Extreme Computing: Homework 2

Michael S. Emanuel, Jonathan Guillotte-Blouin, Yue Sun

April 15, 2019

1 Problem 1: Channel Flow with a central narrowing

Simulating channel flow in 2D with a narrowing is the target.

The type of flow in absence of the narrowing is the well-known Hagen-Poiseuille flow (and by the way, Poiseuille was a physicist and physiologist who studied blood flows in a systematic way).

Let's write the corresponding grid for a straight channel aligned along, say, the x axis.

When constructing the grid and using it in the LBM context you should apply what you have learned about grid management.

Let's recall that in absence of open boundaries (inlet or outlet), each fluid node should always be surrounded by other fluid or wall nodes to preserve mass and momentum locally (no leakage rule).

The solution should employ two different options:

1. A periodic channel (aligned with the x direction): start from a quiescent fluid and apply a body acceleration uniformly through the fluid until you match the Reynolds number at steady state.

2. A non-periodic channel with inlet and outlet faces and by applying a uniform velocity at inlet and outlet with plug flow profiles.

Use no-slip boundary conditions at the walls and a Reynolds number of 10^{-2} computed from fluid velocity imposed at $x = 0$ (the inlet region for non-period channel) and characterstic length being H.

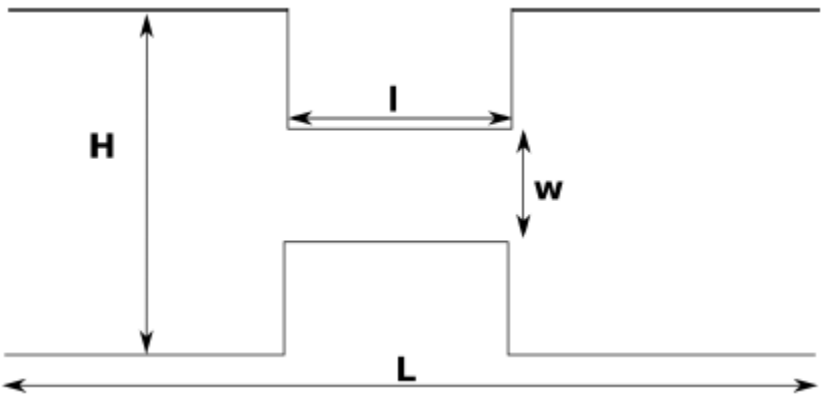

Figure 1: The geometry for the channel with central narrowing.

For a fixed set of values for the geometry as in Fig. 1, (suggested values $L = 200$, $H = 60$, $l = 50$, $w = 30$), calculate the following quantities:

1. The pressure field
2. The velocity field
3. The volume flow rate
4. The average and maximum velocities in the channel

Repeat the simulation by varying the width of the narrowing w.

Our solution to this problem can be found in the folder `hw2/lbm_7` in our team's repository for this course. Here is a brief overview of the components in our solution. The files `lattice.hh` declares and implments the class `lattice` (that is, it's a header-only implemenation). One instance of this class represents a single point on the lattice.

The driver programs `lbm_pbc` and

Plot: Show how the flow rate changes by varying w in the range $0 < w < 50$

Compare the pressure at the narrowing with the simple Bernoulli estimate for inviscid flows, stating that

$$\frac{1}{2}\rho u_o^2 + p_o = \frac{1}{2}\rho u_i^2 + p_i$$

where u_o and p_o are cross-sectional averages of velocity and pressure at $x = 0$ for a periodic system or at the inlet otherwise, and u_i and p_i are the same quantities at the center of the narrowing.

Plot: Plot the obtained pressure vs the Bernoulli estimate by varying the LBM kinematic viscosity in the range $0.05 : 0.1667$.

2 Problem 2: Basic CUDA and GPUs

This problem was submitted by Dan Willen `daniel.p.willen@gmail.com`

Consider the equation

$$\frac{\partial u}{\partial t} = D\nabla^2 u, \tag{1}$$

on the domain $0 \leq x, y \leq \pi$, with $u = u(x, y, t)$, D the diffusive constant, and ∇^2 the Laplace operator. The boundary conditions are of the homogeneous Dirichlet type:

$$u(x = 0, y) = u(x = \pi, y) = u(x, y = 0) = u(x, y = \pi) = 0, \tag{2}$$

and the initial condition is

$$u(x, y, t = 0) = \sin(x)\sin(y). \tag{3}$$

The solution to this equation is

$$u(x, y, t) = \sin(x)\sin(y)\exp(-2Dt) \tag{4}$$

Using a second-order central difference in space and first order forward difference in time, the discretization of (1) is

$$u_{i,j}^{(n+1)} = u_{i,j}^{(n)} + \frac{D\Delta t}{\Delta x^2}\left[u_{i+1,j}^{(n)} + u_{i-1,j}^{(n)} + u_{i,j+1}^{(n)} + u_{i,j-1}^{(n)} - 4u_{i,j}^{(n)}\right], \tag{5}$$

where $u_{i,j}^{(n)}$ is the value of u at the n^{th} time step at grid point (i, j), $\Delta t = \Delta x^2/4D$ is the time step size, and Δx is the grid spacing in the x and y directions.

1. Using Cuda, solve the discretized equations up to a time $t = \pi^2/D$ using the Jacobi method. A skeleton code is provided to assist you, with comments in the locations you should make changes.

   Step I – Declare, allocate, and initialize memory for the field variable `u` on the CPU. You should allocate enough memory for the grid, which has size $nx \times ny$ and initialize `u` to the initial condition. Make sure you free the memory at the end of the program.

   Step II – Declare and allocate the GPU memory for `_u`, `_u_new` and `_error`. Copy the CPU memory to the GPU; the other two arrays have been initialized to zero for you. Make sure you free the memory at the end of the program.

   Step III – Set up the kernel execution configuration for the GPU kernel based on the input domain size and the maximum threads per dimension, which is set at the top of the file as a `#define`. You will need to determine the number of threads per block and the number of blocks in each direction, as well as set up the `dim3` variables.

   Step IV – Write a GPU kernel that advances to the next timestep using the Jacobi method. This should be done in parallel, not in serial.

   Step V – Write a GPU kernel that calculates the error between the numerical and analytic solutions at each point. Be careful to compare solutions at the correct timestep – `_u_new` is at $t = (n+1)\Delta t$ and `_u` is at $t = n\Delta t$. Using this result, a parallel reduction using the Thrust parallel algorithms library has been provided to calculate the total error in order to find the average percent error at each grid point.

   Step VI – At the end of the loop, copy the data back to the CPU.

Your program should take as an input the number of grid cells on one side of the domain. This has been set up for you such that the program can be run from the command line like:

```
./jacobi_solver.cu n
```

where $nx = ny =$ `n` is the size of one side of the square domain. The program should output the percent difference between your result and the analytic solution, averaged over all of the grid nodes:

$$\epsilon = \frac{1}{n_x n_y} \sum_{i=1}^{n_x} \sum_{j=1}^{n_y} \frac{u_{i,j}^{(n)} - U_{i,j}^{(n)}}{U_{i,j}^{(n)}}, \tag{6}$$

where U is the analytic solution.

If you have time, discuss the following:

- How does the error change as you increase the resolution? Does this behavior make sense?
- How does the runtime scale with the resolution? You can get a rough estimate by using the bash command `time` when executing your program, like:

  ```
  ./jacobi_solver.cu n
  ```

  and using the result for `real`.

- How does the runtime change as you change the kernel execution configuration? e.g., play around the `MAX_THREADS_DIM` parameter as well as different schemes for setting up the thread blocks.

- Use shared memory in the Jacobi kernel