NEURON implementation on an 80-node Linux cluster (Fig. 5A).

As expected, increasing the number of model states increased the time needed to calculate one individual in the population for both NEURON and GPU applications (Fig. 5B). This was obviously due to the number of both shared memory bytes used and floating point operations performed for each individual. This increase was not monotonous for the GPU application, as the GTX480 has 32 shared memory banks and the GTX295 16 (NVIDIA, 2009, 2011c). Since each shared memory bank handles every *n*th 32-bit word in the memory, it is important to have a memory access pattern coprime with the number of shared memory banks. This guarantees that each thread in the warp or half-warp accesses bytes from different banks. Thus, when working with a Q matrix of size $NSTATES^2$, a shared memory stride of this size is produced when accessing corresponding elements from each matrix. To prevent the ensuing shared memory bank conflicts, we padded the Q matrix (NVIDIA, 2011a). Before padding, models with four and six states performed poorly compared to models with five and seven states, respectively. After padding, the computation time was significantly reduced for the models with an even number of states. The overhead of the extra, unused, 32-byte words was still clearly evident in the sharp increase from three states to four. It is important to note that each added state in our kinetics schemes added four free parameters to the optimization (see Section 2). The Hodgkin–Huxley model of the potassium channel described above has 4 states with only five free parameters due to the interstate dependence. Clearly, the GA will optimize the 5-parameter problem faster than the 16-parameter problem that is presented by a 4 state Markov chain model similar to those defined in the methods. Thus, to fully compare the performance of the algorithm for other kinetic schemes it is imperative to compare the number of free parameters in the model and the number of states. Finally, we investigated the effect of surrogate data size on the speed of the computation (Fig. 5C). Increasing the number of data points in the surrogate data generated similar qualitative increases in the time required for calculating one individual for both NEURON and GPU applications (Fig. 5C).

The considerable increase in speed (Fig. 5A) and reduction in calculation time for each individual in a larger population (Fig. 4E and F) allowed us to increase the size of the generation without losing much computation time. We have previously observed that the GA converged better with low crossover probabilities in our NEURON application (Gurkiewicz and Korngreen, 2007), a counter-intuitive observation since crossover encourages transfer of genetic material within the population allowing the GA to converge faster. We therefore ran the same GA optimization of Model B using two crossover probabilities on the GPU application. When the population was of similar size to those previously used in the NEURON

