SODECL: An Open-Source Library for Calculating Multiple Orbits of a System of Stochastic Differential Equations in Parallel

ELEFTHERIOS AVRAMIDIS, University of Exeter and University of Cambridge, UK
MARTA LALIK, Isomerase Therapeutics Ltd., UK
OZGUR E. AKMAN, University of Exeter, UK

Stochastic differential equations (SDEs) are widely used to model systems affected by random processes. In general, the analysis of an SDE model requires numerical solutions to be generated many times over multiple parameter combinations. However, this process often requires considerable computational resources to be practicable. Due to the embarrassingly parallel nature of the task, devices such as multi-core processors and graphics processing units (GPUs) can be employed for acceleration.

Here, we present SODECL (https://github.com/avramidis/sodecl), a software library that utilizes such devices to calculate multiple orbits of an SDE model. To evaluate the acceleration provided by SODECL, we compared the time required to calculate multiple orbits of an exemplar stochastic model when one CPU core is used, to the time required when using all CPU cores or a GPU. In addition, to assess scalability, we investigated how model size affected execution time on different parallel compute devices.

Our results show that when using all 32 CPU cores of a high-end high-performance computing node, the task is accelerated by a factor of up to ≈6.7, compared to when using a single CPU core. Executing the task on a high-end GPU yielded accelerations of up to ≈4.5, compared to a single CPU core.

CCS Concepts: • **Theory of computation → Massively parallel algorithms**; • **Hardware → Testing with distributed and parallel systems**; • **Software and its engineering → Software libraries and repositories**;

Additional Key Words and Phrases: Stochastic differential equations, CPU, GPU, HPC, OpenCL, Kuramoto model, computational biology, optimisation

24

Authors' addresses: E. Avramidis, University of Exeter, North Park Road, Exeter, EX4 4QF, UK, University of Cambridge, Roger Needham Building, 7 JJ Thomson Avenue, Cambridge, CB3 0RB, UK; email: ea461@cam.ac.uk; M. Lalik, Isomerase Therapeutics Ltd. Science Village, Chesterford Research Park, Cambridge, CB10 1XL, UK; email: marta_lalik@ hotmail.com; O. E. Akman (corresponding author), University of Exeter, North Park Road, Exeter, EX4 4QF, UK; email: O.E.Akman@exeter.ac.uk.

1 INTRODUCTION

1.1 Modelling Dynamic Systems Using SDEs

Noise affects the behaviour of a vast number of physical and biological phenomena, such as gene expression [Kaern et al. 2005; Keren et al. 2015], the transmission of nerve impulses [Deco et al. 2009; Faisal et al. 2008], and the dynamics of electronic circuits [Darabi and Abidi 2000; Kaern et al. 2005; Scholten et al. 2003]. Systems that are affected by noise are commonly modelled using sets of stochastic differential equations (SDEs) [Allen et al. 2008]. In such models, each equation represents the rate of change of a system variable (e.g., voltage) with time and can be dependent on one or more parameters (e.g., capacitance). Moreover, one or more equations are also affected by a noise process. The Itô form of a first-order coupled SDE system is

$$dX_i(t) = f_i(t, \mathbf{X}(t), \mathbf{p}) \, dt + \sum_{j=1}^{m} g_{ij}(t, \mathbf{X}(t), \mathbf{p}) \, dW_j(s), \tag{1}$$

where $t \geq t_0$ is time; $\mathbf{X} = (X_1, \ldots, X_n)$ is the state vector; $\mathbf{X}_0 = \mathbf{X}(t_0)$ specifies the initial state of the system; $\mathbf{p} = (p_1, \ldots, p_d)$ is a vector of parameter values; $\{W_1(t), \ldots, W_m(t)\}$ are independent, scalar Wiener processes; and the functions $\{f_i(t, \mathbf{X}, \mathbf{p}) : 1 \leq i \leq n\}$ and $\{g_{ij}(t, \mathbf{X}, \mathbf{p}) : 1 \leq i \leq n, 1 \leq j \leq m\}$ are the drift and diffusion coefficients, respectively [Higham 2001; Kloeden and Platen 2011; Platen 1999]. Setting the diffusion coefficients g_{ij} to 0 reduces the system to a set of deterministic, ordinary differential equations (ODEs).

1.2 Optimising the Parameters of SDE Models

By varying the parameters $\mathbf{p}$ of the SDE system (Equation (1)), different classes of dynamical behaviour can be observed (e.g., transitions between bursting and spiking in neural models). However, in many cases, the parameters are difficult to measure experimentally. Consequently, optimisation methods are often employed to find the particular parameter combinations that most closely reproduce the experimentally-measured behaviour of the system of interest [Paláncz et al. 2016]. Performing this parameter optimisation step in a robust, systematic fashion is a critical step in the construction and analysis of biological models [Akman et al. 2008, 2010, 2012; Avramidis and Akman 2017; Doherty et al. 2017; Lillacci and Khammash 2010] because determining the optimal parameter values enables alternative models to be ranked and experimentally testable predictions to be formulated [Ashyraliyev et al. 2009; Avramidis and Akman 2017; Cedersund and Roll 2009; Cullen et al. 1996; Johnson and Omland 2004; Lillacci and Khammash 2010; Slezak et al. 2010; Sun et al. 2012]. The assumptions made in constructing a given model can then be rigorously assessed, and insights obtained into how the model could be modified so as to improve the accuracy of its predictions [Ashyraliyev et al. 2009; Avramidis and Akman 2017; Cedersund and Roll 2009; Lillacci and Khammash 2010; Slezak et al. 2010].

However, optimising the parameters $\mathbf{p}$ of the system shown in Equation (1) can incur a high computational cost, depending on the forms of f_i and g_{ij}, the system size n, and the number of noise terms m. Large systems (high n values), multiple noise processes (high m values), and computationally demanding drift and diffusion terms (e.g., terms containing transcendental functions) can all increase the computational cost considerably. Furthermore, an additional cost is

associated with the generation of the random numbers required to simulate the noise pro-
cesses. The computational cost can also be significantly increased by the particular parameter
optimisation method employed; for example, when using a population-based method—such as
an evolutionary algorithm (EA) or a particle swarm optimiser (PSO)—the equations have to
be integrated multiple times over different parameter values to explore the underlying fitness
landscape [Cedersund et al. 2016; He and Yao 2001; Witt 2008]. Additionally, a resampling
approach might be required to mitigate the effects of noise and uncertainty in model evaluation,
thereby increasing the computational load further [Doherty et al. 2017; Fieldsend 2015].

In general, the evaluation of an SDE model for the optimisation methods mentioned above in-
volves individual tasks with no interdependence or communication. These tasks can therefore be
parallelised in a straightforward manner; hence, the optimisation problem is referred to as embar-
rassingly or pleasingly parallel [Navarro et al. 2014].

1.3 Accelerating the Optimisation Process Using Multi-core and Many-core Devices

The significant computational demands of such embarrassingly parallel tasks means that high-
performance computing (HPC) clusters (i.e., multiple connected computers) are required to obtain
results within a reasonable time frame. Recently, competitively-priced CPU models that provide
between 6 and 16 physical cores have become commercially available. In the case of workstation
CPUs, there are now models with 32 cores.

Moreover, current GPU models, which are commonly used for graphics generation in personal
computers and workstations, can also be leveraged for scientific computing [Mantas and Castro
2016; Papadrakakis et al. 2011]. They contain a large number of processing units—each of which
is relatively slow compared to the cores of a CPU—and are optimised for single instruction,
multiple data (SIMD) processing. This makes them well-suited to large-scale parallelisation.
Indeed, the collective processing power of a GPU can exceed that of a single CPU for certain
tasks. Moreover, depending on the motherboard model, multiple GPUs can be installed on a single
desktop/workstation computer, boosting the potential processing power even further.

In a previous study, we demonstrated that a desktop computer with a GPU was able to robustly
optimise the parameters of a spiking neuron model to experimental eye movement data using a
multi-objective EA [Avramidis and Akman 2017]. As part of this work, we showed that the most
computationally intensive part of the optimisation process was the numerical integration of the
model over the multiple parameter combinations comprising the EA population at each genera-
tion [Avramidis and Akman 2017]. This task can be executed in an embarrassingly parallel fashion,
enabling the current trend for heterogenous HPC architectures to be exploited. Accordingly, in our
spiking neuron model, numerical integration was executed on the GPU, while all the EA operations
(mutation, crossover, fitness evaluation, etc.) were executed on the CPU [Avramidis and Akman
2017]. Utilising the GPU in this manner yielded a speedup of up to ≈20, compared with a high-end
CPU.

