CSE 491-Section 2
Parallel Processing

Homework 4
Due in class, December 9, 2015

Important Notes:

- ***This homework can be done individually or in pairs. We require only one submission from each pair, but make sure to clearly identify the team members in your submitted codes and reports.***
- ***This assignment contains a bonus problem which is worth 40 points. The total points you can receive from all four assignments is 400. To make up for your grades from previous assignments, this bonus problem is a great opportunity!***
- *For your questions about the homework, please use the D2L discussion forum for the class so that answers by the TA, myself (or one of your classmates) can be seen by others.*
- *The performance data must be collected on HPCC's intel14 cluster. You can use the dev-intel14-k20 node for GPU programming.*
- ***Late submission policy:** You can submit one late homework during the semester. However, there will be a 25% penalty for each late day.*

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 a 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 ODEs at each timestep. We will not be focusing on the underlying numerical issues, but if you are interested in learning more about them, please refer to references at the end.

The simulator models the propagation of electrical signals in the heart, and it incorporates a cell model describing the kinetics of a 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, that maintains 2 state variables, and solves one 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. In our case 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 the solution, but for an increased computational cost. In a similar fashion, the smaller the timestep, the more accurate the solution, but at a higher computational cost. To simplify the performance studies, we will run our simulations for a given number of iterations rather than a given amount of simulated time because actual simulation takes too long (several days) on large grids.

---

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

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

Since we are using the method of finite differences to solve the problem, we discretize the variables E and R 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 the constant α is defined in the simulator 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. The constants *kk, a, b, epsilon*, *M1* and *M2* are 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. For convenience, the simulator includes a plotting capability (using gnuplot) which you can use to debug your code, and also to observe the simulation dynamics. The plot frequency can be adjusted from command line. **However, your timings results will be taken when the plotting is disabled.** The simulator has various options:

```
./cardiacsim

With the arguments
 -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 the processor geometry (Used only for your MPI
implementation)
  -y <int> y-axis of the the processor geometry (Used only for your MPI
implementation)
 -k Disable MPI communication
 -o <int> Number of OpenMP threads per process
 Example command line
   ./cardiacsim -n 400 -t 1000 -p 100
```

The example will simulate 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 Monday, November 30th. **Note that these optimizations do not guarantee performance improvement.** Implement them and observe how they affect the performance. For full credit, you have to implement all 4 versions.

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 N=1024 and t=100 for your performance studies. 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 K20 Kepler GPUs on HPCC's Intel14 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-intel14-k20 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.

Turn-in Instructions

- You will turn in your source code in a zip file by emailing it to the TA. Submit all four versions of your code, label them properly. Include all the necessary files to compile your code. Be sure to delete all object and executable files before creating a zip file.

- The first paragraph of your report should clearly state which versions work properly or which version is the best performant one. We will only test the final implementation (version 4) if the report does't guide us.

- Document your work in a well-written report which discusses your findings. Offer insight into your results and plots.

Grading

Your grade will depend on 2 factors: correctness of your implementations and the depth and clarity of your explanations for observed performance in your report.

Implementation (80 points): Version 1 and 4 (30 pts each), Version 2 and 3 (10 pts each)

Report (20 points): implementation description and any tuning/optimization you performed (10 pts), performance study (10 pts).

BONUS. Parallel Weight Balancing with OpenMP Tasks (40 points)

Suppose you have a set of real number of weights W1,W2,……,Wn. You are asked to divide the weights into two sets such that the difference of the sum of weights in these two sets is minimum. Assume, {1.2, 2.3, 4.5, 10.2} is the set of given weights. This set can be divided into two sets{1.2, 2.3, 4.5} and {10.2} so that their difference of sum in two sets becomes {10.2 - (1.2+2.3+4.5)} = 3.2 which is the minimum possible.

This is an NP-complete problem, so the only way to find the exact solution is to enumerate all possible combinations and choose the one which gives the minimum difference between the two sets. Therefore you need to create all subsets of the given set of weights. The pseudocode for generating subsets using backtracking method is given below and the skeleton code provided to you:

```
S = input set
index = index in set S from where the current subset should start

GenerateSubset(index, subset, S) {
      if (index == S.size) { //then this subset is done
            update if it is a better solution than what we have so far
            return
      }

      subset = subset U S(index) //add the index element to the subset
      GenerateSubset(index+1, subset, S)  // generate subsets with the
                                          // index element in them
      remove the added element from the subset
      GenerateSubset(index+1,subset,S)    // generate subsets without
                                          // the index element
```

Keep in mind that there will be 2^n number of subsets for a set of n elements. So you should refrain from saving all subsets, as this will require huge memory space. Instead you can keep a current best solution, and update your solution when you identify a partitioning which is better the current one. For initialization, you can start by setting the best solution to be empty and the global minimum to an arbitrarily large value.

How can you update your current solution? Assume that the sum of all given weights is SUM. Note that the best solution will have the minimum difference between the sum of its weights and SUM/2. So for each generated subset, you can simply compute the difference between your generated subset's sum and (SUM/2) and update your solution if it is less than your current best.

Using the skeleton code provided, first implement a sequential version of the program. Then implement an OpenMP parallel version using tasks — which are ideally suited for parallelizing recursive programs. Each recursive call to the GenerateSubset can be denoted as a new OpenMP task. See Lecture 9 slides, and in particular the Fibonacci numbers example there. Note that the

Reporting Performance

- Experiment with different number of weights such as 15, 20, 25 and 30.
- For your OpenMP parallel version, experiment with different number of threads, i.e. 2, 4, 8, 12, 16, 20.

- The skeleton code provided expects the #weights and #threads to be provided as command line arguments. For the serial version, you still need to provide the number of threads, but you will not be using it.
- Plot the speedups for your parallel code with respect to the serial code. Explain your observations.

Programming Environment

Use the Intel14 cluster for development purposes. Once you make sure that your program is running correctly and you are satisfied with its performance, run your jobs on the compute nodes to get reliable performance numbers. You can request exclusive access to a node and run all your tests in a single job script as follows:

```
#!/bin/bash -login
#PBS -l walltime=00:15:00,nodes=1:ppn=20,feature=intel14
#PBS -j oe
#PBS -N subset

# change to the working directory where your code is located
#cd ~/cse491_fs15_s2/hw4
cd $PBS_O_WORKDIR

# call your executable with different no. of threads
./subset_seq  15 1
./subset_tasks  15 2
./subset_tasks  15 4
# etc..
```

Turning in Your Homework

- You will turn in your source codes in a zip file by emailing it to the TA. Submit both the sequential version of your code and the OpenMP parallel version with tasks, label them properly. Include all necessary files to compile your code, but make sure to delete all object and executable files before creating a zip file.

- Document your work in a well-written report which discusses your findings. Offer insight into your results and plots. You will turn in a hardcopy report.

Grading

Your grade will depend on 2 factors: correctness of your implementations and the depth and clarity of your explanations for observed performance in your report.

Implementation (30 points): serial version (10 pts), OpenMP version (20 pts)

Report (10 points): implementation description and any tuning/optimization you performed, performance study.