CME 213, Introduction to parallel computing
Eric Darve
Spring 2024

Stanford University

Homework 4 — Solution

Total number of points: 100 (+ 15 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[1] grid $G_{i,j}$, $1 \le i \le n_x$, $1 \le j \le n_y$, where we denote by n_x (resp. n_y) the number of points on the x-axis (resp. 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 2nd, 4th or 8th order discretization of 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 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 $C^{(x)}$ (`xfcl` in the code) and $C^{(y)}$ (`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 the maximum value given by the Courant-Friedrichs-Lewy condition.[3] You do not have to worry about this, because the starter code already 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 named `stencil` 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 understand how this function works and pass in the arguments correctly.

Boundary conditions. The starter code contains the functions 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]

[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 4th 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.

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 `gpuStencilGlobal`): this kernel will use global memory and each thread will 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 will update `numYPerStep` points of the mesh (these points form a vertical line). You should use a 2D grid with $n_x \times n_y/$`numYPerStep` threads total.
- (Extra Credit) Shared (function `gpuStencilShared`): this kernel will use shared memory. A group of threads 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 parameters used in the computation are read from the file `params.in`. You will need to modify *some parameters* (see description of the starter code) in this file to answer some of the questions. But this file will not be submitted through the submission script.

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

```
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
```

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

- `*main.cu` — This is the CUDA driver file. Do not modify this file.
- `gpuStencil.cu` — This is the file containing the kernels. You will need to modify this file.
- `Makefile` — `make` will build the binary. `make clean` will remove the build files as well as debug output. You may also change the `nofma` flag. More about this flag is discussed below.
- `*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 are `nx`, `ny` (line 1), and `order` (line 4). This file however will not get submitted through the script.
- `*simParams.h` and `*simParams.cpp` — These files contain the data structure necessary to handle the parameters of the problem. Do not modify these files.
- `*Grid.h` and `*Grid.cu` — These files contain the data structure that models the grid used to solve the PDE. Do not modify this file.
- `*BC.h` — This file contains the class `boundary_conditions` that will allow us to update the boundary conditions duri I ng the simulation. Do not modify this file.
- `*CPUComputation.h` — This file contains the functions related to the CPU implementation of the solver. Do not modify this file

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

- `*Errors.h` — This file contains the functions we will use to test your code, i.e., to check the errors of the output and print out any errors to different file streams.
- `*hw3.sh` — This script is used to submit jobs to the queue.

Note The files in the starter code contain some additional information about the implementation in the form of comments. Additionally, the CPU implementation should provide a clearer picture of the method and should aid your GPU work.

