Programming Assignment 3 — Due May 6th 12:50PM

This assignment builds on the previous assignment's theme of examining memory access patterns. You will implement a finite difference 2D heat diffusion solver in different ways to examine the performance characteristics of the different implementations.

Background on the heat diffusion PDE. The heat diffusion PDE that we will be solving can be written:

$$\frac{\partial T}{\partial t} = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}$$

To solve this PDE, we are going to discretize both the temporal and the spatial derivatives. To do this, we define a two-dimensional[1] grid $(G_{i,j})_{\substack{1\leq i\leq n_x\\1\leq j\leq n_y}}$ where we denote by n_x (resp. n_y) the number of points on the x-axis (resp. y-axis). At each timestamp, we will evaluate the temperature and its derivatives at the points of this grid.

While we will consistenly use a first order discretization scheme for the temporal derivative, we will alternatively use a 2^{nd}, 4^{th} or 8^{th} order to discretize the spatial derivative[2].

If we denote by $T^t_{i,j}$ the temperature at time t at point (i,j) of the grid, the 2^{nd} order spatial discretization scheme can be written as:

$$T^{t+1}_{i,j} = T^t_{i,j} + C^{(x)}\left(T^t_{i+1,j} - 2T^t_{i,j} + T^t_{i-1,j}\right) + C^{(y)}\left(T^t_{i,j+1} - 2T^t_{i,j} + T^t_{i,j-1}\right)$$

The $C^{(x)}$ (`xfcl` in the code) and $C^{(x)}$ (`yfcl` in the code) constants are called Courant numbers. They depend on the temporal discretization step, as well as the spatial discretization step. To ensure the stability of the discretization scheme, they have to be less than a maximum value given by the Courant-Friedrichs-Lewy condition[3]. You do not have to worry about this though, because the starter code already takes care of picking the temporal discretization step as to maximize the Courant numbers while ensuring stability.

Boundary conditions. The starter code contains the code that will update the boundary conditions for you (see file `BC.h`, in particular the function `gpuUpdateBCsOnly`) and the points that are in the border (which has a size of `order` / 2). This way, you do not have to worry about the size of the stencil as you approach the wall[4].

Various implementations. In this programming assignment, we are going to implement 2 different kernels (and you can do a third one for extra credit):

- Global (function `gpuStencil`): this kernel will use global memory and each thread will update exactly one point of the mesh. You should use a 2D grid with $n_x \times n_y$ threads total.
- Block (function `gpuStencilLoop`): this kernel will also use global memory. Each thread will update `numYPerStep` points of the mesh (these points form a vertical line). You should also use a 2D grid with $n_x \times n_y/$`numYPerStep` threads total.
- (Extra Credit) Shared (function `gpuShared`): this kernel will use shared memory. A group of thread must load a piece of the mesh in shared memory and then compute the new values of the temperatures on the mesh[5]. Each thread will load and update several elements.

Parameter file. The computation requires 11 diferent parameters to fully specify the problem, so rather than pass them in at the command line we will read them (whitespace separated) from the file `params.in`. You will need to modify *some parameters* (see description of the starter code) in this file to answer the questions.

Here is a list of parameters that are used in the order they appear in the file:

[1] In fact, the size of the grid is $g_x \times g_y$ but we will only update the interior region, which size is $n_x \times n_y$.

[2] For instance, if we use a 4^{th} order scheme, we will express the derivative with respect to x of T at point (i,j) using $T^t_{i-2,j}$, $T^t_{i-1,j}$, $T^t_{i,j}$, $T^t_{i+1,j}$ and $T^t_{i+2,j}$.

[3] If you are interested, you can read more about this at `http://en.wikipedia.org/wiki/Courant-Friedrichs-Lewy_condition`.

[4] A general way of dealing with this problem is to change the size of the stencil as you approach the wall. This is complicated and we simplified the process for this homework.