1.4 CUDA and OpenCL

Programs that are executed on a GPU have to be written using a specific programming framework.
The two most commonly used frameworks are Compute Unified Device Architecture (CUDA)
and Open Computing Language (OpenCL) [Demidov et al. 2013]. The CUDA framework is a
proprietary architecture specifically designed to be run on NVIDIA compute devices. By con-
trast, OpenCL is a royalty-free standard for general purpose parallel programming across CPUs
and GPUs, giving software developers portable and efficient access to the power of these het-
erogeneous processing platforms [Khronos OpenCL Working Group 2015]. OpenCL includes a
cross-platform intermediate language for writing functions (kernels), which are executed on

OpenCL-supported devices, together with an application programming interface (API) that is used to coordinate the parallel computations across these devices. A simple OpenCL program involves the following steps: identifying the OpenCL device, compiling the OpenCL code that will be run on the device, copying the data to the device, performing the computation, and copying the results back from the device.

A software library equipped with OpenCL functionality can use CPUs and GPUs, removing the need to develop different libraries for each processor type.

1.5 A New Library for Numerically Integrating SDE Models on OpenCL-supported Devices

Here, we present SODECL, a C++ library that uses OpenCL to calculate multiple orbits of an SDE (or ODE) system in an embarrassingly parallel way. We focus on presenting the library for solving SDE models, for which orbits are calculated using the Euler-Maruyama method [Higham 2001; Saito and Mitsui 1996]. ODE orbits can be computed using any of the following integration methods: Euler, Runge-Kutta, Implicit Euler, or Implicit Midpoint. SODECL has previously been used to fit both SDE and ODE versions of an oculomotor control model to experimentally recorded timeseries [Avramidis 2015; Avramidis and Akman 2017].

In the following sections, we describe the design principles of the SODECL library, its organisation, and the numerical algorithms used. Moreover, we describe the experimental protocols that were implemented to measure the execution speed of SODECL for different compute devices (i.e., CPUs and GPUs), and to assess the numerical stability and accuracy of our SDE solver. As part of the speed test experiments, we compare the performance of SODECL on a high-end multi-core CPU with a MATLAB program that also calculates multiple SDE orbits in parallel using the same integration method (i.e., Euler–Maruyama). Lastly, we outline directions for future development of the library.

2 THE SODECL LIBRARY

2.1 Design Principles

We chose to implement SODECL using OpenCL because it does not pose significant constraints on the computer hardware used. As mentioned above, whilst a user must run CUDA executables on an NVIDIA GPU, OpenCL executables can run on both NVIDIA GPUs and AMD/Intel GPUs. Moreover, OpenCL executables can run on Intel and AMD CPUs. This gives OpenCL a key advantage over CUDA in terms of disseminating research methods and results, and promoting scientific collaborations.

We designed SODECL to be relatively portable, modular, and easy to extend. We tested our library on the following operating systems: Windows 10 64-bit; Ubuntu 18.04 64-bit; macOS Sierra (10.12). The SODECL library can easily be integrated into any C++ source code by including the main header of the library and linking it to an OpenCL library. Furthermore, SODECL can be extended by adding OpenCL functions for other SDE integration methods (e.g., the Milstein method [Platen 1999]). The source code is released under the MIT License and is under version control with `git` at https://github.com/avramidis/sodecl.

2.2 Algorithms

The primary algorithms comprising SODECL are the numerical integration method, the noise generator, the procedure used to generate the OpenCL code at runtime, and the execution scheme for computing multiple SDE orbits. Each of these is now described in turn.

Table 1. List of Named Constants Used by the SODECL OpenCL Functions

Named constant	Description
numeq	Number of equations in the SDE system
numnoi	Number of noise variables in the SDE system
numpar	Number of parameters in the SDE system
_m_dt_	SDE integrator timestep in seconds
numsteps	Number of SDE integrator steps per OpenCL kernel call

2.2.1 The Integration Method. SODECL uses the Euler-Maruyama method [Higham 2001; Saito and Mitsui 1996] to integrate the equations in Equation (1). The Euler-Maruyama approximation $\{\mathbf{X}^k = (X_1^k, \ldots, X_n^k) : k \geq 1\}$ to the true solution $\{\mathbf{X}(t) : \mathbf{X}(t_0) = \mathbf{X}_0, t \geq t_0\}$ is defined by the recursion

$$X_i^{k+1} = X_i^k + f_i(t, \mathbf{X}^k, \mathbf{p})\Delta t + \sum_{j=1}^{m} g_{ij}(t, \mathbf{X}^k, \mathbf{p})\sqrt{\Delta t}N_j(0, 1), \tag{2}$$

where $\mathbf{X}^1 = \mathbf{X}_0$, Δt is the timestep (so that $\mathbf{X}^k$ is the approximation to $\mathbf{X}((k-1)\Delta t)$ for $k > 1$), and $N_j(0, 1)$ are independent, normally distributed random variables with zero mean and unit variance.

2.2.2 Noise Generator. To generate the noise variables $N_j(0, 1)$ used in each step of Equation (2), SODECL utilises the Random123 library [Salmon et al. 2011]. Random123 generates uniformly distributed random numbers, which are then converted to normally distributed random numbers using the Box-Muller algorithm [Box and Muller 1958]:

$$\begin{aligned} r_1 &= \sqrt{-2 \ln U_1} \cos(2\pi U_2), \\ r_2 &= \sqrt{-2 \ln U_1} \sin(2\pi U_2). \end{aligned} \tag{3}$$

Here, $\{U_1, U_2\}$ are two independent, uniformly distributed random variables over [0,1], and $\{r_1, r_2\}$ are two independent, normally distributed random variables with zero mean and unit variance. In the case where $j > 2$, the equations in Equation (3) are called multiple times.

2.2.3 OpenCL Code Generation. SODECL creates and builds the OpenCL source code string at runtime. This allows the user to change the f_i and g_{ij} functions in Equation (1) without the need to compile the source code. Moreover, SODECL allows the implementation of wrappers—e.g., for MATLAB or Python—the use of a script/function that runs the SODECL executable, or the use of a MATLAB .mex file without it being recompiled. The procedure for formulating the OpenCL source code string is performed in the four steps outlined below.

Step 1. Initially, SODECL appends the definitions of five named constants to the beginning of the empty OpenCL source code string (see Table 1). The values of these constants depend on the general properties of the SDE system (numbers of equations, noise terms, and parameters), the integrator's timestep, and the number of steps to be executed by the integrator in each OpenCL kernel call. These values are passed to the SODECL library by the user. The remainder of the OpenCL source code string is generated using the code located in three separate files, described in steps 2–4 below.

Step 2. The first file, `integrator_caller.cl`, contains the kernel function that is called by the host to run on an OpenCL device. This function is defined in Figure 1. The parameters of the kernel function are pointers that show the location of the arrays in the global memory of the device. These arrays contain the time points of the SDE system at which to approximate the solution, the values of the SDE system's dependent variables and the parameter combinations for which the

```
__kernel void integrator_caller ( __global double *t ,
                                   __global double *y ,
                                   __global double *params_g ,
                                   __global int  *counter_g )
```

Fig. 1. Definition of the SODECL kernel function. The function parameters are pointers showing the location of the arrays in the global memory of the OpenCL device. These arrays are the time points at which the solution to the SDE system is to be approximated (accessed by t), the corresponding values of the dependent variables (accessed by y), and the combinations of system parameters (accessed by params_g). One additional array of counters (accessed by counter_g) is passed to the kernel for use by the Random123 library.

```
void system_integrator (double t ,
                        double y [ _numeq_ ] ,
                        double yout [ _numeq_ ] ,
                        double p [ _numpar_ ] ,
                        double noise [ _numnoi ])
```

Fig. 2. Definition of the SODECL system_integrator function. The function parameters are the current time value of the SDE system (t), the corresponding values of the dependent variables (y), the parameter values (p), the current noise values (noise), and the new values of the dependent variables calculated by the Euler-Maruyama method (yout).

different orbits are to be calculated. One additional array of counters is passed to the kernel for use by the Random123 library. The kernel generates the independent, normally distributed numbers required for the integration of the SDE system. Subsequently, the same kernel calls the function that integrates the SDE system for one timestep (see step 3). The total number of calls is defined by the constant _numsteps_. Having integrated the SDE system _numsteps_ times, the kernel copies the last integrated state values to the global memory of the OpenCL device, to be used in the next kernel step and accessed by the host.