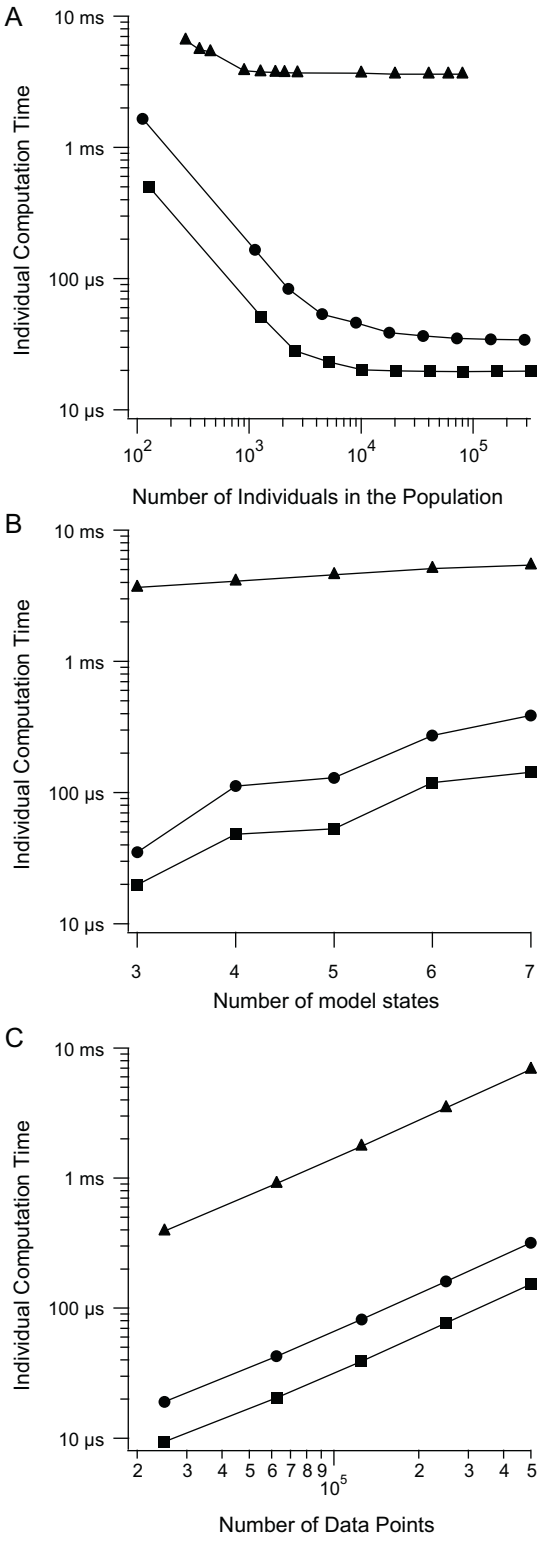


Fig. 5. Comparison of the calculation speed using a NEURON-MPI or GPU. (A) Average speed of calculation for one individual with different population sizes using Model B on different platforms: 80 CPU cluster (triangles), GTX295 (circles) and GTX480 (squares). (B) Comparison of average speed of calculation of one individual with different models on the 3 platforms (as in A). (C) Comparison of average speed of calculation of one individual with different sized of data sets by the 3 platforms (as in A).

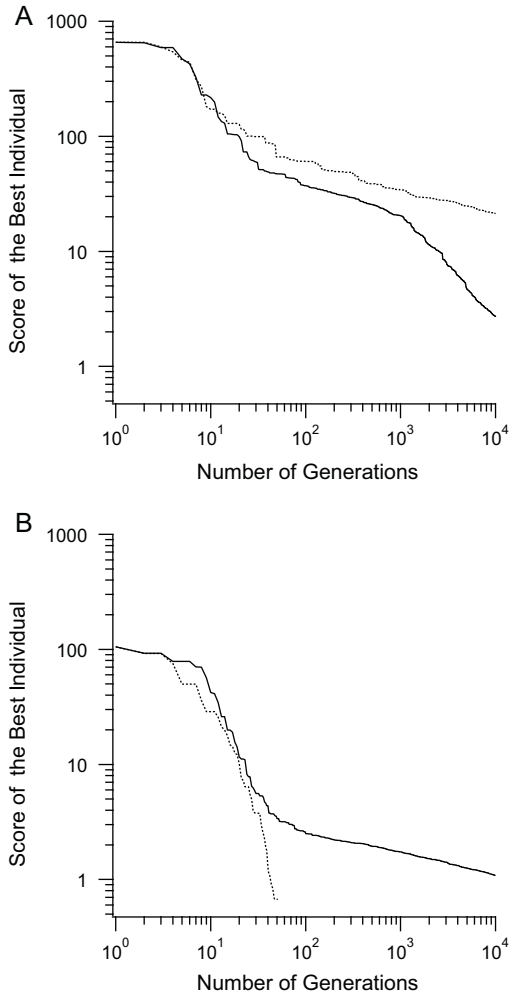


Fig. 6. Advantages of larger populations for GA convergence. (A) Optimization of Model B with population size of 112 and crossover rates of 0.8 (dots) and 0.1 (solid line). (B) Optimization of Model B with population size of 11,200 with crossover probabilities as in A.

application, we were able to reproduce the observation that low crossover probabilities facilitated better GA convergence (Fig. 6A). However, increasing the size of the population one hundred-fold (from 112 to 11,200 individuals) gave much better GA convergence at high crossover probabilities (Fig. 6B).

These experiments prompted us to investigate the relationship between population size and the speed of GA convergence. As the size of the population increased from 112 to 1,120,00 individuals, the convergence of the GA to a χ^2 score of 0.6 required fewer generations (Fig. 7A). For the three large populations (11,200, 112,000 and 1,120,000 individuals) only ~50 generations were required to reach the 0.6 χ^2 score (Fig. 7A). Plotting the same GA convergences as a function of the elapsed time revealed that there was optimal convergence when the population was big enough to allow proper application of the crossover and mutation operators to the population. Above this optimal population size the convergence lasted longer but showed no improvement (Fig. 7B).

