CMSE/CSE 822: Parallel Computing
Fall 2019, Homework 7
Due 11:59 pm, Friday, Dec. 13th, 2019

Important note: *Please use a word processing software (e.g., MS Word, Mac Pages, Latex, etc.) to type your homework. Follow the submission instructions at the end to turn in an electronic copy of your work.*

Late day policy: *Even if you have more late days, the maximum number of late days that can be used for this assignment is 2 due to the deadline for submitting the final grades. Hence, the absolute deadline for this assignment is 11:59 pm, Sunday, Dec. 15th.*

GPU-based Cardiac Electrophysiology Simulation[1] [100 points]

Simulations play an important role in science, medicine, and engineering. For example, a cardiac electrophysiology simulator can be used for clinical diagnostic and therapeutic purposes. Cell simulators entail solving a coupled set of equations: A system of Ordinary Differential Equations (ODEs) together with Partial Differential Equations (PDEs). In this part, you'll implement the Aliev-Panfilov heart electrophysiology simulator, which is a 5-point stencil method and requires the solution of a PDE and two related ODEs. We will be focusing not on the underlying numerics, but on the efficiency aspects of the simulation.

The simulator models the propagation of electrical signals in the heart, and it incorporates a cell model describing the kinetics of the membrane of a single cell. The PDE couples multiple cells into a system. There can be different cell models, with varying degrees of complexity. We will use a model known as the Aliev-Panfilov model, which maintains 2 state variables and solves a single PDE. This simple model can account for complex behavior such as how spiral waves break up and form elaborate patterns. Spiral waves can lead to life threatening situations such as ventricular brillation (a medical condition when the heart muscle twitches randomly rather than contracting in a coordinated fashion).

The simulator models electrical signals in an idealized system in which voltages vary at discrete points in time, called timesteps, on discrete positions of a mesh of points. For simplicity, we'll use a uniformly spaced mesh (irregular meshes are also possible, but are more difficult to parallelize). At each time step, the simulator updates the voltage according to nearest neighboring positions in space and time. This is done first in space, and next in time. Nearest neighbors in space are defined on the north, south, east and west (i.e., a 5-point stencil).

In general, the finer the mesh (and hence the more numerous the points), the more accurate is the solution, but at an increased computational cost. Likewise, the smaller the timestep, the more accurate is the solution, but at a higher computational cost. To simplify your performance analysis, you can run your simulations for a limited number of iterations on relatively small grids – actual simulations would require very large number of timesteps on large grids, and hence take several days.

Simulator

The simulator keeps track of two state variables that characterize the electrophysiology that is being simulated. Both state variables are represented as 2D arrays. The first variable, called the excitation, is stored in the E[ ][ ] matrix. The second variable, called the recovery variable, is stored in the R[ ][ ] matrix. Lastly, we store Eprev, the voltage at the previous timestep, which is needed to advance the voltage over time.

[1] Credit: Simula Research Lab (Norway), Scott Baden (LBNL), Didem Unat (Koc Univ.)

Since we are using the method of finite differences to solve the problem, the variables E and R are discretized by considering the values only at a regularly spaced set of discrete points. The formula for solving the PDE, where E and Eprev refer to the voltage at current and previous timestep, respectively, and α is a constant defined in the code, is as follows:

```
E[i,j] = Eprev[i,j] + alpha*(Eprev[i+1,j] + Eprev[i-1,j] +
                             Eprev[i,j+1] + Eprev[i,j-1] - 4*Eprev[i,j])
```

The formula for solving the ODE is shown below. Here *kk, a, b, epsilon*, *M1* and *M2* are again constants defined in the simulator and *dt* is the time step size.

```
E[i,j] = E[i,j] - dt*( kk * E[i,j] * (E[i,j] - a) * (E[i,j] - 1) +
         E[i,j] * R[i,j]);

R[i,j] = R[i,j] + dt*(epsilon + M1*R[i,j] / (E[i,j] + M2)) *
                      (-R[i,j] - kk * E[i,j] * (E[i,j]-b-1));
```

Serial Code

You are given a working serial simulator that uses the Aliev-Panfilov cell model described above. Command line options for the cardiac electrophysiology simulator are as follows:

```
-t <float> Duration of simulation
-n <int> Number of mesh points in the x and y dimensions
-p <int> Plot the solution as the simulator runs, at regular intervals
-x <int> x-axis of the processor geometry (valid only for MPI implementation)
-y <int> y-axis of the processor geometry (valid only for MPI implementation)
-k Disable MPI communication
-o <int> Number of OpenMP threads per process
```

For convenience, the simulator includes a plotting capability (requires using "-X" argument while accessing HPCC through ssh to forward the application display to your local machine and loading gnuplot afterwards) which you can use to debug your code, and also to observe the simulation dynamics. The plot frequency can be adjusted from the command line. **However, your timings results should be measured with the plotting option disabled.**

```
Example command line

     module load CUDA/10.0.130 gnuplot
     make

     ./cardiacsim -n 400 -t 1000 -p 100
```

The example will execute on a 400 x 400 box, and run to 1000 units of simulated time, plotting the evolving solution every 100 units of simulated time. For this assignment, you do not have to worry about options *-x*, *-y*, *-k*, or *-o*.