Step 3. The second file used in the creation of the OpenCL source is named stochastic_euler.cl. It contains the function system_integrator (defined in Figure 2), which implements the Euler-Muryama integration method. system_integrator itself calls functions that calculate the deterministic and stochastic components of the SDE system (see step 4). Once these have been evaluated, stochastic_euler.cl calculates the values of all dependent variables at the next timestep.

Step 4. The third file contains two functions called sode_system and sode_system_stoch, which calculate the drift and diffusion terms f_i and g_{ij} in the numerical scheme (2), respectively. Their definitions are shown in Figure 3, and they have to be specified by the user (an example is shown in Figure 5).

2.2.4 SDE Integrator Execution Algorithm. The logical flow of the SODECL integration execution algorithm is predominately a loop. During each loop iteration, the corresponding values of the dependent variables for all orbits are copied to the host device, and the copied values are saved in parallel to an array in preparation for the next kernel call. The OpenCL kernel is called to integrate the model for _numsteps_ steps of length _m_dt_, which means that the output of the integration is saved to the array every (_numsteps_ × _m_dt_) seconds of integration time. This allows the data to be stored with a frequency of 1/(_numsteps_ × _m_dt_) Hz. Once the integration over the required time span is completed for all parameter combinations, the algorithm ends. The number

```
void sode_system(double t,
                 double y[_numeq_],
                 double yout[_numeq_],
                 double p[_numpar_])

void sode_system_stoch(double t,
                       double y[_numeq_],
                       double stoch[_numeq_],
                       double p[_numpar_],
                       double noise[_numnoi_])
```

Fig. 3. Definition of the SODECL functions specifying the SDE system. The function parameters are the current time value (t), the corresponding values of the dependent variables of the SDE system (y), the new values of the dependent variables of the deterministic component of the SDEs (yout), the model parameters (p), the noise values (noise), and the new values of the stochastic components of the SDEs (stoch).

```
def sodecl( platform, device, kernel,
            initx, params, solver,
            orbits, nequat, nparams,
            nnoi, dt, tspan,
            ksteps, localgroupsize )
```

Fig. 4. Definition of the SODECL Python wrapper function. The function is used to execute the SODECL executable from within Python. Each argument is defined in Table 2.

of loop iterations, k, is calculated using the equation

$$k = t_{total}/(\Delta t \cdot s), \tag{4}$$

where t_{total} is the time span of the integration, Δt is the integrator timestep, and s is the number of steps the integrator performs in each OpenCL kernel call (i.e., the number of parallel operations, which could include multiple parameter combinations and/or initial conditions). The values of t_{total}, Δt, and s are supplied by the user.

The storing frequency of the results plays an important role in the speedup provided by the GPU compared to the CPU. This is due to the bottleneck caused by the transfer to and from the GPU memory, and the time required to store the data in the host memory for further analysis. A low frequency (250–1,000 Hz) avoids substantial data storage in the host memory. Also, the model size is an important factor, since it sets the number of model integrations that are to be saved on the computer's RAM. Moreover, a larger number of returned data points can increase the analysis time; a low number, however, may be insufficient for an accurate analysis [Avramidis and Akman 2017].

2.2.5 Python Interface. The Python package interface function for SODECL is defined in Figure 4, whilst Table 2 describes the arguments passed to the SODECL executable by the function. The Python library pybind11 (https://github.com/pybind/pybind11) was used to implement the interface. A MATLAB function interface is also provided with SODECL, although it should be noted that this is currently experimental.

2.3 SODECL Speed Evaluation

2.3.1 Protocol. To evaluate the execution speed of SODECL, we used a stochastic variant of the well-established Kuramoto model for describing the dynamics of a population of weakly coupled phase oscillators [Acebrón et al. 2005; Bick et al. 2011; Daido 1992; Nakao 2016; Strogatz 2000]. The

Table 2. Arguments of the SODECL Python Interface

Argument	Variable	Description
1	platform	OpenCL platform number
2	device	OpenCL device number of the selected platform
3	kernel	Path of the file with the OpenCL function defining the SDE system
4	initx	Initial conditions for each orbit of the SDE system
5	params	Parameter sets of the SDE system for all orbits
6	solver	SDE integrator
7	orbits	Number of orbits to be calculated
8	nequat	Number of equations of the SDE system
9	nparams	Number of parameters of the SDE system
10	nnoi	Number of noise processes
11	dt	SDE solver timestep
12	tspan	Integration timespan
13	ksteps	Number of SDE integrator steps executed in each OpenCL kernel call
14	localgroupsize	OpenCL local group size

Kuramoto model is commonly used to model synchronisation of biological oscillators [Acebrón et al. 2005; Bick et al. 2011] and is relatively straightforward to implement and customise (e.g., the oscillator population size can be easily varied). The SDE Kuramoto system we used here has the form

$$d\theta_i(t) = \omega_i + \frac{K}{N} \sum_{j=1}^{N} \sin(\theta_j(t) - \theta_i(t)) + p_i dW_i(t), \tag{5}$$

where $\{\theta_1, \ldots, \theta_N : -\pi \leq \theta_i < \pi\}$ specify the phase of each oscillator; ω_i is the free-running frequency of the ith oscillator; $\{W_1(t), \ldots, W_N(t)\}$ are independent, scalar Wiener processes; p_i controls the strength of the noise effect on the ith oscillator; and K is the strength of the mean-field coupling. The system parameters are thus $\mathbf{p} = (K, \omega_1, \ldots, \omega_N, p_1, \ldots, p_N)$.[1] The initial value of each phase variable θ_i was taken to be uniformly distributed over $[-\pi, \pi)$, the frequencies ω_i to be uniformly distributed over $[0.01, 0.03]$ and the noise strengths p_i to be uniformly distributed over $[0.001, 0.003]$. The following equations define the integration scheme for Equation (5) using the Euler-Maruyama method (cf. Equation (2) above):

$$\theta_i^{k+1} = \theta_i^k + \left(\omega_i + \frac{K}{N} \sum_{j=1}^{N} \sin\left(\theta_j^k - \theta_i^k\right) \right) \Delta t + p_i \sqrt{\Delta t} N_i(0, 1). \tag{6}$$

We set the coupling constant K in Equation (5) to 1 and integrated the model for 400 s with a timestep Δt of 0.05 s. Moreover, we used a double-precision floating-point data type for better accuracy. The SODECL implementation of the stochastic Kuramoto model is shown in Figure 5.

We compared the execution speed of SODECL when using different multi-core and many-core compute devices with the execution speed when using only one core of an HPC CPU. The compute devices used to evaluate SODECL performance are listed in Table 3. The CPU with ID I4790K is a high-end desktop CPU, whereas the 2X6142 is an HPC node with two many-core CPUs. The W8100 and P100 are workstation/server GPUs, with high double-precision compute capabilities required for scientific computing. To gain a better understanding of how the SODECL runtime is

---

[1]In terms of Equation (1), $X_i = \theta_i$, $n = m = N$, $f_i = \omega_i + \frac{K}{N} \sum_{j=1}^{N} \sin(\theta_j - \theta_i)$ and $g_{ij} = p_i \delta_{ij}$.

```
#define NOSC _numeq_

void sode_system(double t,
                 double y[_numeq_],
                 double yout[_numeq_],
                 double p[_numpar_])
{
  for (int i = 0; i<NOSC; i++)
  {
    yout[i] = 0;
    for (int j = 0; j<NOSC; j++)
    {
      yout[i] = yout[i] + (sin(y[j] - y[i]));
    }
    yout[i] = yout[i] * (p[0] / NOSC);
    yout[i] = yout[i] + p[i+1];
  }
}

void sode_system_stoch(double t,
                       double y[_numeq_],
                       double stoch[_numeq_],
                       double p[_numpar_],
                       double noise[_numnoi_])
{
  for (int i = 0; i < NOSC; i++)
  {
    stoch[i] = p[_numpar_ - _numnoi_ + i] * noise[i];
  }
}
```

Fig. 5. SODECL implementation of the stochastic Kuramoto model (Equation (5)). The first function sode_system is used to calculate the deterministic component of the model, whilst the second function sode_system_stoch is used to calculate the stochastic part.