[5] Note however, that the threads that loaded data on the borders of the small piece will stay idle during the computation step.

```
int  nx_, ny_;   //number of grid points in each dimension
double lx_, ly_;   //extent of physical domain in each dimension
int  iters_;    //number of iterations to do
int  order_;    //order of discretization
double ic_;     //uniform initial condition
double bc[4];    //0 is top, counter-clockwise
```

The starter code is composed of the following files:

- `2dHeat_driver.cu` — This is the CUDA driver file. You may modify this file, but it should not be necessary.
- `gpuStencil.cu` — This is the file containing the kernels. You will need to modify this file.
- `Makefile` — `make` will build the heat binary. `make clean` will remove build files as well as debug output. You do not need to modify this file.
- `params.in` — This file contains a basic set of parameters. For debugging, performance testing, and to answer the questions, you will need to modify this file. The only parameters you should modify in this file are `nx`, `ny` (line 0) and `order` (line 3).
- `simParams.h` and `simParams.cpp` — These files contain the data structure necessary to handle the parameters of the problem. You do not need to modify these files.
- `Grid.h` and `Grid.cu` — These files contain the data structure that models the grid used to solve the PDE. You do not need to modify this file.
- `BC.h` — This file contains the class `boundary_conditions` that will allow us to update the boundary conditions during the simulation. You do not need to modify this file.

To run the code, you need to `make`. Once this is done, you can type:

```
./heat [-g] [-b] [-s]
```

where `-g` stands for global, `-b` for block, and `-s` for shared. Note that you can run several of the implementation at the same time (for instance `./heat -g -b` to run the global and block implementations).
The output produced by the program will contain:

- The time and bandwidth for the CPU implementation, as well as for the implementations that you specified when running the program.
- A report of the errors for the GPU implementations. Namely, the program will output:
  - The L_2 norm of the CPU implementation (*i.e.* the reference).
  - The L_∞ norm of the relative error.
  - The L_2 norm of the error normalized by the L_2 norm of the CPU implementation.

Typical ranges for the errors (and for the parameters that you will be using) are:

- $\left[10^{-7}, 10^{-5}\right]$ for L_∞ error.
- $\left[10^{-8}, 10^{-6}\right]$ for L_2 error.

To have a good idea of what you need to implement, search for TODO in `gpuStencil.cu`.

Question 1

(30 points) Implement the function `gpuStencil` that runs the solver using global memory. You must also fill in the function `gpuComputation`.

Question 2

(35 points) Implement the function `gpuStencilLoop` that runs the solver using global memory but where each thread computes several points on the grid. You must also fill in the function `gpuComputationLoop`.

Question 3

(15 points) Plot the bandwidth (GB/s) as a function of the grid size (in MegaPoints) for the following grid sizes: 256×256; 512×512; 1024×1024; 2048×2048; 4096×4096.

You must have 2 plots (or 3 plots if you choose to do the extra credit) as follows:

1. For `order` = 4, plot the bandwidth for the 2 (or 3) different algorithms.
2. For the block implementation, plot the bandwidth for the 3 different orders.
3. If you implemented the shared algorithm, plot the bandwidth for the 3 different orders.

Question 4

(20 points) Which kernel (global, block or shared) and which order give the best performance (your answer may depend on the grid size)? Explain the performance results you got in Question 3.

Question 5

(20 points Extra Credit) Implement the function `gpuShared` that runs the solver using shared memory. You should also fill in the function `gpuComputationShared`.

Total number of points: 120

A Submission instructions

You should submit one zip file containing a folder named `FirstName_LastName_SUNetID_PA3`. Make sure to have a folder with this name otherwise your files will get mixed up with other students when we unzip your file. This folder should contain:

- `gpuStencil.cu`
- a pdf file named `Readme.pdf` containing your answers to the questions (and the plots)

The zip file should be named `PA3.zip`.

To run (and grade) your code, we will copy your source file in our directory and type:

- `make`
- `./heat`

B Hardware available for this class

Machines

We will be using the icme-gpu teaching cluster, which you can access with ssh:

```
ssh sunet@icme-gpu1.stanford.edu
```

We have only provided accounts for those enrolled in the course. If you are auditing the course, we may consider a special request for an account.

Compiling

To use nvcc on the icme cluster, you must copy and paste these lines to your `.bashrc` file (so they will be loaded when you connect on the cluster).

```
module add open64
module add cuda50
```

For the first time, after you changed the `.bashrc`, you need to logout and re-login (so that the changes are taken into account) or source the `.bashrc`.

You can now use `nvcc` to compile CUDA code.

Running

The cluster uses MOAB job control. The easiest way to run an executable is by using interactive job submission. First enter the command

```
msub -I -l nodes=1:gpus=1
```

You can then run any executable in the canonical fashion. For example

```
./heat [-g] [-b] [-s]
```

as you would on your personal computer.

C Advice and Hints

- Make sure you understand what are the parameters of the problem. In particular the difference between `nx` and `gx` is important to understand.
- Spend some time looking at the `simParams` class as it contains the useful parameters to solve the problem.
- Make sure your implementations are able to deal with square grids as well as rectangular ones.