CUDA parallelization of a simplified heat transfer model

Course: Massively Parallel Programming on GPUs

Teacher: Donato D'Ambrosio
Master Degree in Computer Science
Department of Mathematics and Computer Science
University of Calabria, Italy

November 14, 2025

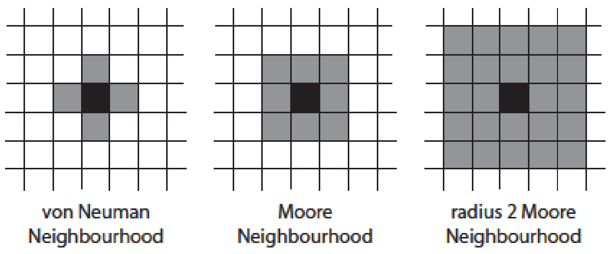

FIGURE 1: Examples of CA neighbourhoods. The first two are the von Neumann and Moore neighbourhoods, both with visibility radius equal to 1. The third is a Moore neighbourhood with visibility radius equal to 2.

1 Introduction to Cellular Automata

The Cellular Automata (CA) computational paradigm is widely adopted in treating complex systems, whose behaviour may be expressed in terms of local laws. Originally introduced by John von Neumann in the 1950s to study self-reproduction issues [2], CA are particularly suited to model and simulate classes of complex systems characterized by a large number of interacting elementary components. The complexity of the system emerges from the interactions of its elementary units (cells), by applying local rules. A CA can be intuitively considered as a d-dimensional space (to which we will refer to as cellular space or domain in the following), partitioned into cells of uniform shape and size. Each cell can be seen as a finite automaton (fa). Input to each fa is given by the state of the cell and the states of the other cells belonging to the cell's neighbourhood. This latter is determined through a geometrical pattern (stencil), which is invariant in space and time. Figure 1 shows well-known examples of two-dimensional neighbourhoods (stencils). At time t = 0, cell states define the CA initial configuration. The CA then evolves step by step by applying the so-called (local) transition function to each cell in parallel.

A CA model can be easily executed on computers. From the computational point of view, the transition function can be applied both sequentially or in a parallel fashion to the domain's cells. In any case, to compute the next state for the cell (step t+1), the transition function needs the state of neighbouring cells at current step (step t). As a consequence, the cell states at step t must be kept in memory during the computation of step t+1. This kind of execution behaviour is referred to as "maximum parallelism". A simple way to address this issue is to use two buffers (for instance, representing matrices) for the space state: the read and the write buffers. During the execution of a generic step, the transition functions are evaluated by reading states from the read buffer and by writing the results to the write buffer. Afterwards, i.e., when the transition function has been evaluated to all the cells, the matrices are swapped and the next computational step can begin. An example of CA execution scheme is provided in Algorithm 1:

Algorithm 1: Cellular Automata execution scheme.

```
init(read_buff,write_buff)
foreach step t do
    foreach cell c do
        n = get_neighbourhood()
        transition_function(c, n)
    swap(read_buff,write_buff)
```

2 Heat transfer CA Model

The goal of the Assignment is to develop a cellular automata model that simulate the unsteady-state conduction of heat in two dimensions[1]. The heat transfer modelled as cellular automaton can be realistically simulated by applying the following simple transition function:

$$T_{i,j}^{t+1} = \frac{4(T_{i,j+1}^t + T_{i,j-1}^t + T_{i+1,j}^t + T_{i-1,j}^t) + T_{i+1,j+1}^t + T_{i+1,j-1}^t + T_{i-1,j+1}^t + T_{i-1,j-1}^t}{20} \tag{1}$$

where $T_{i,j}^t$ represents the temperature in degrees Celsius of the cell at the coordinates (i, j) at the time step t.

Despite its simplicity, the result approaches the solution of the classic PDE-based heat equation for unsteady heat conduction.

3 The Assignment in Brief

The assignment consists in the CUDA parallelization and performance assessment of the above described heat transfer simulation model. A serial reference implementation is provided to be used as a reference starting point for implementation, correctness and performance assessment.

