ISING MODEL

—and—

CUDA

Γιαχούδης Χρίστος ~ 9912

Περιεχόμενα:

1. The problem

The Ising model, named after the physicist Ernst Ising, is a mathematical model of ferromagnetism in statistical mechanics. The model consists of discrete magnetic dipole moments of atomic "spins" that can be in one of two states (+1 or −1). The spins are arranged in a square 2D lattice with periodic boundary conditions, allowing each spin to interact with its four immediate neighbors. The dipole moments update in discrete time steps according to the majority of the spins among the four neighbors of each lattice point. The edge lattice points wrap around to the other side (known as toroidal or periodic boundary conditions), i.e. if side length is n, and grid coordinates are 0:n−1, then the previous to node 0 is node n−1 and vice versa.

The project consists of 4 parts. The first one is a sequential implementation of the Ising model using C. The rest are variations of the first one, but they involve CUDA coding and GPU usage.

Other than the model implementation, the substantial purpose of this project is to learn to use the GPU with CUDA.

2. The approach

sign(G[i-1,j] + G[i,j-1] + G[i,j] + G[i+1,j] + G[i,j+1])

The formula above shows exatly what we have to do, to compute the spin of each magnetic dipole moment. G is the 2D square lattice of size nxn. We sum up the spins of the four neighbors and add them to the moment's spin. If it is positive then the new spin is equal to 1. Else if the sum is negative the new spin is equal to -1. The only problem is that the dipole moments at the edges have less that 4 neighbors. So to create the toroidal form, that the theory demands, we use the modulo function. For example:

... + G[i][(j+1)%n] + ...

Well, of course there will be a problem. The % function does not work properly with negative numbers. So we come up with the following custom made function that computes the modulo of negative numbers:

```
// modulo implementation for negative numbers
int mod(int a, int b){ // b must be always positive
    int r = a % b;
    return r < 0 ? r + b : r;
}
```

So we have:

... + G[i][mod(j-1, n)] + ...

After that we just parallelize the computations, by assigning the computation for each dipole mpment to a GPU thread. A small explanation for each code version follows...

3. About the code

Note that if you want to change the parameters, you need to change the code. The program does not get arguments at the execution command.

Version 0 is the sequential implementation of the solution. The only thing worths mentioning is that, in order to create random numbers between -1 and 1, we produce randomly 0s and 1s. Then if the random number is 0 we use -1 instead. The function that computes the actual spins is implemented into seq_Ising.c. The function that contains is imported and called to main.

Version 1.1 is the first parallel implementation that uses CUDA. The modulo function is now a __device__ function, so that it can be used only by the gpu's threads. The actual implementation exists into the __global__ function Ising. The procedure goes like this:

1. Define the parameters(2D lattice size n, number of blocks m)
2. We copy the lattices(G1 and G2) to the global gpu memory using cudaMemcpy(CUDAG1, G1, n*n*sizeof(int), cudaMemcpyHostToDevice);
3. We call the Ising function so that it does the computations. It reads the data from the older lattice and stores the new ones at the newer lattice.
4. Then the 2 lattices are copied back to the cpu memory using cudaMemcpy(G1, CUDAG1, n*n*sizeof(int), cudaMemcpyDeviceToHost);
5. We swap the 2 lattices.
6. We follow the procedure above for k iterations.
7. The 1.1 version uses only one thread per block. Each one of these threads(so each block too) computes only one spin.

Version 1.2 does exactly the same thing as version 1.1, but now each block contains a number of threads(variable t). That means that each block computes a number of spins.

Version 2 does again the same work, but the difference here is that each thread from each block computes more than one spins. That is handled by the variable w. Each thread computes w*w spins

Version 3 is a little different. Now each block has each own memory and it's called shared memory. The threads of the block read from it and write to it faster than the global one. That's why the first thread of each block copies the spins, that the block(its threads) is goint to compute, from the global memory to the shared one. It copies them to a 1d lattice called lattice. Also it stores the neighbours of the spins that are to the edges, so that it doesn't need to read them from global. That's why the lattice has a size of (t*w+2)*(t*w+2). We add the number 2 because we want the egdes of this lattice to be the neighbours. We compute the spins and store them to the newer array(global memory). !!!The code of version 3 doesn't work and, thus, we do not have measurements.

4. Compiling and execution

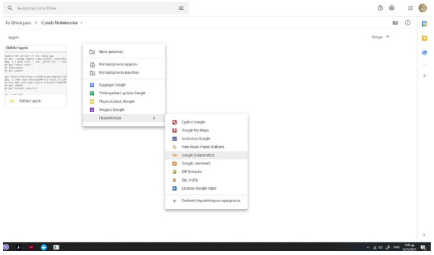
- For the sequential algorithm:

1. Copy the following files into the same folder: main.c, seq_Ising.c, seq_Ising.h.
2. Open a terminal in Linux (I worked on the terminal of a docker).
3. Change directory using the cd command and navigate to the folder that the files are stored.
4. Type the followings to compile:
   -> gcc -c seq_Ising.c
   -> gcc -c main.c
   -> gcc seq_Ising.o main.o -o main.exe
5. Type the following to execute the code:
   -> ./main.exe

- For the other versions(the cuda ones):

The files that contain the code are “filename.cu“ types. The easiest way to run that code(and that we did) is to use the googlecolab platform, a python notebook.
The steps that you have to follow to run the code are:

1. Open your browser and type googlecolab or open your google drive, right click, and create a new google-collab document.
2. Into the document copy the followings(just create a new cell for each part of the code):

