Parallel and Distributed Systems – Exercise 3
CUDA – ISING MODEL

Authors:

Christoforidis Savvas, AEM: 9147, email: schristofo@ece.auth.gr

Portokalidis Stavros, AEM: 9334, email: stavport@ece.auth.gr

Code: https://github.com/schristofo/CUDA-ising-model

v0. Sequential: The first version of our program, basically, implements the *ising()* function given to us by the online tester. Analytically, using two arrays as requested we copy the data from one to the other, so as to use the one of them as an input and the other one as the output. This is achieved by swapping their pointers after each calculation of the new spin values. There is also a check in case of having the spin lattice unchanged using a boolean variable. More deeply, the calculation of new spin values consists of the calculation of the influence and the sign function of each influence. The influence calculation uses mod arithmetic to avoid if statements for the boundaries. Finally, in order to take into account possible floating point errors and check if the influence value is zero, we use a threshold value(10^-7).

v1. GPU with one thread per moment

In this part of the exercise, we were asked to modify the sequential code (v0),make a kernel function and call the kernel with a grid that matches the Ising model and one thread per moment. First, we transfer the data to GPU memory. After that, we define **<< c = blockIdx.x * blockDim.x + threadIdx.x , r = blockIdx.y * blockDim.y + threadIdx.y; >>** to caclulate the weighted influence of each point's neighbors. Finaly, we compute the spin of each point, according to Ising model method. Every GPU's thread will copute the spin of one point.

v2. GPU with one thread computing a block of moments

This version of the method is slightly different from the previous one, due to the fact that each thread is computing a block of moments, increasing by grid value each time. Hence, we use a grid-stride loop by moving the block gridDim.x*blockDim.x in terms of X and gridDim.y*blockDim.y in terms of Y. In this way, we are able to make good use of every utilizable thread.

v3. GPU with multiple thread sharing common input moments

In the last version of the function *ising()* we modified version v2 by adding two attached-to-device memory arrays for faster access. In other words we used two shared memory blocks, one for storing the w matrix and one for storing a block of moments and its radius, accesible from every thread in each block. The result was amaizing, since time performance improved drastically.

Time Performance and Visualization:

System Characterisitcs / GPU model:

Intel® Core™ i7-7500U CPU @ 2.70GHz × 4 / NVIDIA GK208 (GeForce 920M)

CUDA Version 10.2.89

Optimal block size used for testing: **16 x 16**

Results:

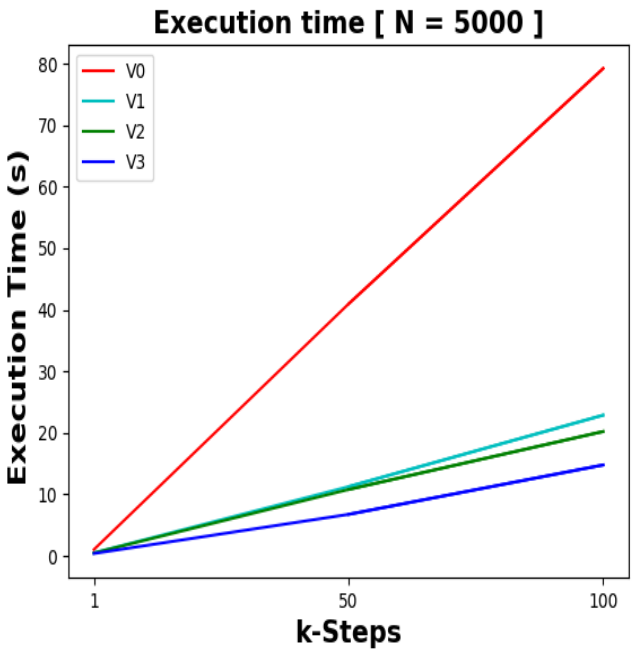
A) Results with constant n

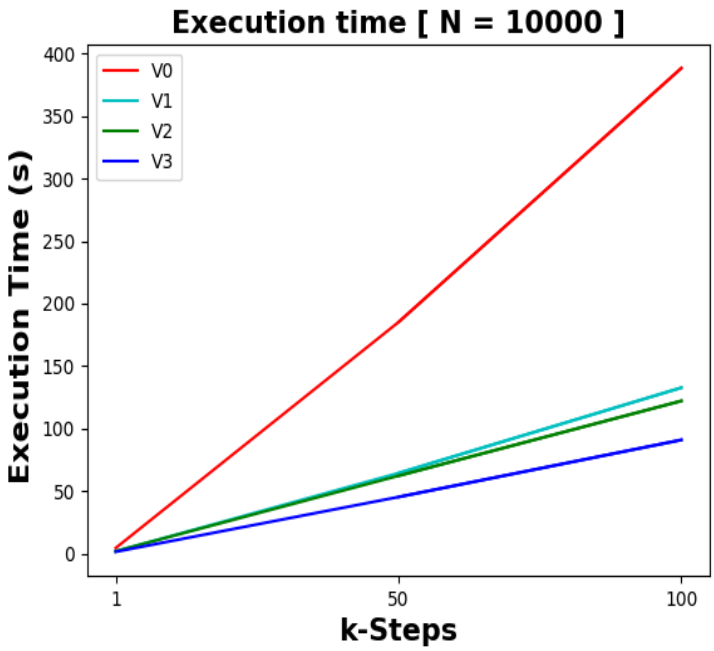
n	k	V0	V1	V2	V3
1000	1	0.070918	0.062	0.059	0.053
1000	50	1.448755	0.751	0.728	0.550
1000	100	2.896730	1.445	1.395	1.063