3.1 Compiling the serial application

In order to compile and run the serial application, provided as the starting point for parallel implementations, you can use the following command:

```
g++ heat_transfer_cli.cpp -o heat_transfer_cli
```

To run the simulation, simply type:

```
./heat_transfer_cli
```

[1] https://demonstrations.wolfram.com/ACellularAutomatonBasedHeatEquation

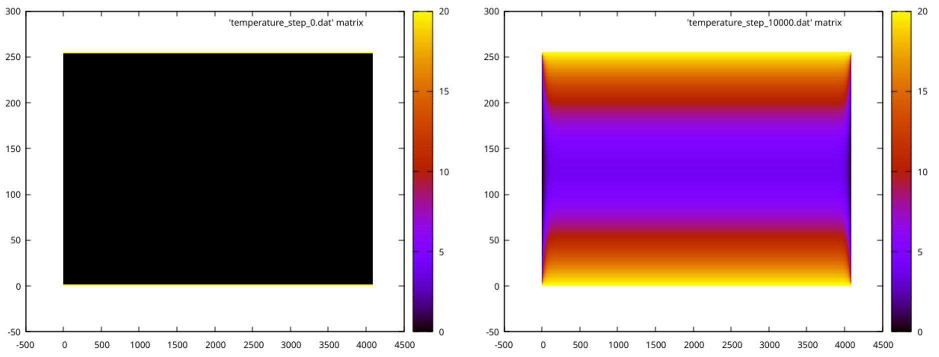

FIGURE 2: Example of heat transfer dynamics. On the left and right are the initial and final confugurations of the system, respectively. The initial configuration is 20 Celsius degrees at the first and last 2 rows and 0 Celsius degrees elsewhere. Boundary conditions remain stationary during the simulation (i.e., the first and last 2 rows and the first and last columns remain unchanged).

The initial configuration and the final outcome of the simulation are saved to files named `temperature_step_0.dat` and `temperature_step_10000.dat`, respectively. To visualize them, e.g. `temperature_step_10000.dat`, run gnuplot and type the following command:

```
plot 'temperature_step_10000.dat' matrix with image
```

Type `quit` to exit gnuplot. See Figure 2 for an example of gnuplot visualization.

3.2 Formal definition of the heat transfer Cellular Automata model

$$\text{HeaT} =< R, X, S, \sigma >$$

where R is the two-dimensional computational domain, subdivided in square cells of uniform size, while X is the Moore neighborhood with visibility radius 1.
S is the set of cell states, representing the cell's temperature.
$\sigma : S^5 \to S$ is the deterministic cell transition function. It is defined by the Equation 1.

3.3 Specifications of the simulation

The cellular space is a grid of square cells of 2^8 rows x 2^{12} columns.

The initial configuration of the system is 20 Celsius degrees at the first and last 2 rows (boundary conditions) and 0 Celsius degrees elsewhere. Boundary conditions remain stationary during the simulation (i.e., the first and last 2 rows and the first and last columns remain unchanged). The simulation steps to be computed are 10.000.

Block configuration
8x8
8x16
8x32
16x8
16x16
16x32
32x8
32x16
32x32

Table 1: Block configurations to be used for the performance assessment task.

4 The Assignment

The goal is to develop several parallel implementations of the heat transfer model, assess their performance and warp occupancy, and apply the Roofline analysis to the developed kernels to reveal their nature, i.e., if they are memory or compute bound with respect to the GPU considered.

4.1 Development

The CUDA parallel implementations to be developed are listed below, each of them using the CUDA unified memory model (namely the `cudaMallocManaged()` API functin for memory allocation):

1. `Global`, namely a straightforward parallelization using the CUDA Unified Memory;
2. `Tiled`, namely a shared memory based tiled parallelization without halo cells;
3. `Tiled_wH`, namely a tiled parallelization in which block's boundary threads perform extra work to copy halo cells from adjacent tiles in the shared memory;

4.2 Performance Assessment

For each implementation and for each block configuration in table 1, run four experiments and plot the best elapsed time registered. Use a bar plot chart for this purpose, by paying attention to label the axes and to write a meaningful caption. An example is shown in Figure 3. Reporting the elapsed times on top of each bars is optional but recommended.