You'll notice that the solution arrays are allocated to be 2 larger than the domain size (n+2). The boundaries are padded with a cell on each side in order to properly handle ghost cell updates using mirroring technique. You can see Lecture 16 for details about handling ghost cells.

Assignment
You will parallelize the cardiac electrophysiology simulator using a single GPU. Starting with the serial implementation provided, we ask you to implement 4 different versions using CUDA:

Version 1: Implement a naive GPU parallel simulator that creates a separate kernel for each for-loop in the *simulate* function. These kernels will all make references to global memory. Make sure that your naive version works correctly before you implement the other three options. In particular, check if all data allocations on the GPU, data transfers and synchronizations are implemented correctly.

Version 2: Fuse all kernels into a single kernel. Note that you can fuse the ODE and PDE loops into a single loop, thus into a single kernel.

Version 3: Use temporary variables to eliminate global memory references for R and E arrays in the ODEs.

Version 4: Optimize your CUDA implementation by using shared memory (on-chip memory) on the GPU by bringing a 2D block into shared memory and sharing it with multiple threads in the same thread block.

You should implement these optimizations on top of each other (e.g. Version 3 should be implemented on top of Version 2). Details of how to implement these optimizations will be discussed in class on Tuesday, December 5th. **Note that these optimizations do not guarantee performance improvement.** Implement them and observe how they affect the performance.

Ghost Cells
Since all CUDA threads share global memory, there is no need to exchange ghost cells between thread blocks. However, the physical boundaries of the domain need to be updated every iteration using mirror boundary updates. You can do this in a series of separate kernel launches on the GPU, which will be a bit costly, or embed mirror boundary updates into a single kernel launch.

Data transfers from CPU to GPU
Note that it is enough to transfer the simulation data from CPU to GPU only once before the simulation iterations start (before the while(t<T) loop in the main()). During the course of the simulation, E, R, and E_prev will be updated and used on the GPU only. Optionally, when you would like to generate a plot, you will need to copy the E matrix from GPU to CPU.

Reporting Performance

- Use t=100 for your performance studies, and experiment with different grid sizes such as n=200, 1000 and 5000. Measure performance without having the plotter on.

- Recall that data transfer time affects the performance. When you measure the execution time and Gflop/s rates, do not include the data transfer time.

- Tune the block size for your implementation and draw a performance chart for various block sizes.

- Compare the performance of your best implementation with the CPU version (serial).

- Document your findings in your write-up.

Programming Environment

You will use the Nvidia K80 GPUs on HPCC's Intel16 cluster.

https://wiki.hpcc.msu.edu/display/hpccdocs/GPU+Computing

Since you will run your simulations on small problems for small number of iterations, your execution times will be short. Therefore, you can use the dev-intel16-k80 nodes for both development and performance testing purposes. But note that everyone in the class will be using the two GPUs located on the same node. So, make sure to give yourself enough time to develop and test your program before the project deadline.

Grading

Your grade will depend on 2 factors: correctness of your implementations and the depth and clarity of your report. Document your work well, discuss your findings, and offer insights into your results and plots.

Implementation (75 points): Version 1, 2 and 4: 20 pts each, Version 3 is 15 pts.

Report (25 points): Implementation description and performance analysis.

Instructions

- **Cloning git repo.** You can clone the skeleton codes through the git repo. Please refer to *"Homework Instructions"* under the *"Reference Material"* section on D2L.

- **Discussions.** For your questions about the homework, please use the Slack group for the class so that answers by the TA, myself (or one of your classmates) can be seen by others.

- **Submission.** Your submission will include:
  - A pdf file named "HW7*_yourMSUNetID*.pdf", which contains your answers to the non-implementation questions, and report and interpretation of performance results.
  - Source file used to generate the reported performance results. Make sure to use the exact files name listed below:
    - cardiacsim.cu

    It is essential that you do not change the directory structure or file names.

  To submit your work, please follow the directions given in the "Homework Instructions" under the "Reference Material" section on D2L. Make sure to strictly follow these instructions; otherwise you may not receive proper credit.

- **Compilation and execution.** You are provided a Makefile for compiling your code. Simply type "make" in the directory where the Makefile is located at.