CME 213, Introduction to parallel computing
Eric Darve
Spring 2024

Stanford University

Homework 3

Total number of points: 100 (+ 10 bonus).

This assignment builds on the previous assignment's theme of examining memory access patterns. You will implement a finite difference solver for the 2D heat diffusion equation in different ways to examine the performance characteristics of different implementations. Please refer to the class and lecture slides on this homework.

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 grid $G_{i,j}$, $1 \le i \le n_x$, $1 \le j \le n_y$[1], where n_x is the number of points on the x-axis and n_y is the number of points on the y-axis. At each time step, we will evaluate the temperature and its derivatives at these grid points.

While we will consistently use a first-order discretization scheme for the temporal derivative, we will use a 2nd, 4th, or 8th order discretization scheme for the spatial derivative.[2]

If we denote by $T^t_{i,j}$ the temperature at time t at a point (i, j), the 2nd 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 constants $C^{(x)}$ (`xfcl` in the code) and $C^{(y)}$ (`yfcl` in the code) 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 the maximum value given by the Courant-Friedrichs-Lewy condition.[3] The starter code we provide takes care of picking the temporal discretization step to maximize the Courant numbers while ensuring stability.

The starter code also contains host and device functions (`stencil` in `CPUComputation.h` and `Stencil` in `gpuStencil.cu`, respectively) which contain the coefficients that go into the update equation. Therefore, you do not need to figure out how to implement the different order updates. You only need to pass the correct arguments when calling the device function.

Boundary conditions. The file `BC.h` in the starter code contains functions that will update the boundary conditions and the temperature at points in the border for you. Therefore, you do not have to worry about the size of the stencil as you approach the wall.[4]

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

[2]For instance, if we use a 4th order discretization scheme, we will express the derivative of T with respect to x at a 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.

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

- Global (function `gpuStencilGlobal`): this kernel will use global memory. Each thread should update exactly one point of the mesh. You should use a 1D `Grid` and 1D `Blocks` with $n_x \times n_y$ threads total.
- Block (function `gpuStencilBlock`): this kernel will also use global memory. Each thread should update `numYPerStep` points of the mesh (these points form a vertical line). You should use a 2D grid with $n_x \times \frac{n_y}{\texttt{numYPerStep}}$ threads total.
- (Extra Credit) Shared (function `gpuStencilShared`): this kernel will use shared memory. A group of threads should load a piece of the mesh into shared memory and then compute the new values of the temperatures on the mesh. Each thread should load and update several elements.[5]

Starter code. The starter code is composed of the following files (`*` means the file will *not* be submitted by our script):

- `*BC.h` — This file contains the class `boundary_conditions` that will allow us to update the boundary conditions during the simulation. Do not modify this file.
- `*CPUComputation.h` — This file contains functions related to the CPU implementation of the solver. The CPU implementation should help you understand the solver and help write your GPU implementation. Do not modify this file.
- `*Errors.h` — This file contains the functions we will use to test your code. Specifically, these functions will check for differences between the output of your implementation and the CPU implementation, and write any errors to different file streams. Do not modify this file.
- `gpuStencil.cu` — This file contains kernels and wrapper functions that you need to implement. Please fill in the sections marked `TODO` in this file.
- `*Grid.h` and `*Grid.cu` — These files contain the data structure that models the grid used to solve the PDE. Do not modify these files.
- `*hw3.sh` and `*run.sh` — The script `hw3.sh` is used to submit jobs to the queue. The script `hw3.sh` calls the script `run.sh` which runs the executable `main` with different combinations of parameters. Do not modify these files.
- `*main.cu` — This file contains the `main` function and one test for each of the kernels we ask you to implement. When you run the executable `main`, the `main` function will be called and will initialize some global variables so that we may use the same starting grid for all three of our tests: `global`, `block`, and `shared`. After initialization in `main`, the script will execute the tests in order. Note that if you aren't doing the extra credit (i.e., implementing the shared kernel), your code will always fail the last test. Do not modify this file.
- `Makefile` — Running `make` will produce an executable `main`. Running `make clean` will remove the build files as well as debug output. You may also change the `nofma` flag by commenting out line 30 of the `Makefile`. This flag is further discussed below.
- `mp1-util.h` — This file contains utilities used in `CPUComputation.h` and `Errors.h`. Do not modify this file.

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

- `*params.in` — The parameters used in the computation are read from the file `params.in`. You will need to modify this file (except the second line) for debugging, performance testing, and to answer the questions. Here is a list of the parameters in the order that they appear in the file `params.in`:

```
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 scheme for the spatial derivative
```

- `plot.py` — This script can be used to generate plots for Question 1 by running the command `python plot.py name_of_your_csv.csv`. You may need to replace `python` with `python3`, and the script expects that the `csv` was generated using the member function `saveStateToFile` of the class `Grid`. If you choose to use this script, you will need to run it in your own Python environment since some required packages are not installed on `cme213-login.stanford.edu`.

- `*simParams.h` and `*simParams.cpp` — These files contain the data structure necessary to handle the parameters of the problem. Do not modify these files.