4. Discussion

We have previously applied genetic optimization algorithms to constrain parameters of Markov ion channel models using

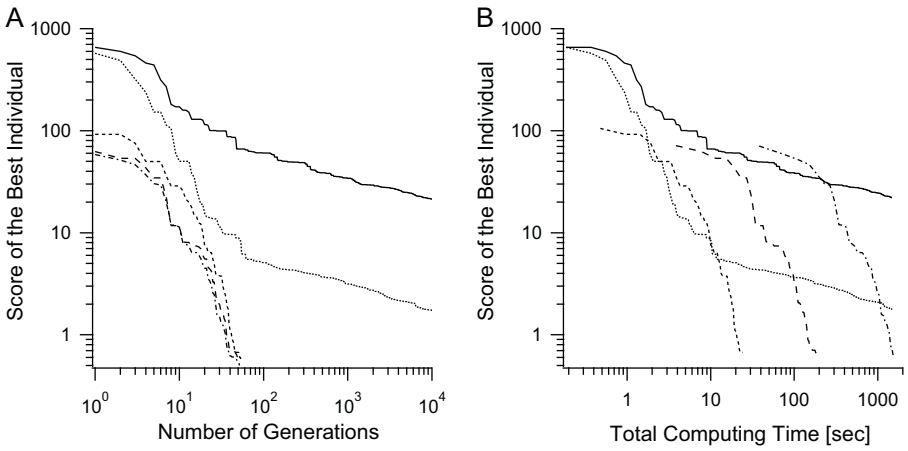


Fig. 7. GA computation times for varying population sizes. All simulations were executed on protocols with 12,500 data points and 112 T/B on a GTX295 card. Simulations terminated when the score function reached a sufficient score (0.6) or exceeded 10,000 generations. (A) Score of the best individual in the population plotted as a function of the number of generations executed using different population sizes. Population size was 112 individuals (solid line), 1120 individuals (dots), 11,200 individuals (dashed line), 112,000 individuals (lines separated with one dot) and 1,120,000 individuals (lines separated with two dots). (B) Score of the best individual in the population plotted as a function of the computing time obtained using different populations sizes (population sizes as in A).

high performance Linux clusters (Gurkiewicz and Korngreen, 2007; Gurkiewicz et al., 2011). We encountered computational exhaustion due to the complexity of the models and the amount of electrophysiological data. Here we show that, using a commercially available GPU, it is possible to speed up the execution of the algorithm on a standard desktop computer. This substantial increase in speed was achieved by using the shared and constant memories of the GPU for rapidly accessed variables and reducing the memory copied to and from the GPU. This approach can be applied to speed up other data-intensive applications requiring iterative solutions of differential equations.

Efficient utilization of the GPU as a computational engine requires adapting any algorithm to the logic imposed by GPU hardware. Two primary points should be considered. First, efficient code execution on the GPU occurs when the algorithm adheres to the SIMT logic. Second, faster code execution can be obtained by using the smaller and faster shared memory rather than the global memory. Moreover, minimizing the shared memory segment used by each thread allows defining more threads per block. Naturally, with more threads running in parallel the code runs faster.

The genetic algorithm is highly suitable for implementation on a GPU since it attempts to optimize a single function using many possible solution vectors (Mitchell, 1996). When the amount of data used by the function to be optimized is small, it can be simply loaded to the shared memory of the GPU giving a fast code execution. In our application, calculating the score of each individual in the population required simulating long traces of voltage-clamp currents (Fig. 1). These could not be copied to the shared memory. But using only the global memory to store voltage-clamp currents and the solution vector population resulted in poor performance. To minimize shared memory used by each thread, we changed the flow of the algorithm (Fig. 3), with the parallelization occurring at the most basic procedure – calculating the current and the fit of one time step in one trace in an individual. In addition, using device kernels called from within the runSweep global kernel (Fig. 3) allowed efficient handling of the shared memory. Thus, it was not necessary to recopy the Q matrix for every call of the dtIter kernel and to copy the model state probabilities. This approach could not be used for storing the long voltage-clamp vectors. Using the shared memory for these vectors would have resulted in overloading the shared memory. Using the global memory to store these vectors would have increased device latency – the SM would have wait for accessing the slow global memory. To address this problem we stored the voltage-clamp protocols and data in the cached constant memory of the GPU (Fig. 3).

The occupancy of the SMs may be regarded as a good measurement for effectiveness of a GPU application. We controlled the occupancy of the SMs by tuning the values for threads per block (T/B) and register per thread (R/T). On the GTX295 the occupancy fluctuated around 20% for all values of T/B tested (Fig. 4A) and on the GTX480 the occupancy peaked at 50% (Fig. 4C). Moreover, on the GTX480 the fastest code execution was observed at occupancy values lower than 40% (Fig. 4D). These occupancy values indicate that the code can be further optimized. However, we achieved the fastest execution of the code when we did not optimize the number of blocks running simultaneously on one SM at the highest value of GPU occupancy (Fig. 4C and D). Thus, optimal performance appears to also depend on factors other than high GPU occupancy.