Table 3. Hardware Used for the Speed
Evaluation Tests

ID	Type	Model
I4790K	CPU	Intel Core i7-4790K
2X6142	CPU	2 x Intel Xeon Gold 6142
W8100	GPU	AMD Firepro W8100
P100	GPU	NVIDIA Tesla P100-PCIE

affected by the size of the system, we varied the number of equations in the Kuramoto model (i.e., we varied the number of oscillators N in the network). Moreover, we examined how the number of orbits M being integrated affects the performance of SODECL on the different compute devices. Eight independent runs of the solver were executed for each choice of N and M.

We also compared the execution speed of SODECL on the desktop CPU against an equivalent parallel MATLAB implementation running on the same CPU. In order to further accelerate this implementation, described in Figure 6, we used the MATLAB Coder to convert MATLAB functions into C++ code. The parallelisation of MATLAB was performed automatically with OpenMP (Open

```
function yout = kuramotoParallel(pop, nequat, x_y0, x_params, x_noise)

  yout=zeros(pop,26);

  parfor (p=1:pop,8)
    y=kuramoto_system(nequat, x_y0(p,:), ...
    x_params(p,:), x_noise(p,:));
    yout(p,:)=y;
  end
end

function yout = kuramoto_system(nequat, init, p, x_noise )
  dt=5e-2;
  sdt=sqrt(dt);
  yout=zeros(1,26);

  for ii=1:25
    for i=1:40
      y=zeros(1,nequat);
      for k=1:nequat
        y(1,k)=0;
        for j=1:nequat
          y(1,k)=y(1,k)+sin(init(1,j)-init(1,k));
        end
        y(1,k)=p(1,k)+y(1,k)*p(1,nequat+1)/nequat;
      end

      for j=1:nequat
        init(1,j)=init(1,j)+y(1,j)*dt+x_noise(1,j)*sdt*randn(1);
      end

      for k=1:nequat
        y(1,k)=init(1,k);
      end
    end
    yout(ii+1)=y(1,1);
  end
end
```

Fig. 6. MATLAB function for calculating orbits of the stochastic Kuramoto model in parallel. The first function kuramotoParallel calls the second function kuramoto_system—which calculates one orbit of the model—multiple times.

Multi-Processing), an application programming interface for parallel applications that uses the cores of CPUs and coprocessors [Dagum and Menon 1998].

2.3.2   Results. The comparative execution speeds and speedups obtained using SODECL with different compute devices, numbers of orbits, and system sizes are shown in Figure 7 (see Tables S1–S6 for the corresponding numerical values). For each compute device, the speedup was calculated relative to one core of the 2X6142 node (i.e., as the ratio of the 2X6142 single core runtime to the compute device runtime). The local group sizes used for each device and system size (N) are shown in Table 4. The local group size is the number of work items that will run in parallel and can communicate. In the case of embarrassingly parallel applications, although there is no communication, the group size can have an effect on the runtime. The values shown were found
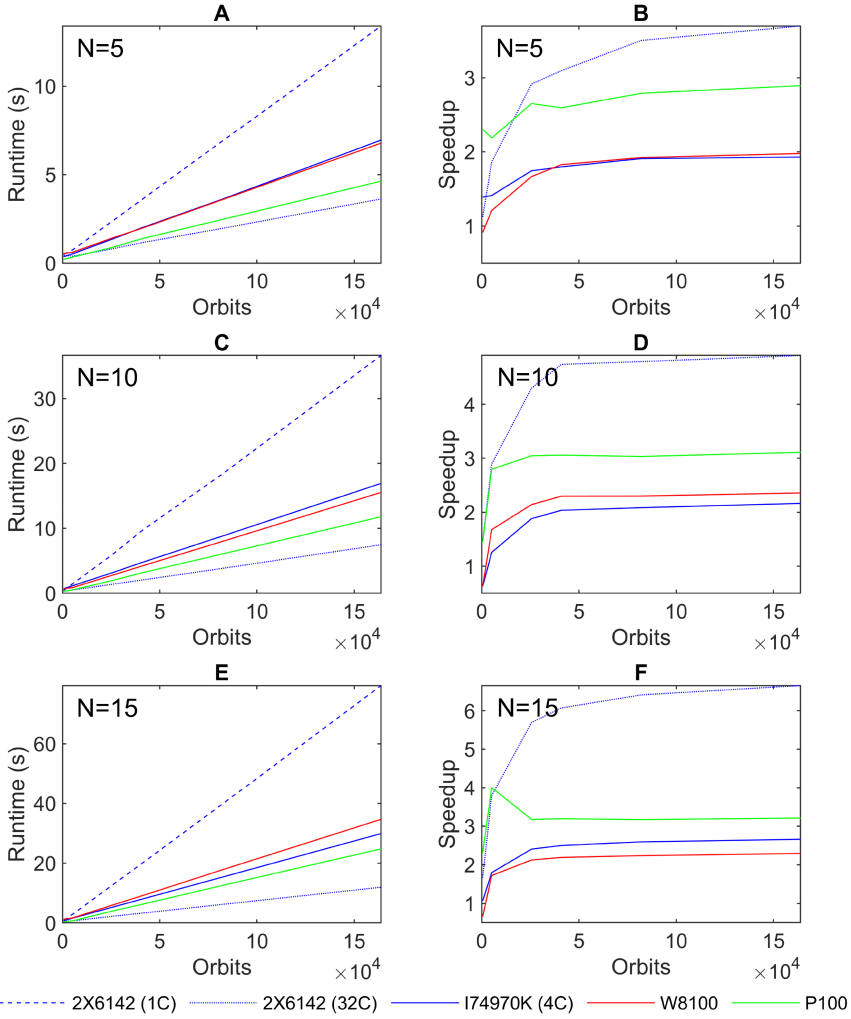
Fig. 7. Runtimes in seconds (s) and speedups for the stochastic Kuramoto model as a function of the number of orbits for varying system sizes (N), when using different parallel computer hardware implementations. Plot A shows the runtimes for $N = 5$, plot C for $N = 10$, and plot E for $N = 15$. The values are means from eight runs. Plots B, D, and F show the corresponding speedups for $N = 5$, $N = 10$, and $N = 15$, respectively, relative to one logical processor of the 2X6142. The hardware used in each case is listed in Table 3.

Table 4. Local Group Sizes Used for Each Compute
Device to Generate the Runtime Results Shown in
Tables S1–S6 and Plotted in Figure 7.

Model size, N	I4790K	2X6142	W8100	P100
5	8	32	256	8
10	8	32	16	8
15	8	32	32	8

by doing exploratory tests, since, to our knowledge, there is no analytical formula for calculating the optimal local group size.

For the CPUs tested, the results indicate that when using all 32 cores of the 2X6142 node, SODECL is between ≈ 1.12 and ≈ 6.65 times faster than one core of the same node across different system sizes and orbit numbers. By contrast, when using all cores of the I4790K CPU and comparing its speed to one core of 2X6142, we observe speedups ranging from ≈ 0.69 to ≈ 2.66 as the system size is increased from 5 to 15 and the number of orbits is increased from 512 to 163,840. The greater speedup obtained overall with 2X6142 is due to its larger number of cores and newer architecture.

For the GPUs tested, SODECL yields speedups in the range 0.6–4.5 compared to using one core of 2X6142, depending on the size of the Kuramoto model and the number of orbits. Larger numbers of model equations and orbits allow for greater use of GPU resources, which translates to improved speedup. When comparing the GPU runtimes to those obtained when all cores of 2X6142 are utilised, it can be seen that the GPUs are generally slower. This could, for example, be due to higher initialisation time overheads.

Finally, Figure 8 compares the runtimes obtained with the SODECL and MATLAB implementations of the Kuramoto model solver on the I4790K CPU (the corresponding numerical values are given in Tables S7–S9). The MATLAB implementation is only faster than SODECL for the smallest system size and lowest number of orbits (see Figure 8(B), (E), and (H)). In all other cases, SODECL is up to ≈ 5.8 times faster.

