cuda-sim
GPU accelerated biochemical network simulation

Yanxiang Zhou and Chris Barnes

September 24, 2010

Contents

1 Overview **3**
1.1 Runtime limit on kernels . . . 3
1.2 Lsoda requires double precision support . . . 3

2 Installation **4**
2.1 Prerequisites . . . 4
2.2 Linux/Mac OS X installation instructions . . . 4

3 Quick tutorial **5**
3.1 Troubleshooting . . . 5

4 Examples **6**
4.1 Example1 : Immigration death . . . 6
4.2 Example2 : p53-Mdm2 oscillations . . . 7

5 Functionality **8**
5.1 SBMLParser.py . . . 8
5.2 Simulator.py . . . 8
5.3 Lsoda.py . . . 8
5.4 EulerMaruyama.py . . . 9
5.5 Gillespie.py . . . 10

6 Providing custom CUDA kernels **12**
6.1 General . . . 12
6.2 ODEs . . . 13
6.3 SDEs . . . 13
6.4 MJPs . . . 14

7 Performance **16**
7.1 Using shared memory . . . 16
7.2 Occupancy . . . 16

1 Overview

Graphics processing units (GPUs) have a highly parallel architecture with a high percentage of silicon dedicated to arithmetic operations and thus are suited to computationally intensive applications that can be parallelized. Biochemical network simulation is one such example as often analysis of models requires hundreds and thousands of simulations in order to understand parameter spaces, calculate probability distributions or perform simulation based inference.

There are a number of manufacturers providing programmable GPUs. NVIDIA has developed a hardware/software architecture called Compute Unified Device Architecture (CUDA) that has received considerable interest from the scientific community because it provides a C API that can be used for general purpose computation. *cuda-sim* uses PyCUDA which is a Python wrapper around CUDA and provides a flexible interface to the GPU.

In CUDA, small codes that run on the GPU are called *kernels*. A single kernel is run by multiple threads arranged into a thread block. Blocks in turn are arranged into a grid. Grids and blocks can be one, two or three dimensional. Since the GPU executes the same code on different data this type of architecture is known as Single Instruction Multiple Data (SIMD). Normal CPU clusters perform Multiple Instruction Multiple Data (MIMD) and this difference in computing paradigm can make porting of existing code to the GPU time consuming with only certain applications being suitable. To complicate things further, the GPU has different memory types, each with different properties such as access speeds and caching.

To run the tools in *cuda-sim*, no in-depth knowledge of CUDA and GPU computing is needed. Despite this there are some things to be aware of.

1.1 Runtime limit on kernels

All operating systems have watchdog timers that monitor the graphics output to detect if the GPU has frozen. As a consequence, the watchdog timer is supposed to terminate kernel calls if they run for more than 5 seconds. For this reason, we recommend running computationally intensive simulations on a separate graphics card that is not connected to a monitor. This issue can be circumvented on Linux by running with the X server disabled. To do this you can issue the command `sudo /sbin/init 3` to kill the X server.

1.2 Lsoda requires double precision support

Most older cards only support floating point arithmetic but ODE integration using LSODA requires CUDA GPUs that support double precision. Only cards that have compute capability 1.3 and greater support double precision. You can see what compute capability your cards support by looking in the NVIDA CUDA programming guide (Appendix A) or on the CUDA website.

2 Installation

More detailed installation instructions can be found in the `README.txt` that accompanies the package.

2.1 Prerequisites

cuda-sim requires the following packages to be first installed

- **CUDA toolkit**
- **numpy**
- **pycuda**
- **libsbml** (only for SBML parsing)

2.2 Linux/Mac OS X installation instructions

Once all required dependencies have been installed successfully and *cuda-sim* downloaded, installing just requires the following

```
$ tar xzf cuda-sim-VERSION.tar.gz
$ cd cuda-sim-VERSION
$ sudo python setup.py install
```

To install to a custom directory, the last line should be replaced with

```
$ sudo python setup.py install --prefix=<dir>
```

which will place the package into the directory

```
<dir>/lib/python2.X/site-packages/
```

where X is the python version.

