Parallel and Distributed Systems-Assignment 3 CUDA-Ising Model

- **Authors:**
  Moustaklis Apostolos AEM:9127 amoustakl@auth.gr
  Papadakis Charalampos AEM:9128 papadakic@auth.gr

- Source code can be found in the link below:
  https://github.com/tolism/Ising

- **Short description of our implementation:**

  1) **V0-Sequential:** The sequential implementation of the program,implements the evolution of the Ising model including the modifications that were mentioned. For the implementation we used the template of the function ising() that was included in the online tester. Specifically, in our ising() function, we have two arrays with the old and the current spin lattices. We calculate the weighted influence of each point, according to the current state of its neighbors, so that we can predict if the state of this specific point will stay the same or will change sign. We keep the total current state we calculated. After that, we swap the pointers for the next iteration ,as we were asked to. Because of the floating errors that occur , we have used a really small value (10e-07) instead of zero, as we check the influence's "behavior".

  2) **V1:** In the first part of the CUDA implementation we were asked to assign to each thread the calculation of the spin of only one point. We modified our sequential code ,doing all the necessary memory allocations and the "copies-transfers" between the CPU and GPU memory, so that we take advantage of the GPU's speed. We,now, have the function kernel2D, which is called as a kernel with a grid of threads,with the same dimensions as the lattice's (gridDim.x*blockDim*x>=n κ gridDim.y*blockDim*y>=n). With the call of that function, we provide that every thread will compute the spin of one point using the GPU.
  3) **V2:** In the second part of the CUDA implementation , the kernel2D function is modified to make each thread to compute the spin of a block of points. The distance between two points of the blocks in X and Y axis , is strideX and strideY,respectively. StrideX and strideY are

calculated by the kernel2D function and depend on the parameters of the grid. We define the size of the grid and the block in the start of our code.

4) **V3:** The third and last part of the CUDA implementation , is based on the V2,although it is modified so that each thread of a block, to compute the spin of the point that is “responsible” for, will not need to “read” information from the main memory of the GPU, but only from the needed points that are moved to the shared memory of the block. The transfer of the points in the shared memory has been performed by the threads of the block. Our goal is to have faster performance.

● Execution time results and graphs

I. The specifications of the system that was used:

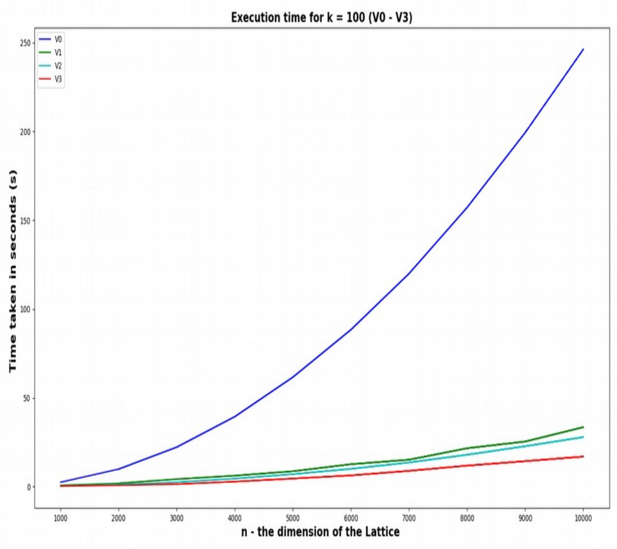
The implementation and results were performed in a laptop with an **Intel® Core™ i7-7700HQ CPU @ 2.80GHz × 8** processor and **8GB RAM.** The Graphic Card (GPU) of the device is **GeForce GTX 1050.**
We have used the latest version of CUDA (10.2), and GPU has **5 Multiprocessors** with **128 CUDA threads each**. The **maximum number of threads “running” per MultiProcessor** is **2048**.

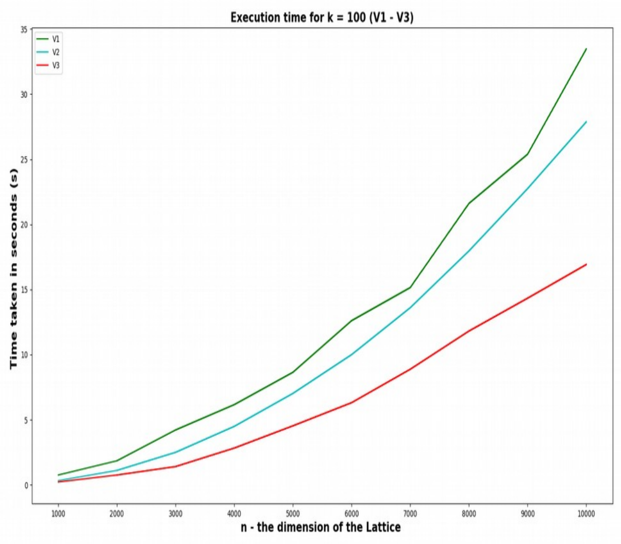
II. Choosing threads per block and grid:

Taking in consideration the fact that the wrap size is 32 and after benchmarking we ended up choosing a block with dimensions **16x16.** As for the grid, we followed the NVIDIA site formula **gridSize = (N + blockSize - 1) / blockSize** which adjusts the gridSize to N (dimension of the array).