2.4 Accuracy and Numerical Stability Tests

2.4.1 Protocol. To assess the accuracy and numerical stability of SODECL, we generated a set of further simulations with $N = 100$, $\omega_i \sim \mathcal{U}(0.2, 0.4)$, $\phi_i(0) \sim \mathcal{U}(-\pi, \pi)$, and $p_i \sim \mathcal{U}(0.01, 0.03)$ for two values $K_1 = 0.02$ and $K_2 = 0.2$ of the global coupling strength K. These values were chosen to lie on either side of the Kuramoto transition that occurs in the deterministic system in the continuum limit ($N \rightarrow \infty$) at $K = K_c = \frac{2}{5\pi} \approx 0.1273$. As K is increased through this critical value, macroscopic mutual entrainment (MME)—a collective synchronised rhythm with a common frequency—is observed in the oscillator population [Daido 1992; Strogatz 2000]. For both values of K, 64 independent integrations were carried out over 400 s (around 19 cycles of the mean period 20.94 s), using the following integration timesteps: $\Delta t = \frac{2^{l-5}}{5}$, $1 \leq l \leq 5$. For each realisation, the degree of synchronisation of the simulated oscillator population was quantified using the complex order parameter [Nakao 2016; Strogatz 2000] defined below:

$$r(t)e^{i\Phi(t)} = \frac{1}{N} \sum_{j=1}^{N} e^{i\theta_j(t)}. \tag{7}$$

In the above equation, the radius $r(t)$ measures the phase coherence and $\Phi(t)$ measures the collective phase, with r values of 0 and 1 corresponding to complete desynchronisation and complete synchronisation of the population, respectively [Nakao 2016; Strogatz 2000]. In the continuum limit of the deterministic model, increasing the coupling strength K through the critical value K_c causes the steady state coherence $\lim_{t \rightarrow \infty} r(t)$ to increase rapidly from 0 to 1 as the population becomes globally synchronised [Acebrón et al. 2005; Nakao 2016; Strogatz 2000]. For finite N, values of K less than K_c yield $O(N^{-1/2})$ fluctuations in $r(t)$, while values of K greater than K_c result in $r(t)$ saturating at a value $\lim_{t \rightarrow \infty} r(t) < 1$, with $O(N^{-1/2})$ fluctuations [Strogatz 2000].

2.4.2 Results. Figure 9 plots how the mean phase coherence $\langle r(t) \rangle$ and the standard deviation of the phase coherence $\sigma(r(t))$ vary with time for the two K values when $\Delta t = 0.05$. As expected, for $K = K_1 < K_c$, $\langle r(t) \rangle$ exhibits small-amplitude oscillations close to 0, while for $K = K_2 > K_c$,
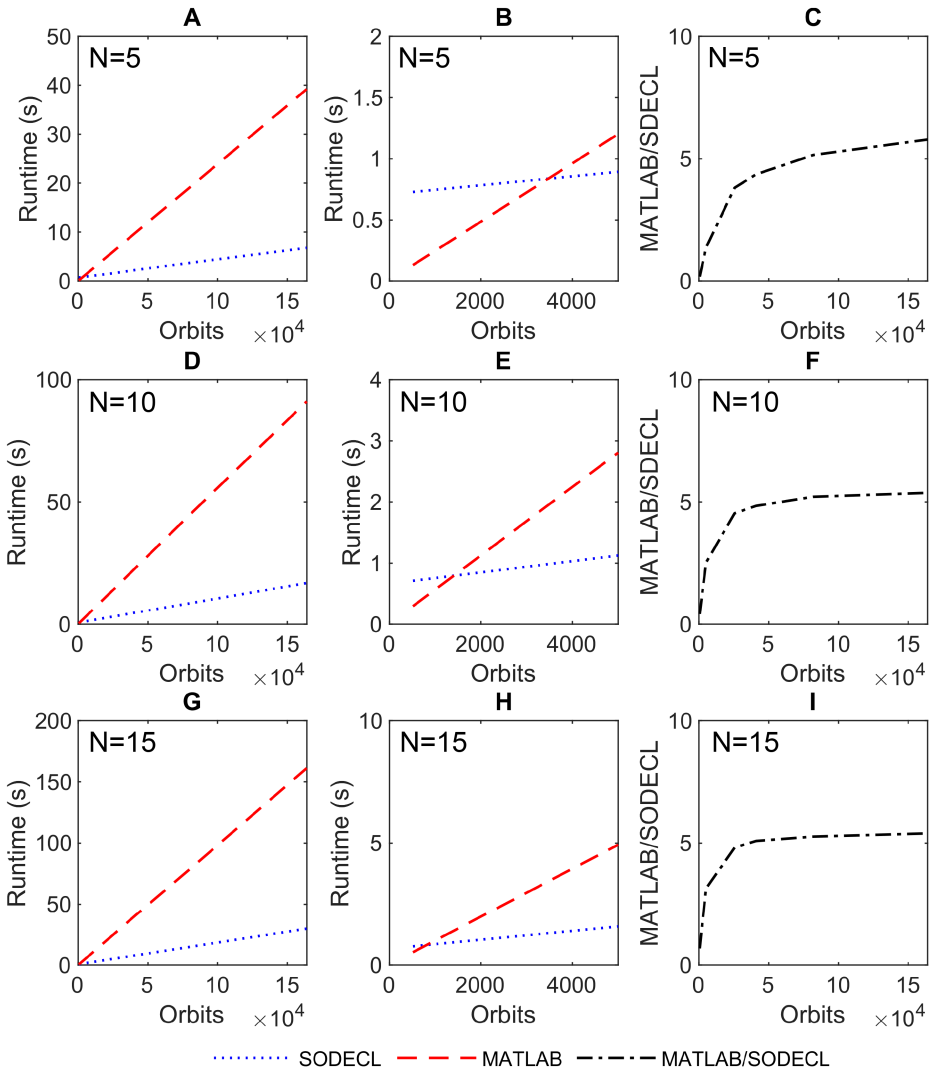
Fig. 8. Runtime as a function of the number of orbits for the stochastic Kuramoto model for varying system sizes (N), using MATLAB and SODECL implementations on all logical processors of the I4790K CPU. Runtimes are given in seconds (s). Plots A and B show results for $N = 5$, D and E for $N = 10$, and G and H for $N = 15$. Plots B, E, and H show the runtimes obtained for the first 5,000 orbits. Plots C, F, and I show the relative speeds of the MATLAB and SODECL implementations for $N = 5$, 10, and 15, respectively.

$\langle r(t) \rangle$ asymptotes to a value close to 1. Kymographs showing the temporal evolution of the population phases for a typical realisation generated with each K value are plotted in Figure 10. These are consistent with the phase coherence plots—the oscillators are desynchronised throughout the integration interval for $K = K_1$, while for $K = K_2$, the oscillators synchronise after around 50 s and remain synchronised thereafter. Finally, Figure 11 plots how the values of $\langle r(t) \rangle$ and $\sigma (r(t))$ at the end of the integration interval $t_{MAX} = 400$ vary with the integration timestep Δt. It can be seen that for both K values, $\langle r(t_{MAX}) \rangle$ and $\sigma (r(t_{MAX}))$ exhibit a weak dependence on Δt, suggesting that the integration algorithm is numerically stable [Kloeden and Platen 2011].
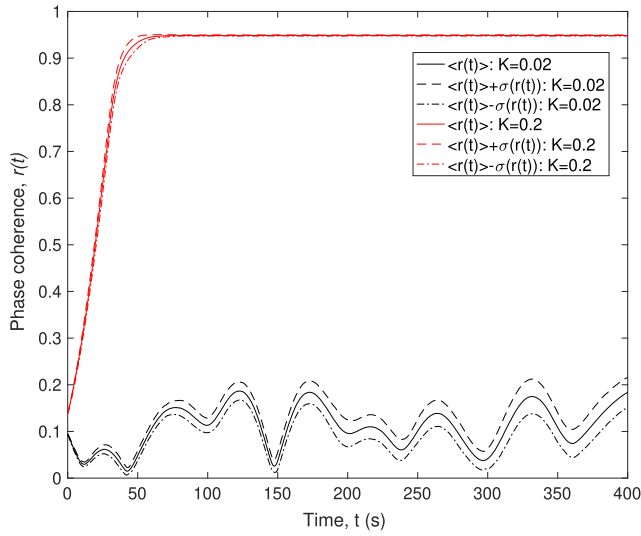
Fig. 9. Temporal evolution of phase coherence $r(t)$ in the stochastic Kuramoto model (Equation (5)) when $N = 100$ for $K = 0.02$ (black lines) and $K = 0.2$ (red lines). $\langle r(t)\rangle$ and $\sigma(r(t))$ denote the mean and standard deviation of $r(t)$, respectively, calculated from 64 independent realisations of Equation (6) with an integration timestep $\Delta t = 0.05$. Distributions of oscillator frequencies ω_i, initial phases $\theta_i(0)$, and noise strengths p_i were as described in the text.