3 Quick tutorial

The following tutorial describes the essential steps for running *cuda-sim* using a SBML model with custom parameters. For testing the software package with ready-to-run scripts see section Examples.

1. Create a new folder.
2. Create a text file for parameters and initialization values respectively in this folder. Separate parameters and species with white-spaces and each set of parameters and species with a linebreak (for examples see `parameter.dat` and `species.dat` files in `example/ex01_immdeath/` or `example/ex02_p53/`).
3. Copy `runExample.py` (can be found in `example/`) to the previously created folder.
4. Open `runExample.py` in your folder with a text editor.
5. Specify the location of your SBML model file in the line `xmlModel = ""`.
   (E.g. set `xmlModel="/home/myFolder/myModel.xml"`)
6. Specify the location of your parameter and initialization file in the lines
   `parameterFile = ""` and `speciesFile = ""`.
   (E.g. set `parameterFile = "param.dat"` and `speciesFile = "species.dat"`).
7. Decide which type of simulation you want to use by setting the line
   `integrationType = "SDE"` to the simulation type.
   (I.e. `integrationType = "ODE"` for ODE integration or
   `integrationType = "MJP"` for using the Gillespie algorithm).
8. Leave everything else as is, save and close `runExample.py`.
9. Run `runExample.py` (by typing `python runExample.py` in the console).
10. The simulation results can be found in the newly created folder `results/`.

3.1 Troubleshooting

If any problems with the package occur, please provide the SBML model, the edited `runExample.py` script, the parameter and initialization file, the error message and as many specifications of the running system as possible (e.g. Python version, PyCUDA version, OS, GPU specifications) and contact us!

4 Examples

Two examples are provided in the `examples` folder that accompanies the package.

4.1 Example1 : Immigration death

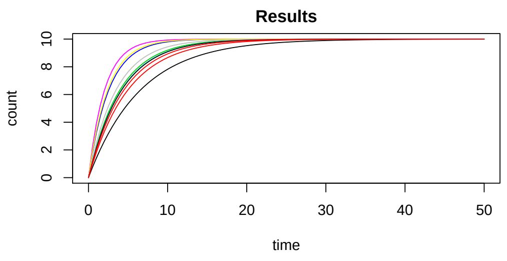
This example models a simple immigration-death process. The model contains one species, has two reactions and is encoded in the SBML file `immigration-death.xml`. To run this example do the following:

```
$ cd examples/ex01_immdeath
$ python runImmdeath.py
```

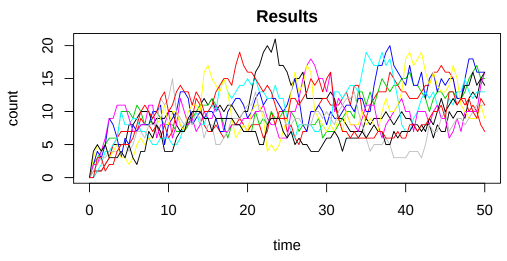
This script reads in the files `param.dat` and `species.dat` which contain the parameters and initial conditions respectively (for the example 1000 sets of parameters and initial conditions were generated); each line represents the value for an individual thread. The simulation results are written into `results/immdeath_result.txt`.

If `R` has been installed the results can be plotted using the `plot.R` script by running R and executing the script in R (Figure 1):

```
$ source('plot.R')
```



(a)



(b)

Figure 1: Simulation results obtained for the immigration-death process for (a) ODEs and (b) MJPs as described above.

4.2 Example2 : p53-Mdm2 oscillations

This example models the p53-Mdm2 negative feedback loop (?). The model contains three species, has two reactions and is encoded in the SBML file `p53model.xml`. To run this example do the following

```
$ cd examples/ex02_p53
$ python runp53.py
```

This script reads in the files `param.dat` and `species.dat` which contain the parameters and initial conditions respectively (for the example 1000 sets of parameters and initial conditions were generated); each line represents the value for an individual thread. The simulation results are written into `results/p53_result.txt`.

