Parallel and Distributed System
Ising Model Evolution Using CUDA

Koutroumpis Georgios, AEM: 9668

2022

1 Introduction

The purpose of this program is to simulate the evolution of a nxn Ising Model for k steps, using CUDA. Each moment in the nxn lattice has a spin of -1 or 1. In each iteration, its new spin is determined by the sign of the sum of the spins of the 4 immediate orthogonal neighbors and itself. If the moment resides at the edge, then it wraps around, considering the moments in the other sides its neighbors. For example, if the moment has coordinates (0,0), its top neighbor would have coordinates (n-1,0) and its left neighbor (0,n-1).
The data used for this project is randomly generated uniformly.

2 Implementations

2.1 v0 - Sequential

In this sequential implementation, each moment's new spin is calculated sequentially for each step. The new spin of the moment is the sign of the sum of the moment's neighbors' spins and its own. To avoid if statements, the mathematical modulo operation is implemented, as

$$(x + n)\%n$$

so when neighbors of edge moments are needed, the indices of the neighbors wrap around the matrix as needed.

2.2 v1 - GPU with one thread per moment

In this implementation, the GPU is utilized. A 2D CUDA grid of 2D thread blocks is created, with each GPU thread being responsible for calculating a specific moment. The thread index to moment index mapping is done with the help of the built-in CUDA variables for thread and block indices. Each block has (thread_width)*(thread_width) threads, with thread_width being predefined. The width of the grid is defined as *(n + thread_width - 1)/thread_width*. The width is padded so there are always enough threads to take on one moment each. (So there is the case that threads that have no work to do are spawned)

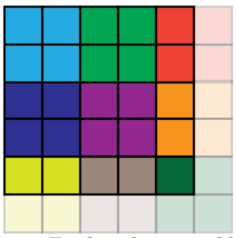

Figure 1: *v1 eg for 5x5 matrix. Each color is a block, having 2x2 threads, and each square is a thread / moment.*

2.3 v2 - GPU with one thread computing a block of moments

In this implementation, each thread is tasked with computing a bxb block of moments (not to be confused with the block of threads!). This decreases the number of (thread) blocks needed, as the width of the grid is now defined as *(n + thread_width*b - 1)/thread_width*b*. Of course, padding is still needed in case that n divided by thread_width*b wouldn't give an integer result. The optimal b was found to be 3, after tests for different thread width, n and k values. (Part of the tests showed in Figure 4 in the next section)

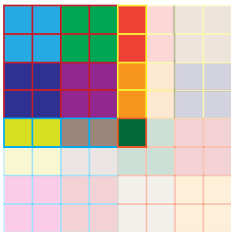

Figure 2: *v2 eg for 5x5 matrix. Each color is a thread computing 2x2 moments, each square is a moment and the grid lines with the same color define a block having 2x2 threads. The grid has 2x2 blocks.*

2.4 v3 - GPU with multiple thread sharing common input moments

This version optimizes v2. In v2, there are many identical/overlapping accesses to the spin matrix. Since global memory access is costly, we can reduce the runtime, by first loading the part of the matrix that each block computes to the shared memory. To do that, a shared memory matrix of width (thread_width*b + 2) is created, which holds the spins' of all the moments that are to be computed from this specific block, in addition to the edge neighbors of these moments. Each thread is responsible for loading the moments of its moment-block to the shared memory matrix. In addition, if the moment is at the edge of the **thread-block**, the thread also loads the moment's edge neighbors to the shared memory matrix. The parameter b is bound by the size of the shared memory. It should always hold that

$$(thread\_width * b + 2) * (thread\_width * b + 2) * sizeof(int) < 48kB$$

for most GPUs. The optimal b was found to be 4, after tests for different thread width, n and k values (Part of the tests showed in Figure 4 in the next section). The value b=4 was kept for both v2 and v3 for consistency.