The GA, implemented in CUDA and executed on a GTX480, ran ~180 times faster than the NEURON-MPI implementation run on an 80 CPU cluster (Fig. 5A). This substantially faster execution of the code allowed us to use larger populations in the GA, which, in turn, reduced the time required for calculating one individual in the population (Fig. 5A). The longer execution times of one individual in small populations were probably due to the time required for initially copying the memory. The GA here converged optimally with high crossover probabilities and large populations (Figs. 6 and 7). Using larger populations with the GA is clearly advantageous. The increased variability in a larger population allows the GA to explore parameter space more effectively. Once a good region in parameter space is found, it can be better exploited by many individuals. That is, larger populations were better at both exploring and exploiting phases of the optimization algorithm. Concomitantly, increasing the population size resulted in a much better fit (Fig. 7).

We compared the speed of the GA on the GPU to that written in NEURON and run on an 80 node Linux cluster. NEURON is a multipurpose program designed to simulate compartmental models of complex neurons and ion channels (Hines and Carnevale, 2000, 2001). Its flexibility adds considerable computational overhead to the computation; this computation overhead is absent in our GPU code. Conversely, our code lacks most of the abilities built into NEURON. Due to its generic implementation, code development in NEURON is rather quick, flexible, and does not contain many options for optimization. Coding the GA in NEURON was straightforward and using it to implement complex neural models on the MPI ring did not consume many man-hours (Keren et al., 2005, 2009). After

considerable optimization our GPU code ran much faster than the NEURON-MPI code. However, the code development process was much longer and more complex on the GPU. Our current experience shows that a CUDA code must be tailored to each algorithm and that the user is well advised to manipulate the T/B parameter to improve performance. Moreover, only algorithms that comply with the SIMT logic can be efficiently transferred to the GPU.

The comparison of our current application to the NEURON-MPI application is thus far from perfect. Theoretically, it is best to compare the code to an identical code running on the CPU. This is not possible due to the large amount of CUDA directives in the code. Nor is it possible at present to run NEURON on the GPU, but a recent study has used CPU threads to split the computation in NEURON (Eichner et al., 2009). Our study has demonstrated that the execution of NEURON can be sped up by roughly the number of CPU cores. Thus, although the analogy is incomplete, running our GPU code on a multi-core CPU will not achieve the speed we obtained on the GPU. NEURON computes using the slower double precision compared to the single precision we used in our CUDA application. This probably partly accounts for the higher speedup of the GPU code.

Finally, we point out not only the net increase in computation speed but also the reduction in the effective cost of the computation. An 80 node Linux cluster costs ~$20,000 to purchase and must be housed in an air-conditioned server room. The running electricity costs are, therefore, substantial. In contrast, a high-end desktop computer housing two GTX480 units should not cost more than ~$2000 and it consumes much less electricity than a Linux cluster. Thus, the cost/computation ratio highly favors the GPU as a computing engine.

Acknowledgements

This work was supported by a professor partnership grant from NVIDIA and by a grant from the German Israeli Foundation (#1091-27.1/2010). Benjamin Razon was supported by a SURF summer fellowship from the California Institute of Technology.