If `R` has been installed the results can be plotted using the `plot.R` script by running R and executing the script in R (Figure 3):

```
$ source('plot.R')
```

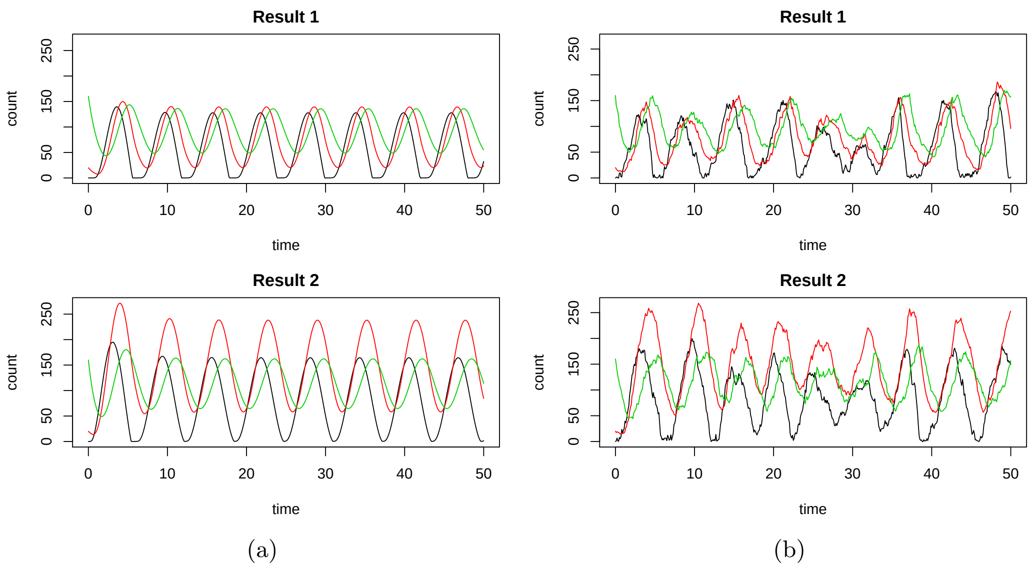


Figure 2: Simulation results obtained for the p53-Mdm2 negative feedback loop for (a) ODEs and (b) SDEs as described above.

5 Functionality

5.1 SBMLParser.py

The SBML parser provides the ability to generate CUDA kernels for ODE, SDE and MJP simulations using the pre-implemented algorithms. The SBML parser is derived from the SBML parser for the ABC-SysBio package.

Call SBMLParser

To access the SBMLParser the following function needs to be called (multiple models can be parsed at once):

```
importSBMLCUDA([xmlModel],[integrationType],[modelName],outpath=temp)
```

- **[xmlModel]** Array of strings with the location of the SBML files.
- **[integrationType]** Array of strings with the simulation types to be used (i.e. “ODE”, “SDE” or “MJP”).
- **[modelName]** Array of strings with the names of the models.
- **outpath** String specifying the folder where the CUDA kernels should be saved.

5.2 Simulator.py

The Simulator class is the superclass for the Lsoda, EulerMaruyama and Gillespie classes. Here, all functions that the user should be able to access are defined. Furthermore, all parameters regulating GPU calls (like blocksize and gridsize) are defined in `Simulator.py`.

5.3 Lsoda.py

The Lsoda class uses the LSODA algorithm to solve ODEs.

Instantiating Lsoda

To run the ODE solver, firstly, the CUDA code has to be compiled. This is done by instantiating:

```
Lsoda(timepoints, cudaCode)
```

- **timepoints** Array of floats specifying the time points at which the integration results should be printed out. Time points do not need to be equidistant.
- **dt** Float specifying time step distances (no relevance for ODE integration)
- **cudaCode** String specifying CUDA kernel.

Run ODE integration

To run the ODE integration the following function has to be called:

```
run(parameters, initValues, timing=boolean)
```

- **parameters** Array of arrays of floats specifying parameters for every thread and all different parameters in the model. The length of the parameters array defines the total number of threads and should be equal to the length of the initValues array.
- **initValues** Array of arrays of floats specifying initialization values for every thread and all different species in the model. The length of the initialization values array should be equal to the length of the parameters array.
- **timing** Boolean specifying if the runtime should be reported.