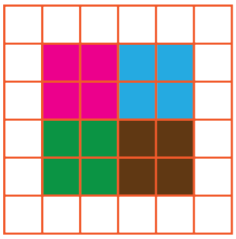

Figure 3: *v3 eg of the shared memory. For a block with 2x2 threads, and each thread calculating 2x2 moments, the shared memory for this block is (5+2)x(5+2)*

3 Plots & Tables

THREAD_WIDTH	b	n	k	time (ms)	version	THREAD_WIDTH	b	n	k	time (ms)	version
8	1	10000	10	29504	0	12	1	10000	10	29788	0
8	1	10000	10	117.480476	1	12	1	10000	10	136.2566	1
8	1	10000	10	118.057983	2	12	1	10000	10	131.8492	2
8	1	10000	10	151.605988	3	12	1	10000	10	159.9068	3
8	2	10000	10	112.13475	2	12	2	10000	10	131.7687	2
8	2	10000	10	99.637283	3	12	2	10000	10	106.1586	3
8	3	10000	10	112.132317	2	12	3	10000	10	124.9247	2
8	3	10000	10	82.547714	3	12	3	10000	10	87.68442	3
8	4	10000	10	121.43306	2	12	4	10000	10	127.1931	2
8	4	10000	10	75.108734	3	12	4	10000	10	80.16218	3
8	5	10000	10	270.633118	2	12	5	10000	10	208.9495	2
8	5	10000	10	87.34848	3	12	5	10000	10	87.05402	3
10	1	10000	10	29710	0	16	1	10000	10	44920	0
10	1	10000	10	148.259613	1	16	1	10000	10	121.1127	1
10	1	10000	10	149.313797	2	16	1	10000	10	129.7428	2
10	1	10000	10	179.686401	3	16	1	10000	10	146.4998	3
10	2	10000	10	142.431229	2	16	2	10000	10	114.0237	2
10	2	10000	10	135.486465	3	16	2	10000	10	103.8048	3
10	3	10000	10	141.106491	2	16	3	10000	10	122.2875	2
10	3	10000	10	99.773758	3	16	3	10000	10	90.74381	3
10	4	10000	10	142.081497	2	16	4	10000	10	128.5728	2
10	4	10000	10	90.227715	3	16	4	10000	10	83.94445	3
10	5	10000	10	173.964508	2	16	5	10000	10	322.5846	2
10	5	10000	10	93.907486	3	16	5	10000	10	101.8696	3

Figure 4: *Runtimes for different values of b, for set n, k. b=3, b=4 are optimal for v2 and v3 respectively*

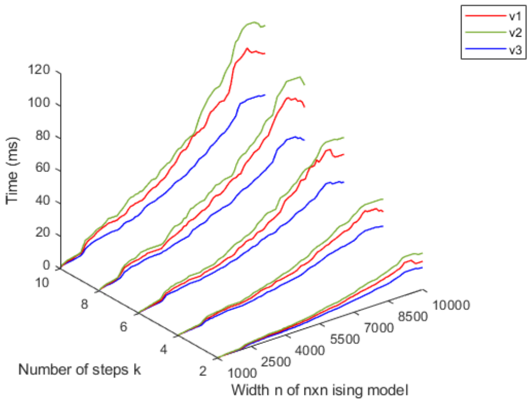

Figure 5: *Runtimes of v1,v2,v3 for different n, k*

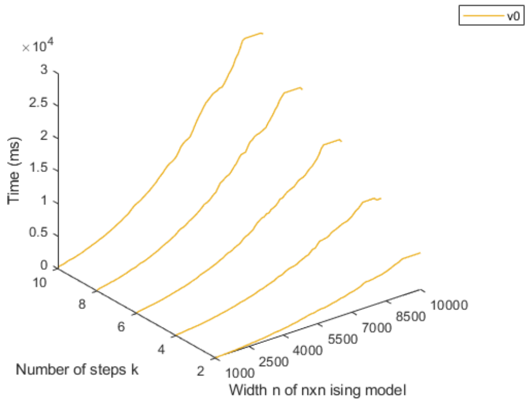

Figure 6: *Runtimes of v0 for different n, k*

All tests were run locally using an NVIDIA GeForce GTX 1070 GPU. As shown, v3 is always the fastest for large n values, with v1, v2, and v0 following.

GitHub Link
Alternate Google Drive link