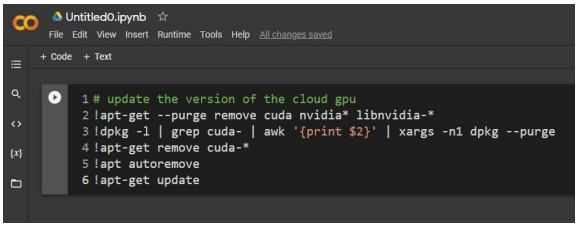


1st cell:

```
!apt-get --purge remove cuda nvidia* libnvidia-*
!dpkg -l | grep cuda- | awk ‘{print $2}’ | xargs -n1 dpkg --purge
!apt-get remove cuda-*
!apt autoremove
!apt-get update
```

2nd cell:

```
!wget https://developer.nvidia.com/compute/cuda/9.2/Prod/local_installers/cuda-repo-ubuntu1604-9-2-local_9.2.88-1_amd64 -O cuda-repo-ubuntu1604-9-2-local_9.2.88-1_amd64.deb
!dpkg -i cuda-repo-ubuntu1604-9-2-local_9.2.88-1_amd64.deb
!apt-key add /var/cuda-repo-9-2-local/7fa2af80.pub
!apt-get update
!apt-get install cuda-9.2
```



3rd cell:

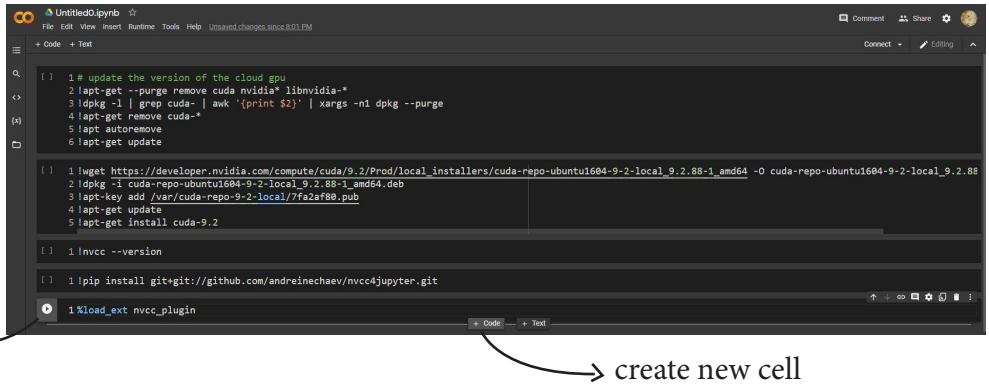
```
!nvcc --version
```

4th cell:

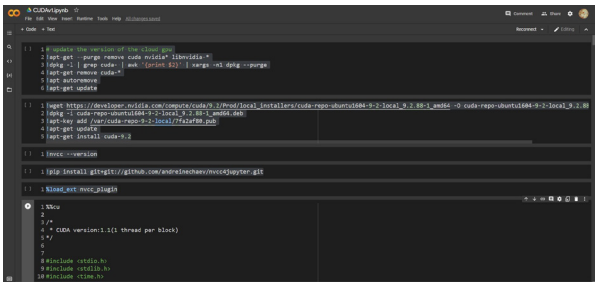
```
!pip install git+git://github.com/andreinechaev/nvcc4jupyter.git
```

5th cell:

```
%load_ext nvcc_plugin
```



3. Create a new cell, copy the code from the “.cu“ file you want to test and paste it to the new empty cell.
4. To run the code firstly run each cell in order except the actual implementation. You do that by clicking the run button. You do not need to run the first 5 cells every time you want to run the code. Only every time you open the googlecolab file. This procedure is needed so that we can import c code in a python jupiter and run it.
5. Click the run button of the cell that contains the implementation and scroll to the bottom to see the results.



5. The results

The sizes of the lattices, we tested are small, because the googlecolab platform could not handle a big number of data. Also, the version 3 does not work, so we do not have measurements for it. The results we retrieved are shown here:

The time measurements

code version / n*n(size of lattice)	V0	V1.1	V1.2	V2
8*8	0.000012 sec	0.000065 sec (64 blocks, 1 thread per block)	0.000112 sec (25 blocks, 16 threads per block)	0.000067 sec (4 blocks, 4 threads per block, 4 computations per thread)
20*20	0.000071 sec	0.000352 sec (400 blocks, 1 thread per block)	0.00038 sec (25 blocks, 16 threads per block)	0.000352 sec (25 blocks, 4 threads per block, 4 computations per thread)
50*50	0.000504 sec	0.002053 sec (2500 blocks, 1 thread per block)	0.002069 sec (25 blocks, 100 threads per block)	0.003992 sec (25 blocks, 25 threads per block, 4 computations per thread)
100*100	0.001909 sec	0.008306 sec (10000 blocks, 1 thread per block)	0.008259 sec (100 blocks, 100 threads per block)	0.008267 sec (100 blocks, 25 threads per block, 4 computations per thread)

Some observations:

- The sequential algorithm is faster for small datasets. It seems, though, that for bigger datasets the parallel implementations do a really good jod, something that could be established, if we could do some more tests.
- Something similar happens with the threads. V1 with smaller datasets is faster than V2. But for bigger datasets the times get close.
- Also, it seems that using more blocks makes the procedure faster.
- At the end, though the last implementation didn't work, it is assumed that we do multiple readings from the shared memory(faster reading/writing speeds) instead of the global one and that reduces the time of the procedure.