Returned result

The `run` function returns the result as four-dimensional array containing float values in the follwing form (this is the same as for EulerMaruyama and Gillespie):

```
[#threads][beta][#timepoints][speciesNumber]
```

- **#threads** Index for different threads.
- **beta** Index for different repeats with the same set of parameters. (Can only be 0 since no repeats for ODE integrations will be carried out using the same set of parameters).
- **#timepoints** Index for each time point that was specified when calling `run`.
- **speciesNumber** Index for different species.

5.4 EulerMaruyama.py

The EulerMaruyama class uses the Euler-Maruyama algorithm to simulate SDEs.

Instantiating EulerMaruyama

To run SDE simulations, firstly, the CUDA code has to be compiled. This is done by instantiating:

```
EulerMaruyama.EulerMaruyama(timepoints, cudaCode, dt, beta)
```

- **timepoints** Array of floats specifying the time points at which the integration results should be printed out. Timepoints do not need to be equidistant.
- **cudaCode** String specifying CUDA kernel.
- **dt** Float specifying time step distances.
- **beta** Integer specifying the number of repeated simulations with the same set of parameters. Should be used to explore sensitivity of the results to noise.

Run SDE simulation

To run the simulation, following function has to be called (this is the same as for Gillespie):

```
run(parameters, initValues, timing=boolean, seed=seedValue)
```

- **parameters** Array of arrays of floats specifying parameters for every thread and all different parameters in the model. The length of the parameters array defines the total number of threads and should be equal to the length of the initValues array.
- **initValues** Array of arrays of floats specifying initialization values for every thread and all different species in the model. The length of the initialization values array should be equal to the length of the parameters array.
- **timing** Boolean specifying if the runtime should be reported.
- **seed** Integer between 0 and 4294967295 to seed random number generators. For consecutive runs, the optional seed can be left out since it will be updated internally.

Returned result

The `run` function returns the result as four-dimensional array containing float values in the follwing form (this is the same as for Lsoda and Gillespie):

```
[#threads][beta][#timepoints][speciesNumber]
```

- **#threads** Index for different threads.
- **beta** Index for different repeats with the same set of parameters.
- **#timepoints** Index for each time point that was specified when calling `run`.
- **speciesNumber** Index for different species.

5.5 Gillespie.py

The Gillespie class uses the Gillespie algorithm to simulate MJPs.

Instantiating Gillespie

To run MJP simulations, firstly, the CUDA code has to be compiled. This is done by instantiating:

```
Gillespie(timepoints, cudaCode, beta)
```

- **timepoints** Array of floats specifying the time points at which the integration results should be printed out. Time points do not need to be equidistant.
- **cudaCode** String specifying CUDA kernel.
- **beta** Integer specifying the number of repeated simulations with the same set of parameters. Should be used to explore sensitivity of the results to noise.

Run MJP simulation

To run the simulation following function has to be called (this is the same as for Euler-Maruyama):

```
run(parameters, initValues, timing=boolean, seed=seedValue)
```

- **parameters** Array of arrays of floats specifying parameters for every thread and all different parameters in the model. The length of the parameters array defines the total number of threads and should be equal to the length of the initValues array.
- **initValues** Array of arrays of floats specifying initialization values for every thread and all different species in the model. The length of the initialization values array should be equal to the length of the parameters array.
- **timing** Boolean specifying if the runtime should be reported.
- **seed** Integer between 0 and 4294967295 to seed random number generators. For consecutive runs the optional seed can be left out since it will be updated internally.

Returned result

The `run` function returns the result as four-dimensional array containing float values in the follwing form (this is the same as for Lsoda and EulerMaruyama):

```
[#threads][beta][#timepoints][speciesNumber]
```

- **#threads** Index for different threads.
- **beta** Index for different repeats with the same set of parameters.
- **#timepoints** Index for each time point that was specified when calling `run`.
- **speciesNumber** Index for different species.