n	k	V0	V1	V2	V3
5000	1	1.02888	0.503	0.475	0.394
5000	50	35.714700	11.264	10.797	6.753
5000	100	79.232514	22.883	20.225	14.787

n	k	V0	V1	V2	V3
10000	1	4.457116	1.812	1.938	1.459
10000	50	155.13	60.418	58.262	39.293
10000	100	388.272135	132.711	122.130	90.952

Graphs:



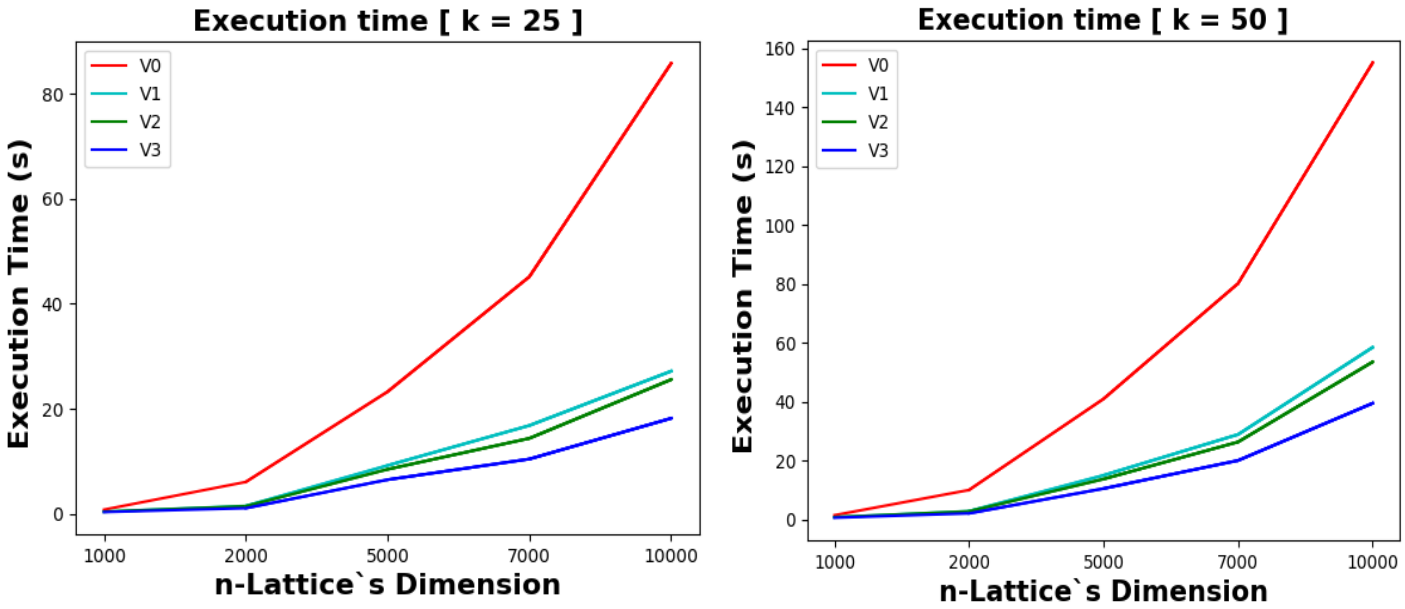


B) Results with constant k

n	k	V0	V1	V2	V3
1000	25	0.763684	0.389	0.3844	0.310
2000	25	6.044902	1.476	1.426	1.049
5000	25	23.2494	9.193	8.451	6.477
7000	25	45.163928	16.805	14.361	10.415
10000	25	85.8424	27.165	25.561	18.182

n	k	V0	V1	V2	V3
1000	50	1.422374	0.755	0.745	0.572
2000	50	9.992142	2.787	2.747	2.055
5000	50	35.9537	11.996	10.713	6.485
7000	50	70.163928	28.865	26.319	20.033
10000	50	155.12897	60.460	58.518	39.469

Graphs:



Easily, we can observe that execution time grows linearly for constant n and growing k. On the other hand, when we keep k constant and we increase n there is an expondential growth in execution time. Finally, as far as the version execution times are concerned, we see that tv0 > tv1 > tv2 > tv3.