INF3201- Parallel Programming

Assignment 3 - Heat Distribution Problem with Cuda and OpenMP

9 October - 29 October 2020

1 Heat Distribution Problem (HDP)

An area has known temperatures along each of its edges as seen in Figure 1. Three are kept cold at absolute zero ($-273.15°C$). The forth side, the top one, is hot ($40°C$). For HDP[5], we simulate how the temperatures spread in the system and stabilize to some temperature gradient.

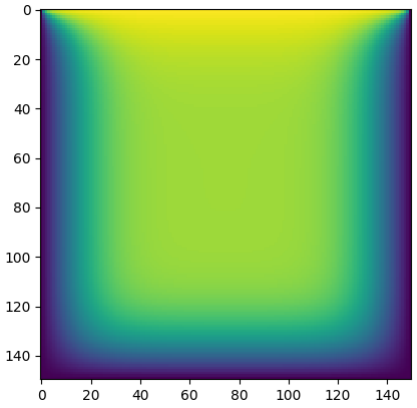

Figure 1: Heat Distribution visualization

2 Task

Using the Successive Over-Relaxation (SOR) method, implement and parallelize three techniques to solve the Heat Distribution Problem using both CUDA[3] C and OpenMP[1] written in C/C++. See Section 6 for more details.

Successive Over-Relaxation is used to calculate a temperature for each point for every iteration based on its neighbors. See Figure 2 and read more in [7] Sec. 2.1.

Your solution should terminate the computations when the system converges: when the calculations reach a point where the difference between iterations is below a set threshold..

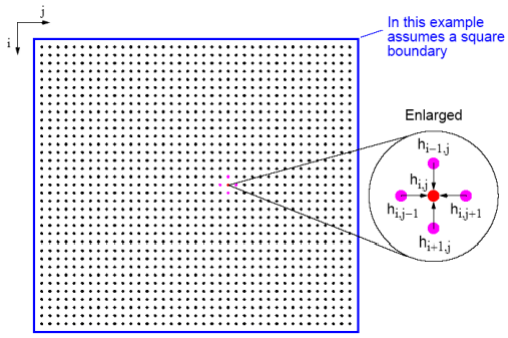

Figure 2: Calculating a single heat point

2.1 Simple

The *Simple* technique is a naive way to implement the SOR. Basically iterating through all the points in the HDP plane and calculate. See Listing 1. This technique is non-deterministic in the parallel version.

```
do
  diff=0.0
  for i = 1 to n-1
    for j = 1 to n-1
      temp = A[i,j]
      A[i,j] = (omega / 4.0) * (A[i+1,j] + A[i-1,j] + A[i,j+1] + \
        A[i,j-1]) + (1 - omega) * A[i,j]
      diff = diff + abs (A[i,j] - temp)
while diff > epsilon
```

Listing 1: Simple technique

The omega parameter is used to balance the weight of the old value in `A[i,j]` vs. the neighbour cells when computing the new values for the cell.

2.2 Red-Black Ordering

A technique where points from HDP are divided into alternating red and black "dots". A red "dot" is updated only looking at black "dots" and vice versa. See Figure 3. This technique is deterministic and should return the same result always - for the same input parameters. Read more in [8] Sec. 5.6.2 p. 233.
In the book it is referred to as `Odd-even transposition sort`, `red-black ordering` follows the same principle excluding the sort.

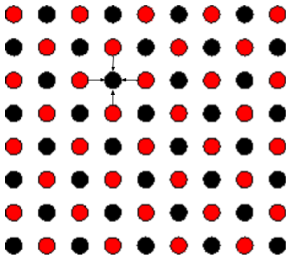

Figure 3: Red-Black Ordering

2.3 Double Buffered

A technique where two copies of the system are stored at the same time. Calculations and updates are done on one copy at a given iteration using the values from the second copy - which stores "older" values. The next iteration updates values in second copy using values from previously updated first copy. It is important to instantiate both copies before starting calculations. See Listing 2. This technique is also deterministic and should return the same result always - for the same input parameters.

```
for x in range(0, some_number):
    if x % 2 == 0:
        array_a = update(array_b)
        continue
    else:
        array_b = update(array_a)
```

Listing 2: Double buffering example

3 Precode

3.1 Cuda precode

In the repository you can find a directory called `Code`. Inside that directory there is another one called `FrostTrap_Cuda` where you will find a few files.