6 Providing custom CUDA kernels

Instead of using the SBML parser, it is also possible to directly provide CUDA kernels. In the following sections, the most important aspects of the CUDA kernels for the different algorithms are described.

6.1 General

Header

Firstly, all CUDA kernels need to specify the number of species, parameters and reactions. This is done using `define` statements (please note that there is one additional parameter for the compartment):

```
#define NSPECIES 3
#define NPARAM 9
#define NREACT 6
```

Parameters

For all three algorithms, parameters are stored in texture memory. The parameters can be accessed by calling the following function:

```
tex2D(param_tex,index,tid)
```

The argument `index` has to be replaced by the index of the parameter to access. The index 0 is always reserved for the compartment parameter. Since texture memory is read-only memory, parameters cannot be modified directly. Rather a wrapping function has to be defined, that, depended on the time or other parameters, returns a modified parameter. E.g. if you want the parameter with index 3 to be doubled between the time points 10 and 15 you have to define the following function:

```
__device__ float function(float a1, float t){
    if(t > 10 && t < 15)
        return 2*a1;
    else
        return a1;
}
```

and instead of calling the reference to the texture directly, the call has to be wrapped with the function defined above:

```
function(tex2D(param_tex,3,tid), t)
```

Species

Species are stored in registers that can also be written. They can be directly accessed and changed by referencing with the index of the species:

```
y[index]
```

6.2 ODEs

Overview

For solving ODEs, either the difference quotient for each species or the full Jacobian should be specified inside structures. In our examples and by using the provided SBML parser, difference quotients are used. For details how to specify the Jacobian please refer to the comments in the header of `cuLsoda_all.cu`.

Specify difference quotients

The struct frame should look as follows:

```
struct myFex{
    __device__ void operator()(int *neq, double *t, double *y, double *ydot){
    }
};
```

Inside the `operator` function, the difference quotient for the species should be defined by specifying `ydot` for each species, e.g.:

```
ydot[0] = tex2D(param_tex,1,tid) * y[0];
```

6.3 SDEs

Overview

For the SDEs, two versions of the code should be written. One for using texture memory and one for using shared memory for storing parameters. The only difference between these versions is how parameters are accessed. Both version should be put into the same file. The version for using texture memory should begin with the following comment:

```
//Code for texture memory
```

and the version for using shared memory should begin with:

```
//Code for shared memory
```

To access parameters via their index in shared memory, use the following call:

```
parameter[index]
```

In contrast to parameters stored in texture memory (as described above), parameters in shared memory can also be directly changed. I.e. the following statement is possible:

```
parameter[0] = 1.5;
```

Step-function

The function that is called in every time step should have this form for the version for using texture memory:

```
__device__ void step(float *y, unsigned *rngRegs, int tid){
}
```

and for using shared memory, additionally, a pointer to the shared memory, where the parameters are stored is needed:

```
__device__ void step(float *parameter, float *y, unsigned *rngRegs){
}
```

The constant of each time step is stored in the global constant:

```
DT
```

Normally distributed random number

To generate normally distributed random numbers with a mean of 0 and a variance of `DT`, call the following function:

```
randNormal(rngRegs,WarpStandard_shmem,sqrt(DT));
```

It is **very important** that all threads in a warp call this function at the **same time**. No calls of this function should occur after any intra-warp divergence. This can be ensured if no conditional statements like `ifs` are place inside the `step()` function before a call to `randNormal()` is made.

6.4 MJPs

Overview

For using MJPs, the stochiometry matrix and a hazards function have to be specified.

Stochiometry matrix

The stochiometry specifies the changes of species if a certain reaction occurs. The "matrix" has one column for each species and one row for each reaction. E.g. for a model with two species (X and Y) and the three following reactions:

- $\emptyset \to X$
- $X \to 2Y$
- $Y \to \emptyset$

the stochiometry matrix should be defined the following way (with X being the first and Y being the second species):

```
__constant__ int smatrix[]={
    1.0,  0.0,
   -1.0,  2,0,
    0.0, -1.0
};
```

Hazards function

The hazards function specifies the probability of each reaction occurring. For the example above, the hazards function would be defined as follows:

```
__device__ void hazards(int *y, float *h, float t, int tid){
    h[0] = tex2D(param_tex,1,tid);
    h[1] = tex2D(param_tex,2,tid)*y[0];
    h[2] = tex2D(param_tex,3,tid)*y[1];
}
```

Here, each reaction is determined by another parameter. (Note that the indices begin with 1, since the 0-th parameter represents the compartment).