Running the program. Type `make` to compile the code. Once this is done, you can use the command

```
$ srun -p gpu-turing --gres=gpu:1 ./main
```

to run the executable using the parameters in `params.in`, or just execute

```
$ sbatch hw3.sh
```

to run with all of the different parameter combinations you may need to answer the questions below.

The output produced by the program will contain:

- The time and bandwidth for the CPU implementation and the different GPU implementations.

- A report of the errors for the different implementations. If A is the CPU solution and B is the GPU solution, the program will output:
  - `L2Ref`=$\sqrt{\frac{1}{\text{gx}\cdot\text{gy}}\sum_{i=1}^{\text{gy}}\sum_{i=1}^{\text{gx}} A_{ij}}$, the L_2 norm of the CPU solution.
  - `L2Inf`=$\max_{1\le i\le \text{gy}, 1\le j\le \text{gx}}|A_{ij}-B_{ij}|$, the L_∞ norm of the error between the CPU solution and the GPU solution.
  - `L2Err`=$\sqrt{\frac{1}{\text{gx}\cdot\text{gy}}\sum_{i=1}^{\text{gy}}\sum_{i=1}^{\text{gx}} (A_{ij}-B_{ij})}$, the L_2 norm of the error between the CPU solution and the GPU solution.

The output of running the `hw3.sh` script can be found in the files `result.txt`, `globalErrors.txt`, and `sharedErrors.txt`. With `-gpu=nofma`, the errors should be equal to 0 and the files `globalErrors.txt` (and `sharedErrors.txt` if you implemented the shared kernel) should be empty. We will test your code with this flag, so make sure that your code passes all of our tests with this flag set.

Without `-gpu=nofma`, typical ranges for roundoff errors are:

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

- $\left[10^{-9}, 10^{-8}\right]$ for L_2 norm error.

~~Question 1~~

(30 points) Implement the function `gpuStencilGlobal` that runs the solver using global memory and creates 3D surface plots of temperature on a 256×256 grid at iteration 0, 1000, and 2000 respectively, with an 8th order discretization scheme. You must also fill in the function `gpuComputationGlobal`. The difference (in terms of the norms) between your implementation and the CPU implementation should be in the expected range. The class `Grid` has a member function `saveStateToFile` to dump all the data of the grid to a `CSV` file. You can call this function in e.g. the for loop in `gpuComputationGlobal` to save the state of the grid at a particular iteration. You can use `plot.py` or your own tools (Python, MATLAB, etc.) to generate the plots and include them in your writeup.

~~Question 2~~

(35 points) Implement the functions `gpuStencilBlock` and `gpuComputationBlock`. You should use 2D blocks and grids to implement the blocking strategy we talked about in class. You are responsible for choosing block and grid dimensions as well as the value of `numYPerStep` to optimize the performance of your code. The difference (in terms of the norms) between your implementation and the CPU implementation should be in the expected range.

Question 3

(15 points) Plot the bandwidth (in GB/s) as a function of the grid size (in MegaPoints, i.e. in millions of points) for the following grid sizes: 256×256; 512×512; 1024×1024; 2048×2048; 4096×4096. You can choose the number of iterations to run to generate your results. You should run enough iterations so that results are stable and meaningful (clearly 1 is not enough).

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

1. For `order` = 8, 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.

Use a log scale for the x-axis of your plots.

Question 4

(20 points) Which kernel (global, block, or shared) and which order (2, 4, or 8) gives the best performance? Your answer may depend on the grid size. Explain your results from Question 3. Specifically, explain performance differences (i) among kernels, (ii) from varying order, and (iii) from varying grid size.

Question 5

(Extra credit 10 points) Implement the function `gpuStencilShared` that runs the solver using shared memory. You should also fill in the function `gpuComputationShared`. Note that you have to answer the questions related to shared memory implementation in Questions 3 and 4 to get the full extra credit.

A Submission instructions

1. For all questions that require answers besides source code, put those answers in a separate PDF file and upload this file on Gradescope.
2. Make sure your code compiles on `cme213-login.stanford.edu` and runs. To check your code we will run `make` on `cme213-login.stanford.edu`, and this should produce an executable `main`.

3. Copy your submission files to `cardinal.stanford.edu`. The script `submit.py` will copy only the files below to a directory accessible to the CME 213 staff. Any other required files (e.g., Makefile) will be copied by us. Therefore, make sure you make changes only to the files below. You are free to change other files while debugging, but make sure you test with the default files before submitting. Do not use external libraries, additional header files, etc. that would prevent the teaching staff from compiling the code successfully. Here is the list of files we are expecting and that will be copied:

   ```
   gpuStencil.cu
   ```

4. To submit, execute

```
$ /afs/ir.stanford.edu/class/cme213/script/submit.py hw3 <directory with your submission files>
```

5. You can submit up to 10 times before the deadline and each submission will replace the previous one.

B Advice and hints

- Make sure you understand each parameter of the problem by looking at the class `simParams`. In particular, the difference between `nx_` and `gx_` is important to understand.
- Keep in mind where your data are when choosing kernel parameters. You will be reading from global memory using read-only cache (L1, 128-byte cache lines). Writes to global memory cannot be cached in L1, therefore have to go to the shared L2 cache, which has 32-byte cache lines.
- Make sure your implementations can deal with rectangular grids as well as square ones.