- `FrostTrap.py` - contains precode for the assignment. If you have CUDA environment installed correctly you should be able to run the code. This python code takes several optional parameters as input. To know the parameters please run:
  `python3 FrostTrap.py --help`
  For example, if you want to solve and visualize the HDP with the *simple* method then the execution command will be:
  `python3 FrostTrap.py simple -g`
  When you run this command the python code will look for the *simple* kernel (In CUDA programming, a kernel is a function) in `kernel.cu` and

run the solution. Because of the command line argument `-g`, the program will show the visualization of HDP while it is running. The program will be terminated automatically when it converges with the given threshold. [1]

- `kernel.cu` - Here you need to implement the three techniques to solve the HDP (simple, rb and double buffer) using CUDA C. The functions that you need to implement (simple, rb and double-buffer) are already created with the required parameters with an empty body.See Listing 3.

```
/*
 * simple
 */
__global__ void simple(float *trap, int h, int w, float omega,
float epsilon, int iter, float *delta) {
  // implement me
}

/*
 * rb
 */
__global__ void rb(float *trap, int h, int w, float omega,
float epsilon, int iter, float *delta) {
  // implement me
}

/*
 * dbuf
 */
__global__ void dbuf(float *trap, int h, int w, float omega,
float epsilon, int iter, float *delta) {
  //implement me
}
```

Listing 3: kernel.cu

3.2 OpenMP precode

You will also find another sub-directory within the directory called `code` called `FrostTrap_OpenMP` within you will find some files.

- `FrostTrap.c` - contains precode for the assignment. When you compile the given file you should be able to run it. The executable takes two optional parameters as an input:
  `./FrostTrap [simple|dbuf|rb] [dump]`.
  The latter parameter determines whether the executable should save data during the execution taht can be used to draw heat distribution later - check file plot_data.py for more details. In order to make your life easier you can modify the code from Listing 3 by adding your own cases.

[1] NOTE: you don't need to change anything in the `FrostTrap.py`

- `FrostTrap_numpy.py` - is a python3 implementation (using numpy arrays) of the HDP. It can be used as a help in understanding the C code or/and to check other sequential implementations.

- `plot_data.py` - python3 code used to draw heat distribution from data saved by `FrostTrap.c`. It can be used for a visual validation of your solution.

```
    switch(argc) {
        case 3:
            if (strcmp("dump", argc[2]) == 0) {
                DO_DUMP = 1
            }

        case 2:
            if (strcmp("simple", argv[1]) == 0) {
                cur_solver = solve_simple;
            }
            else if (strcmp("dbuf", argv[1] == 0) {
                cur_solver = solve_dbuf;
            }
            else if (strcmp("rb", argv[1] == 0) {
                cur_solver = solve_rb;
            }
            else {
                printfn ("Can't figure out which you want. Assuming
     simple. \n);
                cur_solver = solve_simple;
                scheme = "simple;
            }
            scheme = argv[1];
            break;
        default:
            scheme = "simple";
            cur_solver = solve_simple;
    }
```

Listing 4: FrostTrap.c

4 Environment

You are free to choose between your own computer with NVIDIA GPU or the uvcluster (See section 5). In order to run and profile the program correctly in your computer, make sure that you have Numpy[2], Pycuda[6], CUDA Toolkit[3], and Java-8-JDK (required by NVIDIA visual profiler) installed correctly.

5 uvcluster

At the Computer Science department we have a small cluster of nodes with Nvidia GPUs that can be used to run the parallelized solutions written in CUDA C. The cluster has a distributed file system which means that you will need to copy your files only to the frontend server (uvcluster.cs.uit.no) in order to access them from all the other nodes in the cluster.

In order to login to the cluster please use the following command (in linux): `ssh abc123@uvcluster.cs.uit.no` where abc123 is your UiT login.

Please make sure that you are familiar with welcome message from the cluster - see Listing 5. For the given assignment you should list the hardware with NVIDIA GPUs list(3), pick up a node with NVIDIA GPU (ex. compute-1-0), check the load of the selected nodes(4), ssh to the selected node (ex. abc123@compute-0-10) and run your code there. Please remember that you can only ssh to the cluster nodes when you are already logged on the front-end server.

After logging into your desired node you need to enable CUDA and pycuda. To enable CUDA and python with pycuda you can run these commands (you can also add them to your $HOME/.bashrc, remember to use python3 and pip3):
`source /share/apps/cuda.sh`
`source /share/apps/python385.sh`
Then you can use this command to verify that the NVIDIA card is working properly and you can check the NVIDIA driver version:
`nvidia-smi`
To check that you have successfully enabled CUDA Toolkit, you can run this command:
`nvcc --version`
Also remember to run `scl enable devtoolset-8 bash` in order to enable gcc8 if you haven't already. Otherwise you will be using an older version of gcc and OpenMP.