7 Performance

cuda-sim was developed to increase the performance of simulating biochemical networks. Even though in comparison to simulations on single CPUs substantial performance gains are attained, to get the best performance out of GPUs, dependent on the model, further fine tuning may be required. In the following sections we discuss some of these issues. For a general introduction to the concepts of GPU computing and CUDA please refer to the NVIDIA CUDA Programming Guide and Dr. Dobb's introduction on CUDA – "CUDA, Supercomputing for the Masses".

7.1 Using shared memory

By using PyCUDA we can employ *runtime code generation. We make use of this for SDE simulations and* cuda-sim automatically determines at runtime if the parameters fit in shared memory. By using the faster shared memory instead of texture memory we could decrease the runtime for the p53-Mdm2 model by 2.4%. The decrease in runtime will strongly depend on the memory access intensity.

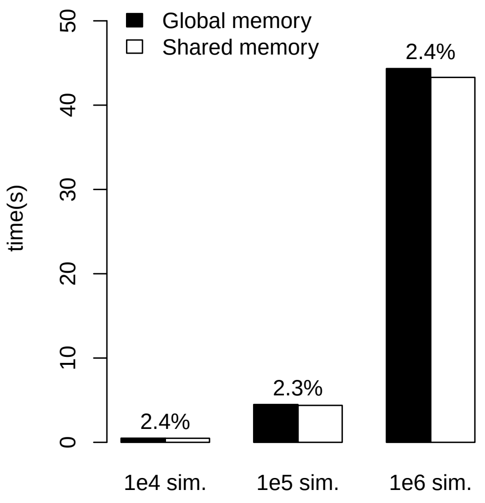


Figure 3: Timing comparisons between using texture memory and shared memory. Timing for different numbers of simulations of length 100 hours using the p53-Mdm2 model storing the parameters in texture memory and shared memory. The decrease in runtime for using shared memory is indicated above the bars. All timing results are the averages from three runs.

7.2 Occupancy

A major determinant of performance is how well latency is hidden by multiple warps per multiprocessor. In this case, warps that are not ready for execution can be "parked" while other warps are executed. In general, it is therefore desirable to have a high number of warps per multiprocessor (occupancy). However, depending on the exact ratio of float point operations to memory access, there may be a value for occupancy above which no performance gain can be expected, because the multiprocessors are already working to their full capacity.

Since the maximum number of blocks per multiprocessor for all compute capabilities is restricted to 8, blocksizes of 96, 128 or 192 threads per block (or multiples thereof) for compute compatibility 1.0 & 1.1, 1.2 & 1.3 and 2.0 respectively are required to maximize occupancy. But this only holds true if the number of registers or shared memory used per kernel are not limiting the number of warps per multiprocessor. In our case, the number of registers will be the limiting factor for the maximum number of warps per multiprocessor. Therefore the blocksizes for achieving a high occupancy will not be those values mentioned above and vary between different models.

Another issue with choosing the blocksize is the number of unused threads depending on the total number of threads the user wants to run. In the worst case, a complete block (minus one thread) might be needed to be initialized and run (for example when a blocksize of 64 threads is chosen, and 65 threads should be run). The smaller the blocksize is, the smaller the wasted number of threads will be. This is especially important for running smaller numbers of threads.

For the reasons mentioned above, we set the default blocksize to 64 (or reduce it to 32, if the number of registers used per kernel only allows 32 threads per block). We think that 64 is a good compromise between having a high occupancy and a small overhead. For specific applications (especially when you are running very time-consuming simulations) you might want to test if other blocksizes will give you an improved performance.