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
```

or just run

```
$ sbatch hw3.sh
```

which has all of the different param combinations we will ask about

When your run this program, the main function will be called and will initialize some global variables so that we may reuse the starting grid for all three of our tests `global`, `block`, and `shared`. Once the initialization is done in main, then the script will execute the tests in order. Note that if you aren't doing the extra credit (i.e., implementing the shared kernel), then your code will always fail the last test. This is expected. The output produced by the program will contain:

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

The output of `hw3.sh` script can be found in the `result.txt` file that will be created when you run the script. With `-gpu=nofma`, the errors should be equal to 0. We will test your code with this flag, so make sure that your code passes all of our tests with this flag set. You may set the flag to true when calculating the bandwidth scores.

Without this option, 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.

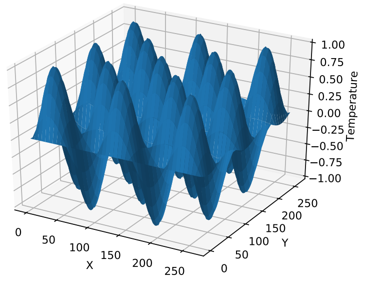
Search for `TODO` in `gpuStencil.cu` to see where you need to implement code.

Question 1

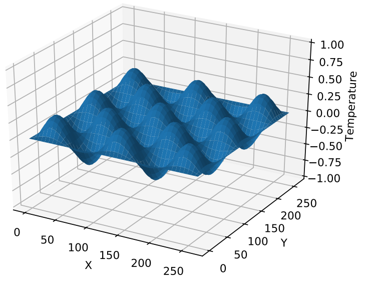
(30 points) Implement the function `gpuStencilGlobal` that runs the solver using global memory, and creates 3D surface plots of temperature on a 256×256 grid at iteration 0, 1000, and 2000 respectively, with 8th order. You must also fill in the function `gpuComputationGlobal`. The difference (in terms of the norms) between your implementation and the reference should be in the expected range. `Grid` class implements member function `saveStateToFile` to dump all the data of the grid to a `CSV` file. You can choose your own tools (Python, MATLAB, etc.) to generate those plots and include them in your writeup.

Solution

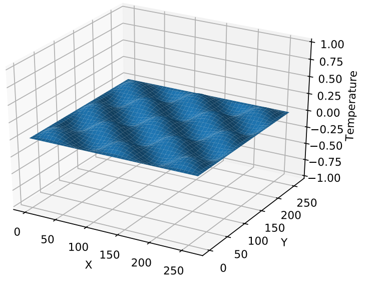
See code and Figure 1.



(a) Iteration 0



(b) Iteration 1000



(c) Iteration 2000

Figure 1: Surface plots of temperatures

Question 2

(35 points) Implement the function `gpuStencilBlock` that runs the solver using global memory but where each thread computes `numYPerStep` points on the grid. You must also fill in the function `gpuComputationBlock`. You should use 2D blocks and grids to implement the blocking strategy we talked about in class. You need to decide your block and grid dimensions, as well as `numYPerStep` to optimize the performance of your code. The difference (in terms of the norms) between your implementation and the reference should be in the expected range.

Solution

See code.

Question 3

(15 points) Plot the bandwidth (GB/s) as a function of the grid size (in units of MegaPoints) 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 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.

Solution

See Figure 2 and 3.

Question 4

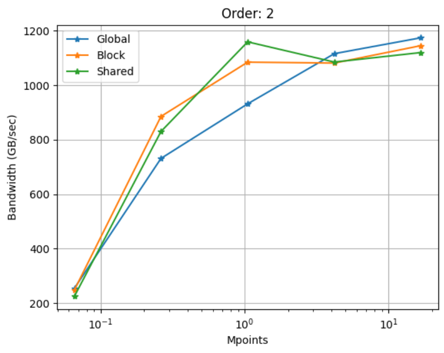
(20 points) Which kernel (global, block, or shared) and which order gives the best performance (your answer may depend on the grid size)? Explain the performance results you got in Question 3. Specifically, explain performance differences (i) among kernels, (ii) from varying order, and (iii) from varying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

Solution

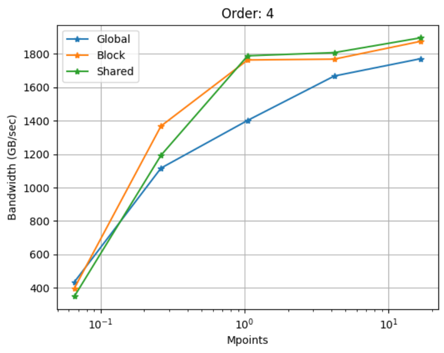
The highest bandwidth (~800 GB/s) is achieved by the shared implementation at order 8 for the biggest problem size.

Comparison of kernels The performance increase from the global implementation to the block implementation is due to better reuse of the cache. The square block pattern allows for greater memory reuse resulting in a lesser number of bytes to be actually read from global memory, and therefore higher flops/byte accessed. This makes the kernel more efficient in terms of the time taken and increases the bandwidth. If not for cache reusability, these two implementations would not have any difference other than the number of blocks/threads launched.