Fig. 10. Temporal evolution of oscillator phases $\theta_i(t)$ in the stochastic Kuramoto model. Each kymograph corresponds to one of the realisations of Equation (6) used to compute the phase coherence statistics shown in Figure 9. Plots A and B shows simulations for $K = 0.02$ and $K = 0.2$, respectively.

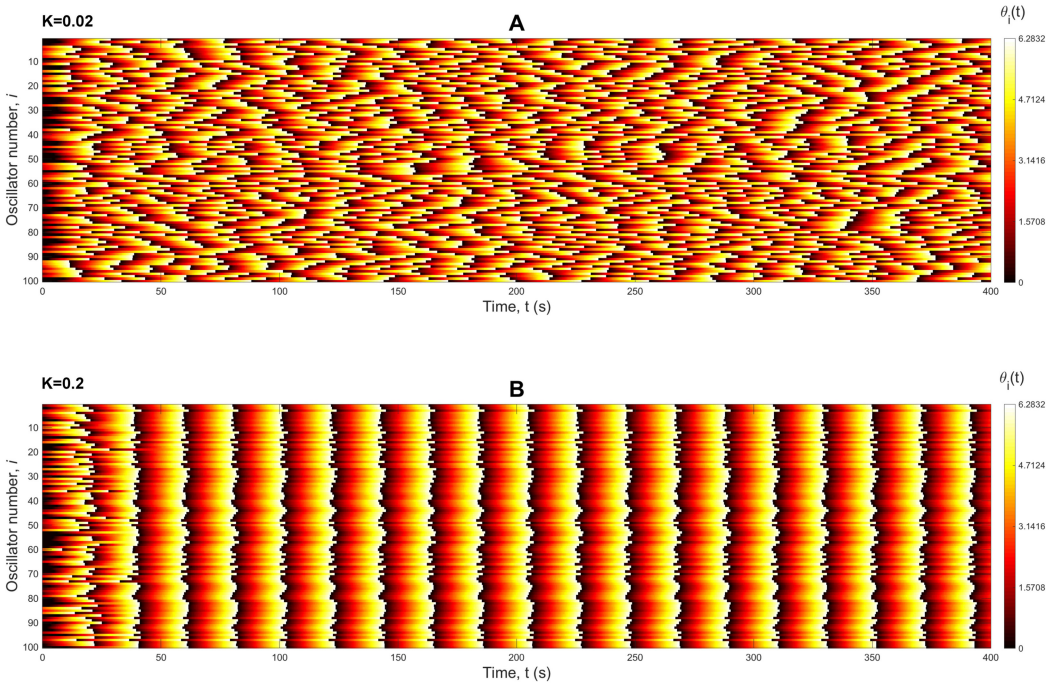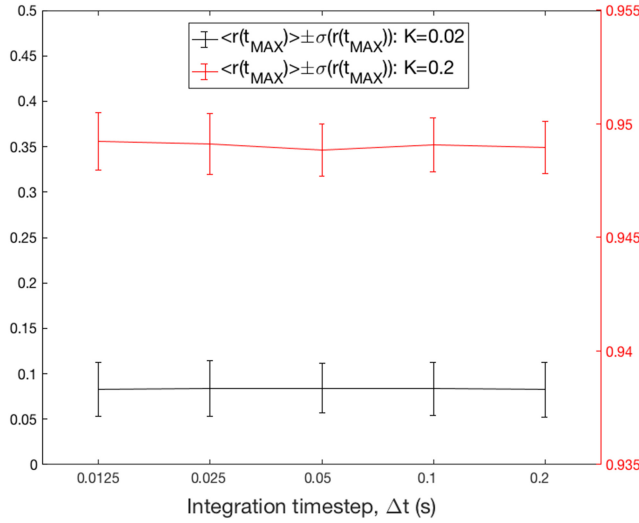Fig. 11. Dependence on the integration timestep Δt of the phase coherence statistics $\langle r(t) \rangle$ and $\sigma(r(t))$ at the end of the integration interval $t_{MAX} = 400$. Black lines and axes show results for $K = 0.02$; red lines and axes show results for $K = 0.2$. In each case, $\langle r(t_{MAX}) \rangle$ and $\sigma(r(t_{MAX}))$ were computed from 64 independent realisations of Equation (6) with $N = 100$. Distributions of oscillator frequencies ω_i, initial phases $\theta_i(0)$, and noise strengths p_i were as described in the text. Note the logarithmic scale on the x-axis.

3   CONCLUSIONS

We have described the design and performance of SODECL, an open source C++ library that uses OpenCL to calculate multiple orbits of a SDE system in an embarrassingly parallel way. The advantage of SODECL is that it can use the parallel capabilities of multi-core CPUs and GPUs to reduce computation time, thereby allowing large numbers of simulations to be performed simultaneously. This capability is critical when using population-based metaheuristics—such as genetic algorithms—for model calibration, as it facilitates the optimisation and analysis of SDE models in a practical time frame [Adams et al. 2013]. SODECL was designed to compile and run on Windows, Linux, and macOS operating systems, to be user-friendly, fast, and fairly easy to extend.

Although execution speed was not the main priority in our design of SODECL, we showed that it was faster in almost all cases when integrating the stochastic Kuramoto system compared to a parallel MATLAB implementation. We also showed that when SODECL utilises a GPU or many-core CPUs, there is considerable speedup compared to a high-end desktop CPU. However, the speedup can be affected by the type and size of the SDE system being integrated. It would therefore be instructive to obtain performance benchmarks for other canonical stochastic models.

In our previous work [Avramidis and Akman 2017], SODECL achieved a speedup of ≈ 20 on the W8100 GPU compared to all cores of the I4790K CPU. By contrast, the greatest speedup obtained with the same GPU against I4790K in this study was ≈ 1.16. We believe that the difference in speedups between our previous and current studies is that in the work of Avramidis and Akman [2017], the SODECL OpenCL code could not be vectorised efficiently to facilitate better utilisation of the CPU, for the following reasons. Firstly, the system was a deterministic model with six heterogeneous equations (i.e., equations containing different functional forms). Similar equations, if vectorised, could run in parallel more efficiently. Secondly, two of these equations included a case mathematical function. Case functions are implemented with one or more if statements that cause branch divergence. Most of the time, branch divergence cannot be predicted and a compiler

cannot vectorise the code for branches in which each case implements a different mathematical operation. Thirdly, for the numerical integration of the model, the implicit, midpoint Euler method was used. This method includes a loop with a variable iteration number. All of the above factors hinder the compiler from efficiently vectorising the execution of the code on a parallel device.

There are a number of aspects of SODECL that we did not examine, which could provide interesting avenues for future work. Firstly, we only measured SODECL execution speed using double-precision, rather than single-precision, floating-point data type. Single-precision can be used if high accuracy is not necessary and has the advantage that compute devices yield significantly better execution speeds when using lower precision. Secondly, we did not measure the OpenCL initialisation and finalisation time in SODECL. This would indicate whether different compute devices require more time for these tasks, perhaps explaining why the MATLAB implementation is comparable to SODECL for small numbers of orbits. Thirdly, we did not evaluate the effects of branch divergence on different compute devices using the Euler-Maruyama method. This potentially would have shown which devices are less affected. However, to assess the extent to which performance is affected by equations with many branches, a more extensive study would need to be carried out involving a broader range of SDE models. Lastly, we did not examine the memory requirements for different orbit numbers and model sizes. A follow-up study could examine the impact that the high memory requirements of larger models (i.e., more than 100 equations) could have on the performance of different compute devices.

We remark that the future development of SODECL would involve incorporating additional methods for numerically integrating SDE systems, such as the Milstein and Runge-Kutta schemes [Platen 1999]. Here, we have implemented the Euler-Maruyama numerical integration scheme, which, despite its simplicity, is still widely used, in part owing to the relative ease of implementation. We note, however, that the inefficiency of the Euler-Maruyama method compared to more sophisticated numerical schemes is offset by the significant acceleration conferred by parallelising the computation of trajectories across multiple CPU cores or GPU compute units. The incorporation of additional solvers would involve modifying the SODECL `system_integrator` function so as to implement the chosen schemes (see Figure 2).