You should already have access to the cluster, but if for some reason you have not, please let us know as fast as possible.

```
1 List nodes available right now: /share/apps/ifi/available-nodes.sh
2 List the cluster hardware: /share/apps/ifi/list-cluster-static.sh
3 Command to find the nodes with NVIDIA cards and CUDA toolkit: /
      share/apps/ifi/list-cluster-static.sh | grep -E '(Quadro K|
      Quadro 6|GeForce)'
4 List nodes by load: /share/apps/ifi/list-nodes-by-load.sh
5 List all your processes: /share/apps/ifi/list-cluster-my-processes.
      sh
6 Clean out all your processes: /share/apps/ifi/cleanup.sh
```

Listing 5: uvcluster welcome message

6 Requirements

- Implement and parallelize the three techniques described in Section 2:

  - write code using C/C++ and OpenMP (in `FrostTrap.c`).
  - write code using CUDA C (in `kernel.cu`). Use multiple cores on the GPU.
  - write code in an understandable way and make sure the code is well-commented.

- Analyze your solutions:
  - Analyze your Cuda solution using the Nvidia Visual Profiler [4]
  - Compare these three techniques implemented in Cuda - how well does each of these techniques scale with regard to problem size, number of threads and number of cores? You need to run some experiments in order to answer this question.
  - Try to find an explanation for any observed differences between the methods. For a slightly more advanced approach, you can *optionally* try to form this like a hypothesis that you can test with further experiments.
  - Profile and trace your OpenMP solutions and compare them to the Cuda solutions. Observe differences in performance and form hypotheses of why based on profiling and tracing results.

- Write report:
  - Required sections for the report are as follow (section names might be slightly different):
    * `Introduction` - describe your task.
    * `Parallel solution analysis` - profiling, speedup, expectations, assumptions.
    * `Design` - description of your parallelized techniques - your approach to parallelization (Have you adequately explained your solution?).
    * `Experiments` - describe your experiments with different problem size - different input parameters - different number of CUDA cores - different number of CPU threads for OpenMP.
    * `Discussion` - discuss positive and negative sides of your solution. Compare theoretical maximum speedup with your results in details. (Have you critically evaluated your solution? Have you made your assumptions clear and separate from your measurements?).
    * Summary.
  - Did you find any unexpected results or behaviour of the system? Try to investigate and find a possible explanation. This might require tweaking the source code or experiment parameters to test ideas and compare results.

  - Make your figures clear and understandable. Remember about references, captions and axes descriptions.
  - Remember to take into consideration the hardware you are using.

- Extra Credit: This is optional, but we include it if you want an extra challenge.
  - One way to reduce the number of memory transactions is to use some kind of blocking where a region in the array is copied to the shared memory of a CUDA SM. Threads in a thread block can then read and write to the shared memory instead of using the memory bus when computing. Things to consider include: a) communication between regions/thread blocks (do you need it? .. how do you do it?), b) should you run multiple steps internally in a thread block before communicating with neighbours?, c) overlapping regions (search for "ghost points").
  - Any other improvement you can think of.

7 General Hints

- Start solving the OpenMP part of the assignment. Doing this will in all likelihood significantly shorten the total amount of time spent on the assignment.
- When doing sequential analysis of your solutions it is enough to do a sequential analysis of the OpenMP solution and use that theoretical speedup for the CUDA portion as well.
- When benchmarking your solution choose a single computer and write down the hardware specifications for that machine. Doing benchmarking on multiple computers with different specifications is not ideal.

8 Hand-in

GitHub classroom is used as a hand-in platform for the course. You can and probably should commit and push as often as you would like.

Please remember to push all your changes before 29 October 2020 23:59:59. Any changes pushed to your repository after that deadline will not be evaluated.

References

[1] Lawrence Blaise Barney. Openmp. `https://computing.llnl.gov/tutorials/openMP/`, 2019.

[2] The SciPy community. Numpy user guide. https://numpy.org/devdocs/user/quickstart.html, 2019.

[3] NVIDIA Corporation. Cuda c programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2019.

[4] NVIDIA Corporation. Cuda toolkit documentation - profiler. https://docs.nvidia.com/cuda/profiler-users-guide/index.html, 2019.

[5] Brett D. Estrade. 2d heat distribution prediction in parallel. https://www.cct.lsu.edu/~estrabd/bde_2dheat_691_fall2004.pdf, 2004.

[6] Andreas Kloeckner. Pycuda's documentation. https://documen.tician.de/pycuda/, 2008.

[7] Sparsh Mittal. A study of successive over-relaxation (sor) method parallelization over modern hpc languages. https://hal.archives-ouvertes.fr/hal-01103035/document, 2015.

[8] Peter S. Pacheco. *An introduction to parallel programming.* Morgan Kaufmann, 30 Corporate Drive, Burlington, MA, USA, 2011.