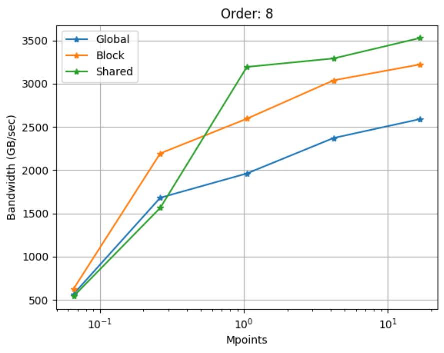
The shared memory version outperforms the global/block implementations because we get to manage the memory use more optimally. In the other two versions, although there is some memory reuse from the cache lines, a significant portion of it gets evicted and re-read since we can't explicitly control it.



(a) 2nd order

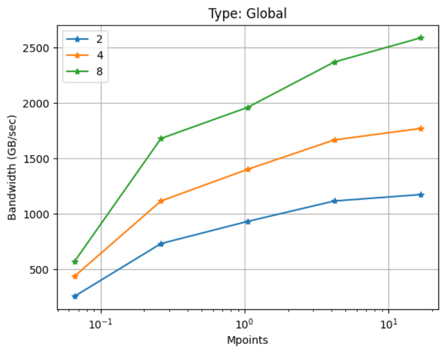


(b) 4th order

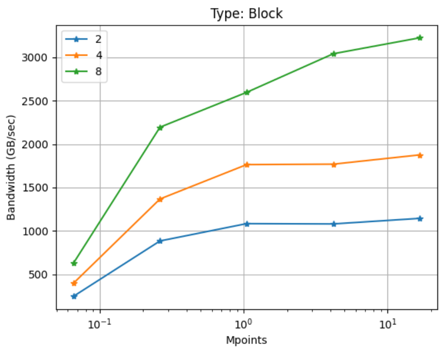


(c) 8th order

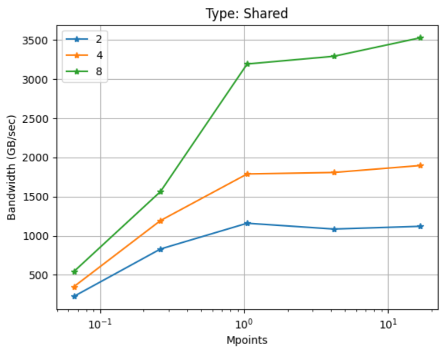
Figure 2: Comparison of bandwidth for the different implementations with different orders of discretization.



(a) Global



(b) Block



(c) Shared

Figure 3: Comparison of bandwidth for different orders of discretization for the different implementations.

Bandwidth increase with order Note that the bandwidth is computed as

$$\text{Bandwidth} = \frac{\text{No. of bytes accessed by kernel}}{\text{Time taken by the kernel}}$$

As order increases, the words per each stencil calculation increases (6, 10, and 18) for the three orders respectively and therefore the numerator increases proportionally. However, the time taken can be split as Time taken for global memory accesses + Time taken for shared memory/cache accesses. As the order increases for a particular grid size, the number of global memory accesses increases only by a small fraction whereas most of the extra bytes in the higher-order stencils come from cache/shared memory. Since the time required for global memory access is significantly larger than that for shared memory, the increase in the time taken does not compensate for the increase in the number of bytes and hence we see a bandwidth increase.

Bandwidth increase with problem size The major reason for the bandwidth increase with problem size is due to the effect of boundaries. Notice that, as the problem size increases, the number of points on the boundary increases only linearly compared to the number of points in the interior of the domain. Therefore, the boundary takes a lesser and lesser fraction of the time for interior calculations. Additionally, as the problem size gets larger, the boundary condition update kernel will have higher occupancy as well.

Secondly, there is a decrease in the number of blocks/threads on the edges of the domain which could be idle (relative to the total number of blocks).

Question 5

(Extra credit 15 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.

Solution

See code.

A Submission instructions

To submit:

1. For all questions that require explanations and answers besides source code, put those explanations and answers in a separate PDF file and upload this file on Gradescope.

2. Make sure your code compiles on `icme-gpu` and runs. To check your code, we will run:
   ```
   $ make
   ```
   This should produce one executable: `main`

3. The homework should be submitted using a submission script on `cardinal`. The submission script must be run on `cardinal.stanford.edu`.

4. 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. Only these files will be copied. 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 for your own debugging purposes, but make sure you test it with the default test files before submitting. Also, 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
```

5. To submit, type:

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

6. You can submit at most 10 times before the deadline; each submission will replace the previous one.

B 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.
- Keep in mind where your data are when trying to decide kernel parameters. You will be reading from global memory using Read-Only cache (L1, 128-byte cache lines). Writes to global memory can not 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 square grids as well as rectangular ones.