Also, additional optimisations could be explored for both GPUs and CPUs, such as using newer versions of OpenCL. This could potentially further accelerate SODECL on certain devices. Furthermore, we believe that splitting the integration algorithm over multiple OpenCL kernels could also potentially increase the execution speed.

SUPPLEMENTARY TABLES

Table S1. Runtimes in Seconds (s) for the Stochastic Kuramoto Model as a Function of the Number of Orbits for System Size $N = 5$, When Using Different Parallel Computer Hardware Implementations

Orbits	2X6142 (1C)	2X6142 (32C)	I4790K (4C)	W8100	P100
512	0.504	0.449	0.728	0.552	0.201
5,120	0.746	0.401	0.898	0.617	0.363
25,600	2.373	0.814	1.612	1.424	0.935
40,960	3.602	1.163	2.254	1.973	1.400
81,920	6.869	1.961	3.800	3.573	2.512
163,840	13.390	3.623	6.779	6.772	4.719

The hardware used in each case is shown in Table 3.

Table S2. Runtimes in Seconds (s) for the Stochastic Kuramoto Model as a Function of the Number of Orbits for System Size $N = 10$, When Using Different Parallel Computer Hardware Implementations

Orbits	2X6142 (1C)	2X6142 (32C)	I4790K (4C)	W8100	P100
512	0.441	0.301	0.714	0.708	0.280
5,120	1.444	0.499	1.138	0.864	0.513
25,600	5.956	1.385	3.160	2.784	1.714
40,960	9.676	2.042	4.746	4.211	2.566
81,920	18.308	3.821	8.784	7.981	4.828
163,840	36.641	7.471	16.939	15.589	9.315

The hardware used in each case is shown in Table 3.

Table S3. Runtimes in Seconds (s) for the Stochastic Kuramoto Model as a Function of the Number of Orbits for System Size $N = 15$, When Using Different Parallel Computer Hardware Implementations

Orbits	2X6142 (1C)	2X6142 (32C)	I4790K (4C)	W8100	P100
512	0.8247	0.495	0.772	1.271	0.323
5,120	2.8846	0.759	1.605	1.679	0.672
25,600	12.6576	2.223	5.256	5.968	2.940
40,960	19.9707	3.288	7.987	9.128	4.566
81,920	39.6412	6.185	15.279	17.736	9.056
163,840	79.5124	11.953	29.882	34.700	17.656

The hardware used in each case is shown in Table 3.

Table S4. Speedups Obtained for the Stochastic Kuramoto Model as a Function of the Number of Orbits for System Size $N = 5$, When Using Each of the Compute Devices Listed in Table 3

Orbits	2X6142	I4790K	W8100	P100
512	1.122	0.692	0.914	2.498
5,120	1.861	0.831	1.209	2.051
25,600	2.915	1.471	1.666	2.537
40,960	3.095	1.597	1.826	2.572
81,920	3.501	1.807	1.922	2.733
163,840	3.696	1.975	1.977	2.837

Speedups are calculated as the runtime for one core of the 2X6142 CPU divided by the runtime of the compute device. In the case of the CPUs (2X6142 and I4790K), all cores were used.

Table S5. Speedups Obtained for the Stochastic
Kuramoto Model as a Function of the Number of
Orbits for System Size $N = 10$, When Using Each
of the Compute Devices Listed in Table 3

Orbits	2X6142	I4790K	W8100	P100
512	1.462	0.617	0.622	1.574
5,120	2.895	1.269	1.671	2.814
25,600	4.301	1.884	2.139	3.474
40,960	4.738	2.039	2.298	3.771
81,920	4.791	2.084	2.294	3.792
163,840	4.905	2.163	2.350	3.934

Speedups are calculated as the runtime for one core of
the 2X6142 CPU divided by the runtime of the compute
device. In the case of the CPUs (2X6142 and I4790K), all
cores were used.

Table S6. Speedups Obtained for the Stochastic
Kuramoto Model as a Function of the Number of
Orbits for System Size $N = 15$, When Using Each
of the Compute Devices Listed in Table 3

Orbits	2X6142	I4790K	W8100	P100
512	1.664	1.068	0.649	2.552
5,120	3.796	1.796	1.718	4.289
25,600	5.692	2.408	2.121	4.305
40,960	6.073	2.500	2.188	4.373
81,920	6.409	2.594	2.235	4.377
163,840	6.652	2.661	2.291	4.503

Speedups are calculated as the runtime for one core of
the 2X6142 CPU divided by the runtime of the compute
device. In the case of the CPUs (2X6142 and I4790K), all
cores were used.

Table S7. Runtimes for the Stochastic Kuramoto Model as
a Function of the Number of Orbits for System Size $N = 5$,
Using SODECL and Parallel MATLAB Implementations on
All Cores of the I4970K CPU

Orbits	SODECL	MATLAB	MATLAB/SODECL
512	0.728	0.131	0.180
5,120	0.898	1.229	1.369
25,600	1.612	6.149	3.813
40,960	2.254	9.841	4.364
81,920	3.800	19.588	5.154
163,840	6.779	39.213	5.784

Runtimes are given in seconds (s). The last column (MAT-
LAB/SODECL) shows the ratio of the runtimes of the MATLAB and
SODECL implementations (the speedup). Speedup values greater
than one indicate that SODECL was faster.

Table S8. Runtimes for the Stochastic Kuramoto Model as
a Function of the Number of Orbits for System Size
$N = 10$, Using SODECL and Parallel MATLAB
Implementations on all Cores of the I4970K CPU

Orbits	SODECL	MATLAB	MATLAB/SODECL
512	0.714	0.297	0.415
5,120	1.138	2.875	2.525
25,600	3.161	14.362	4.543
40,960	4.747	23.005	4.846
81,920	8.780	45.837	5.218
163,840	16.939	91.111	5.378

Runtimes are given in seconds (s). The last column (MAT-
LAB/SODECL) shows the ratio of the runtimes of the MATLAB and
SODECL implementations (the speedup). Speedup values greater
than one indicate that SODECL was faster.

Table S9. Runtimes for the Stochastic Kuramoto Model as
a Function of the Number of Orbits for System Size
$N = 15$, Using SODECL and Parallel MATLAB
Implementations on All Cores of the I4970K CPU

Orbits	SODECL	MATLAB	MATLAB/SODECL
512	0.772	0.529	0.686
5,120	1.606	5.050	3.144
25,600	5.256	25.322	4.817
40,960	7.987	40.642	5.087
81,920	15.279	80.467	5.266
163,840	29.882	161.250	5.396

Runtimes are given in seconds (s). The last column (MAT-
LAB/SODECL) shows the ratio of the runtimes of the MATLAB and
SODECL implementations (the speedup). Speedup values greater
than one indicate that SODECL was faster.

REFERENCES

Juan A. Acebrón, Luis L. Bonilla, Conrad J. Pérez Vicente, Félix Ritort, and Renato Spigler. 2005. The Kuramoto model: A simple paradigm for synchronization phenomena. *Rev. Mod. Phys.* 77, 1 (2005), 137.

Richard Adams, Allan Clark, Azusa Yamaguchi, Neil Hanlon, Nikos Tsorman, Shakir Ali, Galina Lebedeva, Alexey Goltsov, Anatoly Sorokin, Ozgur E. Akman, Carl Troein, Andrew J. Millar, Igor Goryanin, and Stephen Gilmore. 2013. SBSI: An extensible distributed software infrastructure for parameter estimation in systems biology. *Bioinformatics* 29, 5 (2013), 664–665.

Ozgur E. Akman, James C. W. Locke, Sanyi Tang, Isabelle Carré, Andrew J. Millar, and David A. Rand. 2008. Isoform switching facilitates period control in the Neurospora crassa circadian clock. *Mol. Syst. Biol.* 4 (2008), 164.

Ozgur E. Akman, David A. Rand, Paul E. Brown, and Andrew J. Millar. 2010. Robustness from flexibility in the fungal circadian clock. *BMC Syst. Biol.* 4, 1 (2010), 88.

Ozgur E. Akman, Steven Watterson, Andrew Parton, Nigel Binns, Andrew J. Millar, and Peter Ghazal. 2012. Digital clocks: Simple Boolean models can quantitatively describe circadian systems. *J. Roy. Soc. Interface* 9, 74 (2012), 2365–2382.

