COMP 429/529- Parallel Programming: Assignment 3

Due: 23rd May, 2019

Spring 2019

Cardiac Electrophysiology Simulation

In this project you'll implement the Aliev-Panfilov heart electrophysiology simulator using CUDA. Since implementing/debugging parallel codes under CUDA can be very time consuming, give yourself sufficient time to complete the assignment. In addition, the project requires you to conduct various performance studies. Please also allow yourself enough time for performance measurements. The performance results will be collected on the KUACC cluster.

Serial Code

You are provided with a working serial simulator that uses the Aliev-Panfilov cell model. 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. Your timings results will be taken **when the plotting is disabled.**
The code has various options as follows.

```
./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
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.

You'll notice that the solution arrays are allocated 2 larger than the domain size (n + 2). We pad the boundaries with a cell on each side, in order to properly handle ghost cell updates.
You will parallelize the cardiac simulation using CUDA. Starting with the serial implementation provided, the assignment requires 4 versions of the same code:

- Version 1: Parallelize the simulator using single GPU. First implement a naive version that makes all the references to global (device) memory. Make sure that your naive version works correctly before you implement the other three versios. Check if all the data allocations on the GPU, data transfers and synchronisations are implemented correctly.
- Version 2: Fuse all the kernels into one kernel. 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: Then optimise 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).
- Note that these optimisations do not guarantee performance improvement. Implement them and observe how they affect the performance.

Ghost Cells

Since all the 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.

Reporting Performance

- Use N=1024 and t=500 for your studies.
- Conduct a performance study without the plotter is on.
- Observe that data transfer time affects the performance. When you measure the execution time and Gflop/s rates, do not include the data transfer time.
- Run the stream benchmark variant discussed in Lecture 20.

  The bandwidth rate measured by the benchmark is the highest bandwidth rate that can be achieved on the device. Then compare it against the bandwidth rate that your implementation achieves. As you optimise the code you will get closer to this peak bandwidth but you will never achieve the same bandwidth, can only achieve a fraction of it.
- 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).
- Compare the performance of your best implementation with the MPI version you implemented in Assignment 2.
- Document these in a report.

Environment

We will be using the KUACC cluster for the assignment.

- You need to add gres parameter to be able to use GPUs. You can fine the example jobscript at /kuacc/jobscripts/mumax3/mumax3_submit.sh
- or on the web https://docs.computecanada.ca/wiki/Using_GPUs_with_Slurm
- For interactive GPU usage, you can use the following commands,

```
srun  -A users -p short -n1 --gres=gpu:1 --qos=users --pty $SHELL

srun  -A users -p short -n1 --gres=gpu:gtx_1080ti:1 --qos=users --pty $SHELL
```

- For batch jobs, you need to add the following parameter into your scripts

```
#SBATCH --gres=gpu:tesla_k20m:1
or

#SBATCH --gres=gpu:1

#SBATCH constraint=tesla_k20m
```

- In KUACC, there are 32 GPUs with five K20, four K40, thirteen K80 and ten 1080ti, and Tesla K40s are the busiest ones.

References

https://www.simula.no/publications/stability-two-time-integrators-aliev-panfilov-system.
Prof. Scott Baden http://cseweb.ucsd.edu/ baden/

GOOD LUCK.