References

Colquhoun D, Hawkes A. A Q-matrix cookbook. In: Sakmann B, Neher E, editors. Single-channel recording. New York: Plenum Press; 1995. p. 589–633.
Colquhoun D, Sigworth F. Fitting and statistical analysis of single-channel records. In: Sakman B, Neher E, editors. Single-channel recording. New York: Plenum; 1995. p. 483–585.
Eichner H, Klug T, Borst A. Neural simulations on multi-core architectures. Front Neuroinform 2009;3:21.
Gropp WaT, Rajeev, Lusk E. Using MPI-2: advanced features of the message passing interface. 2nd ed. MIT Press; 1999.
Gurkiewicz M, Korngreen A. A numerical approach to ion channel modelling using whole-cell voltage-clamp recordings and a genetic algorithm. PLoS Comput Biol 2007;3:e169.
Gurkiewicz M, Korngreen A. Recording analysis, and function of dendritic voltage-gated channels. Pflugers Arch 2006;453:283–92.
Gurkiewicz M, Korngreen A, Waxman SG, Lampert A. Kinetic modeling of Nav1.7 provides insight into erythromelalgia-associated F1449V mutation. J Neurophysiol 2011;105:1546–57.
Hamill OP, Marty A, Neher E, Sakmann B, Sigworth FJ. Improved patch-clamp techniques for high-resolution current recording from cells and cell-free membrane patches. Pflugers Arch 1981;391:85–100.
Hines ML, Carnevale NT. Expanding NEURON's repertoire of mechanisms with NMODL. Neural Comput 2000;12:995–1007.
Hines ML, Carnevale NT. The NEURON simulation environment. Neural Comput 1997;9:1179–209.
Hines ML, Carnevale NT. NEURON: a tool for neuroscientists. Neuroscientist 2001;7:123–35.
Hodgkin AL, Huxley AF. The components of membrane conductance in the giant axon of Loligo. J Physiol 1952a;116:473–96.
Hodgkin AL, Huxley AF. Currents carried by sodium and potassium ions through the membrane of the giant axon of Loligo. J Physiol 1952b;116:449–72.
Hodgkin AL, Huxley AF. The dual effect of membrane potential on sodium conductance in the giant axon of Loligo. J Physiol 1952c;116:497–506.
Hodgkin AL, Huxley AF. A quantitative description of membrane current and its application to conduction and excitation in nerve. J Physiol 1952d;117:500–44.
Hodgkin AL, Huxley AF, Katz B. Measurement of current–voltage relations in the membrane of the giant axon of Loligo. J Physiol 1952;116:424–48.
John ES, Jan S, David JH, Kirby LV, Wen-mei WH, Klaus S. High performance computation and interactive display of molecular orbitals on GPUs and multi-core CPUs. In: Proceedings of 2nd workshop on general purpose processing on graphics processing units. Washington, D.C.: ACM; 2009.
Keren N, Bar-Yehuda D, Korngreen A. Experimentally guided modelling of dendritic excitability in rat neocortical pyramidal neurones. J Physiol 2009;587:1413–37.
Keren N, Peled N, Korngreen A. Constraining compartmental models using multiple voltage recordings and genetic algorithms. J Neurophysiol 2005;94:3730–42.
Liu Y, Maskell DL, Schmidt B. CUDASW++: optimizing Smith–Waterman sequence database searches for CUDA-enabled graphics processing units. BMC Res Notes 2009;2:73.
Manavski SA, Valle G. CUDA compatible GPU cards as efficient hardware accelerators for Smith–Waterman sequence alignment. BMC Bioinformatics 2008;9(Suppl. 2):S10.
Mitchell M. An introduction to genetic algorithms. Cambridge, MA, US: The MIT Press; 1996.
Nageswaran JM, Dutt N, Krichmar JL, Nicolau A, Veidenbaum AV. A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors. Neural Networks 2009;22:791–800.
NVIDIA. CUDA C best practices guide; 2011a.
NVIDIA. Nsight; 2011b.
NVIDIA. NVIDIA CUDA compute unified device architecture programming guide; 2011c.
NVIDIA. NVIDIA Fermi compute architecture whitepaper; 2009.
NVIDIA. NVIDIA Tesla GPU computing technical brief; 2007.
Pinto N, Doukhan D, DiCarlo JJ, Cox DD. A high-throughput screening approach to discovering good forms of biologically inspired visual representation. PLoS Comput Biol 2009;5:e1000579.
Qin F, Auerbach A, Sachs F. Estimating single-channel kinetic parameters from idealized patch-clamp data containing missed events. Biophys J 1996;70:264–80.
Qin F, Auerbach A, Sachs F. Maximum likelihood estimation of aggregated Markov processes. Proc Biol Sci/R Soc 1997;264:375–83.
Rossant C, Goodman DF, Fontaine B, Platkiewicz J, Magnusson AK, Brette R. Fitting neuron models to spike trains. Front Neurosci 2011;5:9.
Sakmann B, Neher E. Single channel recording. New York: Plenum; 1995, p. 700.
Stone JE, Phillips JC, Freddolino PL, Hardy DJ, Trabuco LG, Schulten K. Accelerating molecular modeling applications with graphics processors. J Comput Chem 2007;28:2618–40.
Tobias B, Lothar T. A comparison of selection schemes used in evolutionary algorithms. Evol Comput 1996;4:361–94.
Willms AR, Baro DJ, Harris-Warrick RM, Guckenheimer J. An improved parameter estimation method for Hodgkin–Huxley models. J Comput Neurosci 1999;6:145–68.
Won-Ki J, Beyer J, Hadwiger M, Blue R, Law C, Vazquez-Reina A, et al. Analysis tools for large-scale neuroscience data sets. IEEE Comput Graph Appl 2010;30:58–70.