Edward J. Allen, Linda J. S. Allen, Armando Arciniega, and Priscilla E. Greenwood. 2008. Construction of equivalent sto-chastic differential equation models. *Stoch. Anal. Appl.* 26, 2 (2008), 274–297.

Maksat Ashyraliyev, Yves Fomekong-Nanfack, Jaap A. Kaandorp, and Joke G. Blom. 2009. Systems biology: Parameter estimation for biochemical models. *FEBS J.* 276, 4 (2009), 886–902.

Eleftherios Avramidis. 2015. *Optimisation and Computational Methods to Model the Oculomotor System with Focus on Nystagmus.* Ph.D. Dissertation. University of Exeter.

Eleftherios Avramidis and Ozgur E. Akman. 2017. Optimisation of an exemplar oculomotor model using multi-objective genetic algorithms executed on a GPU-CPU combination. *BMC Syst. Biol.* 11 (2017), 1.

Christian Bick, Marc Timme, Danilo Paulikat, Dirk Rathlev, and Peter Ashwin. 2011. Chaos in symmetric phase oscillator networks. *Phys. Rev. Lett.* 107, 24 (2011), 244101.

G. E. P. Box and Mervin E. Muller. 1958. A note on the generation of random normal deviates. *Ann. Math. Statist.* 29, 2 (1958), 610–611.

Gunnar Cedersund and Jacob Roll. 2009. Systems biology: Model based evaluation and comparison of potential explanations for given biological data. *FEBS J.* 276, 4 (2009), 903–922.

Gunnar Cedersund, Oscar Samuelsson, Gordon Ball, Jesper Tegnér, and David Gomez-Cabrero. 2016. Optimization in biology parameter estimation and the associated optimization problem. In *Uncertainty in Biology: A Computational Modeling Approach*, Liesbert Geris and David Gomez-Cabrero (Eds.). Cham: Springer, 177–197.

Kathleen E. Cullen, Claudio G. Rey, Daniel Guitton, and Henrietta L. Galiana. 1996. The use of system identification techniques in the analysis of oculomotor burst neuron spike train dynamics. *J. Comput. Neurosci.* 3, 4 (1996), 347–368.

Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An industry standard API for shared-memory programming. *IEEE Comput. Sci. Eng.* 5, 1 (1998), 46–55.

Hiroaki Daido. 1992. Order function and macroscopic mutual entrainment in uniformly coupled limit-cycle oscillators. *Prog. Theor. Phys.* 88, 6 (1992), 1213–1218.

Hooman Darabi and Asad A. Abidi. 2000. Noise in RF-CMOS mixers: A simple physical model. *IEEE J. Solid-State Circuits* 35, 1 (2000), 15–25.

Gustavo Deco, Viktor Jirsa, Anthony R. McIntosh, Olaf Sporns, and Rolf Kötter. 2009. Key role of coupling, delay, and noise in resting brain fluctuations. *Proc. Natl. Acad. Sci. U.S.A.* 106, 25 (2009), 10302–10307.

Denis Demidov, Karsten Ahnert, Karl Rupp, and Peter Gottschling. 2013. Programming CUDA and OpenCL: A case study using modern C++ libraries. *SIAM J. Sci. Comput.* 35, 5 (2013), C453–C472.

Kevin Doherty, Khulood Alyahya, Ozgur E. Akman, and Jonathan E. Fieldsend. 2017. Optimisation and landscape analysis of computational biology models: A case study. In *Proc. GECCO'17 (2017).* 1644–1651.

A. Aldo Faisal, Luc P. J. Selen, and Daniel M. Wolpert. 2008. Noise in the nervous system. *Nat. Rev. Neurosci.* 9, 4 (2008), 292–303.

Jonathan E. Fieldsend. 2015. Elite accumulative sampling strategies for noisy multi-objective optimisation. In *Proc. EMO 2015* (2015). 172–186.

Jun He and Xin Yao. 2001. Drift analysis and average time complexity of evolutionary algorithms. *Artif. Intell.* 127, 1 (2001), 57–85.

Desmond J. Higham. 2001. An algorithmic introduction to numerical simulation of stochastic differential equations. *SIAM Rev.* 43, 3 (2001), 525–546.

Jerald B. Johnson and Kristian S. Omland. 2004. Model selection in ecology and evolution. *Trends Ecol. Evol.* 19, 2 (2004), 101–108.

Mads Kaern, Timothy C. Elston, William J. Blake, and James J. Collins. 2005. Stochasticity in gene expression: From theories to phenotypes. *Nat. Rev. Genet.* 6, 6 (2005), 451–464.

Leeat Keren, David Van Dijk, Shira Weingarten-Gabbay, Dan Davidi, Ghil Jona, Adina Weinberger, Ron Milo, and Eran Segal. 2015. Noise in gene expression is coupled to growth rate. *Genome Res.* 25, 12 (2015), 1893–1902.

Khronos OpenCL Working Group. 2015. *The OpenCL Specification, Version 2.1, Document Revision: 23.* Technical Report. Lee Howes (Ed.).

Peter E. Kloeden and Eckhard Platen. 2011. *Numerical Solution of Stochastic Differential Equations.* Springer: Berlin.

Gabriele Lillacci and Mustafa Khammash. 2010. Parameter estimation and model selection in computational biology. *PLoS Comput. Biol.* 6, 3 (2010), e1000696.

Marc Mantas, José M. De la Asunción and Manuel J. Castro. 2016. An introduction to GPU computing for numerical simulation. In *Numerical Simulation in Physics and Engineering*, Inmaculada Higueras, Teo Roldán, and Juan J. Torrens (Eds.). Springer, 219–251.

H. Nakao. 2016. Phase reduction approach to synchronisation of nonlinear oscillators. *Contemp. Phys.* 57, 2 (2016), 188–214.

Cristóbal A. Navarro, Nancy Hitschfeld-Kahler, and Luis Mateu. 2014. A survey on parallel computing and its applications in data-parallel problems using GPU architectures. *Commun. Comput. Phys.* 15, 2 (2014), 285–329.

Béla Paláncz, Kent Stewart, József Homlok, Christopher G. Pretty, J. Geoffrey Chase, and Balázs Benyó. 2016. Stochastic simulation and parameter estimation of the ICING model. *IFAC-PapersOnLine* 49, 5 (2016), 218–223.

Manolis Papadrakakis, George Stavroulakis, and Alexander Karatarakis. 2011. A new era in scientific computing: Domain decomposition methods in hybrid CPU–GPU architectures. *Comput. Methods Appl. Mech. Engrg.* 200, 13 (2011), 1490–1508.

Eckhard Platen. 1999. An introduction to numerical methods for stochastic differential equations. *Acta Numer.* 8 (1999), 197–246.

Yoshihiro Saito and Taketomo Mitsui. 1996. Stability analysis of numerical schemes for stochastic differential equations. *SIAM J. Numer. Anal.* 33, 6 (1996), 2254–2267.

John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. 2011. Parallel random numbers: As easy as 1, 2, 3. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (2011).* 1–12.

Andries J. Scholten, Luuk F. Tiemeijer, Ronald Van Langevelde, Ramon J. Havens, Adrie T. A. Zegers-van Duijnhoven, and Vincent C. Venezia. 2003. Noise modeling for RF CMOS circuit simulation. *IEEE Trans. Electron. Devices* 50, 3 (2003), 618–632.

Diego. F. Slezak, Cecilia Suárez, Guillermo A. Cecchi, Guillermo Marshall, and Gustavo Stolovitzky. 2010. When the optimal is not the best: Parameter estimation in complex biological models. *PloS One* 5, 10 (2010), e13283.

Steven H. Strogatz. 2000. From Kuramoto to Crawford: Exploring the onset of synchronization in populations of coupled oscillators. *Physica D* 143, 1–4 (2000), 1–20.

Jianyong Sun, Jonathan M. Garibaldi, and Charlie Hodgman. 2012. Parameter estimation using metaheuristics in systems biology: A comprehensive review. *IEEE/ACM Trans. Comput. Biol. Bioinform.* 9, 1 (2012), 185–202.

Carsten Witt. 2008. Population size versus runtime of a simple evolutionary algorithm. *Theor. Comput. Sci.* 403, 1 (2008), 104–120.