In addition, report the elapsed times registered by the best (implementation, block configuration) couples in a table. Furthermore, perform a

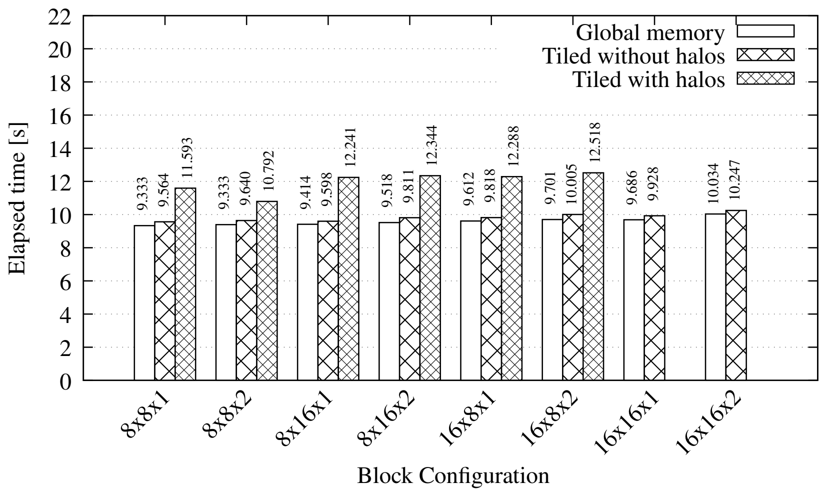

FIGURE 3: Examples of bar plot chart.

profiling analysis of the best performing (implementation, block configuration) executions to assess the warp occupancy of each of them. Report the results in a new column of the same table used for the elapsed times.

Note that the experiments must be executed on the GTX 980 GPU for comparison.

4.3 Applying the Roofline Model

Define the Roofline plot for the GTX 980 GPU, for each best (implementation, block configuration) couple. You can either use the ERT (Empirical Roofline Toolkit) [3] or the *gpumembench* [1] micro-benchmark[2]. Then, for each of couple, evaluate the Operational Intensity of the transition function kernel, and assess its performance by means of `nvprof`. Then, put a point for the kernel in the Roofline plot and briefly comment the kernel's nature, if compute or memory bound. An example of Roofline chart is shown in Figure 4.

References

[1] Elias Konstantinidis and Yiannis Cotronis. A quantitative performance evaluation of fast on-chip memories of gpus. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 448–455, 2016.

[2]Note that you need to change the NVCODE macro in the CUDA makefiles in: `NVCODE = -gencode=arch=compute_52,code="compute_52" -ftz=true` in order to generate the binaries for the compute architecture 5.2 only.

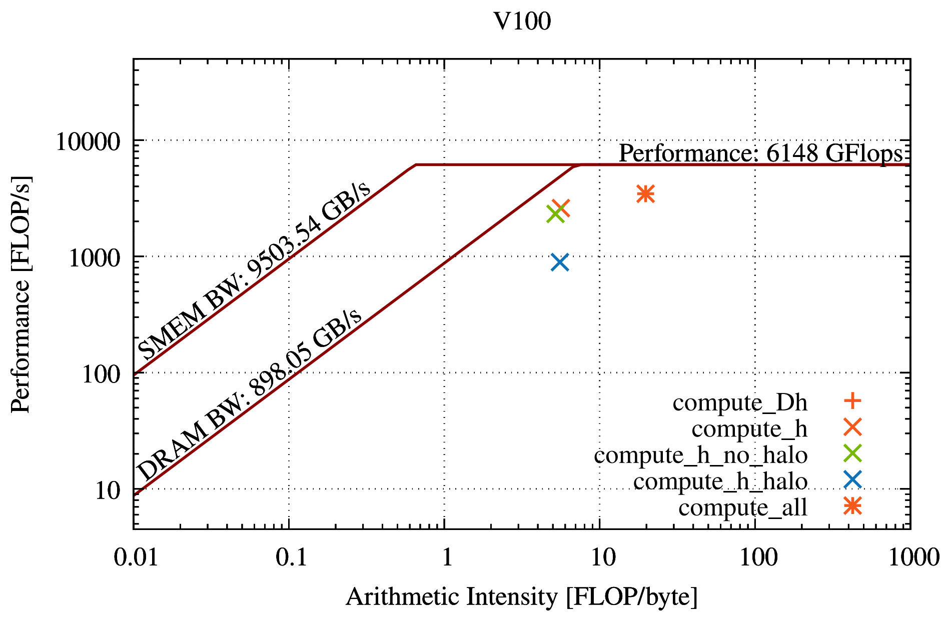

FIGURE 4: Examples of Roofline chart.

[2] John von Neumann. *Theory of Self-Reproducing Automata.* University of Illinois Press, Champaign, IL, USA, 1966.

[3] Charlene Yang, Thorsten Kurth, and Samuel Williams. Hierarchical roofline analysis for gpus: Accelerating performance optimization for the nersc-9 perlmutter system. *Concurrency and Computation: Practice and Experience*, 32(20):e5547, 2020.