Team project - Poisson solver

ME 471/571 - Parallel Scientific Computing

Michal A. Kopera

We will work on creating a complete Poisson solver with CUDA. Each team will work on one piece of the code, and we will put them all together on Wednesday.

Just to remind you, the Poisson problem is a boundary value problem given by the partial differential equation:

$$\nabla^2 u = f(x,y),$$

defined here on a unit square $(x,\ y) \in [0,\ 1] \times [0,\ 1]$, with Dirichlet boundary conditions

$$u(x,\ y) = g(x,\ y)$$

prescribed at the boundary. Functions $f(x,\ y)$ and $g(x,\ y)$ are known forcing and boundary condition functions respectively, and for this project are given by:

$$f(x,\ y) = -\ 8\pi^2\ \sin(2\pi x)\ \sin(2\pi y), \tag{1}$$

$$g(x,\ y) = 0.$$

This choice of forcing and boundary condition leads to the exact solution:

$$u_{exact} = \sin(2\pi x)\ \sin(2\pi x)$$

We will solve the Poisson problem numerically, and compare our solution to the exact one. We will use the finite difference method to discretize the differential operators, and Jacobi iteration method to find the solution. This will lead us to evaluating the following algorithm:

repeat until error $< \epsilon$:

$$u_{i,j}^{new} = \frac{1}{4}\left(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - h^2 f_{i,j}\right) \tag{2}$$

$$\text{error} = \frac{||\,u^{new} - u\,||}{N^2}$$

$$u = u^{new}$$

where $h = \frac{1}{N-1}$. The code for this problem would look like this:

1) Allocate necessary memory
2) Initialize the forcing term and solution on GPU
3) Initialize boundary points (i.e. left $(0, k)$, right $(N-1, k)$, bottom $(k, 0)$, top $(k, N-1)$, where $k \in [0, N-1]$) on GPU
4) Initialize the error on CPU to a large value
5) Repeat until the error is smaller than ϵ:

   5)a. evaluate equation (2) on the interior points (i.e. $i, j \in [1, \ldots, N-2]$) on GPU

   5)b. compute error between new and previous solution on GPU

   $$\text{error} = \frac{||u^{new} - u||}{N^2}$$

   5)c. Update solution on GPU ($u = u^{new}$)

To implement the code on CUDA, we will work in teams, where each team will write a piece of code according to specifications outlined below:

Team 1 - Host code

Team 1 will work on setting up the host code which will be ready for plugging in the kernels written by other teams. The host code needs to be in poisson.cu file and accomplish the following tasks:

1. Include kernel file kernels.cu, where the kernel definitions will be stored. For now the kernel.cu has dummy kernels with interfaces and print statements only, for testing purposes.
2. Set problem size N
3. Declare host and device arrays, allocate them on host and device. We will need:

   u - solution array $N \times N$ - host and device

   f - forcing array $N \times N$- device only

   u_new - updated solution array $N \times N$ - device only

   u_exact - exact solution on host only

   error - a scalar variable on both host and device
4. Initialize exact solution on host only. You can re-use the function from poisson.c.

5. Call initialization kernel on GPU. The interface for the kernel is given by:

   __global__ void initData(float *u, float *f, int N)

6. (skip for now) Call boundary conditions kernel on GPU. The interface for the kernel is given by

   __global__ void applyBC(float *u, int N)

7. Initialize the error on host to a large value
8. Loop while error is smaller than pre-defined $\epsilon = 1e - 8$ variable:

   8.1. Call poisson kernel, with interface defined as:

   __global__ void poissonKernel(float *u, float *u_new, float *f, int N)

   8.2. Call error computation kernel:

   __global__ void computeError(float error, float *u, float *u_new, int N)

   8.3. Call update kernel

   __global__ void updateSolution(float *u, float *u_new, int N)

   8.4. Transfer the value of error from device to host

9. Transfer solution array from device to host
10. Compute error between exact and computed solution on host, report the error. You can use L2_error function from the poisson.c file here.

Testing:

Run the host code and check whether each kernel (for now the dummy one) executes as expected. Beware of infinite loop, since your kernels are not updating the error value, so it will never go below desired epsilon.

Team 2 - Initialization and Poisson kernel

Team 2 will write two kernels. You can divide the work between yourselves, or work together on both.

1. *poissonKernel* kernel implements a single update of the Jacobi iteration of the Poisson problem. The kernel needs to execute the following operation on all global points $i, j \in [1, \ldots, N-2]$ in the interior of the domain (note we do not touch boundary points, i.e. i,j = 0 and N-1):

$$u_{i,j}^{new} = \frac{1}{4}\left(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - h^2 f_{i,j}\right),$$

where $h = \frac{1}{N-1}$.

The kernel interface is as follows:

__global__ void poissonKernel(float *u, float *u_new, float *f, int N)

You can choose which decomposition you want to use for this kernel. Looking up examples like finite_diffence.cu and sumMatrix.cu from previous weeks, and your experience with the Project 3, may help.

2. *initData* will set the values of appropriate arrays to the specifications outlined in the problem description above. The device arrays and variables which need to be set are:

   u - solution array which needs to be initialized to 0

   f - forcing array which needs to have values defined by equation (1)

   The interface for the kernel is as follows:

   __global__ void initData(float *u, float *f, int N)

Testing:

Run the test_team2.cu code, making sure that your kernels are placed in the kernels.cu file, which is being included in the test code. The code should show you the values of the device memory for the variables you use, as well as expected values. The test is run on a 6x6 problem size using 2d decomposition of threads and blocks.

Team 3 - Error and update kernel

Team 3 will write two kernels. You can divide the work between teammates, or work on both together. You can utilize dot-product kernels we worked on before spring break, or the norm kernel you might have written for Project 3 mastery question.

1. *computeError* kernel computes the grid-normalized Euclidian norm of the difference between two vectors:

$$\text{error} = \frac{||u^{new} - u||}{N^2} = \frac{1}{N^2}\sqrt{\sum_{i=0}^{N-1}\sum_{j=0}^{N-1}(u_{i,j}^{new} - u_{i,j})^2}$$

The interface for the kernel is as follows:

__global__ void computeError(float error, float *u, float *u_new, int N)

2. *updateSolution* kernel, which copies values from u_{new} to u:

$$u_{i,j} = u_{i,j}^{new}, \qquad i,j \in [0, N-1] \times [0, N-1]$$

The interface for the kernel is as follows:

```
__global__ void updateSolution(float *u, float *u_new, int N)
```

Testing:

Run the test_team3.cu code, making sure that your kernels are placed in the kernels.cu file, which is being included in the test code. The code should report the values of error after initialization (I set u = 0 and u_new = 1, so the error should be 1), and after the update (this time u_new = u, so the error should be 0, or very close).

Optional - BC kernel

applyBC kernel, which imposes boundary conditions on the solution. We will set:

$$u_{i,j} = 0$$

at the boundary point only: left $(0, k)$, right $(N-1, k)$, bottom $(k, 0)$, top $(k, N-1)$, where $k = 0, 1, \ldots, N-1$. The interface for the kernel is:

```
__global__ void applyBC(float *u, int N)
```

Note that you have a choice of decomposition here, and also can exploit the fact that we only touch $4N$ boundary points, not all $N \times N$ points in the domain.