Execution time results tables and graphs

A) Results with constant k and varying n:

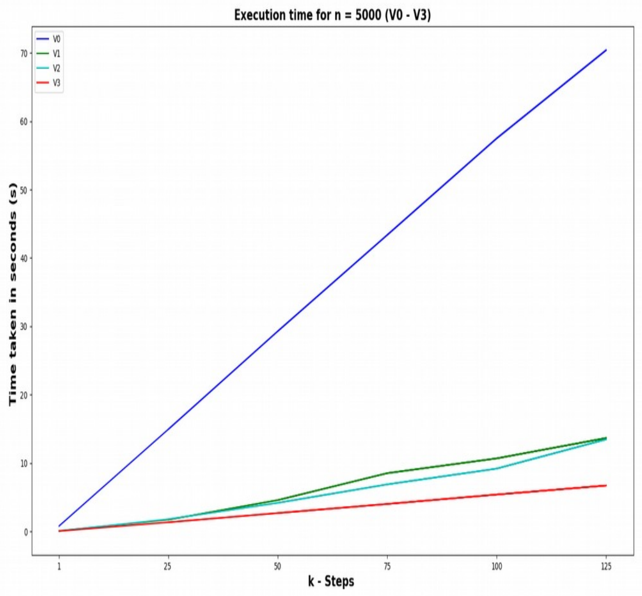


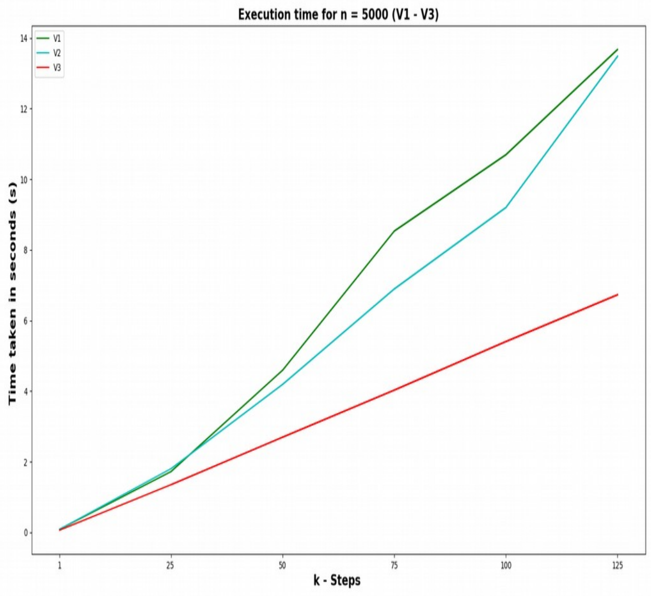


n	k	V1	V2	V3	V0
1000	50	0.168697	0.173563	0.108270	1.356768
2000	50	0.552297	0.558308	0.381066	5.470742
3000	50	1.224196	1.326230	0.855628	12.123081
4000	50	2.220162	2.604449	1.516052	21.718499
5000	50	3.476432	3.873485	2.369527	33.798925
6000	50	6.164669	6.904914	3.417228	48.820715
7000	50	9.424111	8.598156	4.646367	66.324201
8000	50	11.322116	8.927300	6.069370	86.614458
9000	50	19.187438	11.370383	7.680302	110.24394
10000	50	23.109088	13.884807	9.473470	139.03927

n	k	V1	V2	V3	V0
1000	100	0.759972	0.324305	0.224733	2.457079
2000	100	1.852972	1.112558	0.759315	9.851103
3000	100	4.219177	2.501977	1.407933	22.207692
4000	100	6.163287	4.493912	2.827959	39.388650
5000	100	8.645404	7.024960	4.535924	61.626034
6000	100	12.610303	10.001017	6.313410	88.369274
7000	100	15.151311	13.604959	8.879699	119.95826
8000	100	21.607439	17.959872	11.812002	157.19734
9000	100	25.388354	22.747589	14.335195	199.37339
10000	100	33.462701	27.869173	16.918780	246.24206

B) Results with constant n and varying k:





n	k	V1	V2	V3	V0
5000	1	0.087822	0.088799	0.065288	0.797618
5000	25	1.724643	1.805451	1.353295	14.962656
5000	50	4.595846	4.193700	2.698232	29.303439
5000	75	8.532982	6.894486	4.029829	43.371945
5000	100	10.693339	9.199043	5.404677	57.492804
5000	125	13.671451	13.478343	6.728139	70.392742

Observation: It is quite clear that as the load of the problem (k and n) increases the parallelism is getting even more efficient and fast. To see even more difference we could use greater n and k.