Programming Assignment 2

Due April 25, 2012 12:50 PM

In this assignment you will be using CUDA to implement a 2-dimensional heat diffusion algorithm. In the process you will learn more about the memory structure in CUDA, and the importance of optimizing your memory accesses.

1 Background

The computation requires several parameters, so rather than pass them in at the command line we will compute them from the file `params.in`. Here is a list of parameters that are used:

```
int    nx_, ny_;      //number of grid points in each dimension
int    gx_, gy_;      //number of grid points including halos
double lx_, ly_;      //extent of physical domain in each dimension
double alpha_;        //thermal conductivity
double dt_;           //timestep
int    iters_;        //number of iterations to do
double dx_, dy_;      //size of grid cell in each dimension
double ic_;           //uniform initial condition
double xcfl_, ycfl_; //cfl numbers in each dimension
int    order_;        //order of discretization
int    borderSize_;  //number of halo points
double bc[4];         //0 is top, counter-clockwise
```

These parameters define the number of particles in the grid, the rate of diffusion, and the amount of time to model the system. An important parameter is the *order*. This controls how many neighbors we will consider in the computation. With a higher order, we will consider both our immediate neighbors and neighbors further away. You will use the points whose distance along the x and y dimensions are at most halo = order/2 from the current point. We call these points the *stencil*. The distance from the center points changes the *weight* of a point in the computation, which you will see in our CPU implementation.

2 What we give you

- `2dHeat.cu` - This is the starter code for your implementation. You should add CUDA code here to implement the 2-D Heat Diffusion algorithm using both global and shared memory. You should modify this file.
- `Makefile` - `make` will build the 2dHeat binaries. `make clean` will remove the executable. You should be able to build and run the program when you first download it, however only the host code will run. You do not need to modify this file.
- `mp1-util.h` - This contains some support functionality. You do not need to modify this file
- `params.in` - This file contains some parameters and input to the algorithm. You may modify this file if you would like to test different configurations. You should change the size of the grid (the first line) and the order (the fifth line) during your analysis.

These files can be found in `/afs/ir/class/cme213/assignments/pa2` on Leland machines (such as the corn and pup clusters).

3 Your Tasks

We already have a CPU implentation in `2Dheat.cu`. You are tasked with implementing a 2D heat diffusion using both *global* and *shared* memory.

3.1 Code Implementation

Your first task is to correctly implement the algorithm using CUDA. For the questions below, you will be graded based on correctness.

- Implement the algorithm using global memory. You may not use shared memory in this part of the problem. We provide some starter code for implementing the algorithm in CUDA. You should create one thread for every point in the grid. You will have to do the following:
- Calculate the correct grid dimension
- Implement the update algorithm for the 2^{nd}, 4^{th}, and 8^{th} order stencils. You should use the CPU implementation as reference.

- Implement the algorithm using shared memory for better performance. You must use shared memory in this part of the problem. Recall that you can declare memory as being shared with the `__shared__` keyword. Shared memory can provide as much as two orders of magnitude of speedup, so you should see a significant improvement using shared memory. We have only given you the kernel declarations. You must:
- Calculate the input to the `gpuShared` template parameters for each of the order methods.
- Correctly load the grid into shared memory.
- Implement the update algorithm for the 2^{nd}, 4^{th}, and 8^{th} order stencils. You should use the CPU implementation as reference.

Hints

- It is more efficient to represent a matrix as a one dimensional array than a two dimensional array.
- In order to avoid unnecessary code duplication, we use both *wrappers* — methods which interface into sub methods and *templates* — type and value parameters which can be decided at compile time. This allows us to create cleaner code and compile a little more quickly. Your code should preserve these properties.
- To get more speedup, avoid *bank conflicts* by ensuring that you are not sending (reading) multiple requests to the same bank from the same warp.
- Memory accesses are very expensive. *Coalesing* accesses allows for better performance.

3.2 Analysis

Your second task is to analyze your code by doing the following. Make sure to obtain all the plots and to comment on your observations.

1. When plotting against size of the grid, plot points at 1000 x 1000, 1000 x 2000, 2000 x 2000, 2000 x 4000, and 4000 x 4000. Your y-axes should be both GBytes/second (memory) and GFlops/second (computation).
2. Plot bandwidth vs. size of the grid, and flops vs. size of the grid for each of order. Do this for shared memory on one plot and global memory on a separate plot.
3. How does the shared memory performance compare with the cache only performance? Explain why one outperforms the other. Do larger orders favor the cache or shared memory implementation? Try to explain all your findings.
4. [Extra Credit — 5%] Create a plot of the ratio of double to float for bandwidth and flops against the size of the input. Do this only for order 4.
5. [Extra Credit — 5%] How do the flops and bandwidth compare between float and double?

4 Scoring

Correctness 40%. Ensure that all orders are correct for both global and shared memory. You should be able to diff against the cpu output. The code we provide will supply some information about incorrect values.

Analysis 60%.

5 Submission instructions

You should submit the complete source code for your working solution and a brief pdf file (no more than one or two pages excluding plots) containg your analysis.

To submit:

- Create a directory, for example `pset2`, on your Leland account.
- Copy the files you would like to submit in the directory `pset2`. Do not include auxiliary files that are not needed.
- From within the directory `pset2`, run the shell command
  `corn:> /afs/ir/clas/cme213/bin/submit pa2`
  This will send the entire content of the directory to us and will label it `pa2`.

A timestamp is also included. We will use your last submission before the deadline for grading. Please send a message to the course staff through piazza if you encounter a problem.

6 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.

To learn more about these machines and how to use them, visit
`http://icme.stanford.edu/Computer Resources/gpu.php`
The tutorial is very helpful if you are unsure how to proceed.

Compiling

To use nvcc on the icme cluster, you must copy and paste these lines to your .bashrc file:

```
module add open64
module add openmpi
module add cuda40
module add torque
```

You can now use nvcc to compile CUDA code. We have provided a makefile which will compile 2dHeat by typing `make`.

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 nodes=# :ppn=# :gpus=#
```

where # correspond to the number of nodes, processors, and gpus respectively. For this assignment, you can use 1 for every value.

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

```
./2dHeat
```

as you would on your personal computer.