Ising Model: Cellular Automata CUDA Report

Evangelos Kyriakos

January 28, 2024

1 Introduction

In this report, we present the results of paralleling the Ising model simulation using CUDA. We compare the performance of different architecture implementations for various problem sizes. The Ising model is a proposed schema which simulates ferromagnetic properties, usually found in statistical mechanics.

The schema consist of two dipole quantum moments of particle 'spins' than contain two states either 1 or -1. The model is arranged on a 2D lattice with periodic bound conditions, meaning that the lattice represents any infinite repeating patterns along a two step modular surface.

According to the Ising Model, each lattice point dipole moment is updated according to the majority of the spin among the four neighboring sides. The wrap of the lattice results in a toroidal coordinate from of 0 to $n-1$ where n the bound length of the lattice.

The closed dipole ferromagnetic moment change can be modeled in the form of a Hamiltonian over a sum over the graph edged $E(G)$ as:

$$H(\sigma) = -J \sum_{<i,j>} \sigma_i \sigma_j$$

or in the context of a fixed functional iterative rule:

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

Here we experiment with different configurations in a NVIDIA GPU in order to map the model closer to reality as possible. We will make use the GPU extended parallel power along its cores as well as its physical topology in order to reflect the concurrent behavior of physical processes on real time.

2 GPU Architecture investigation

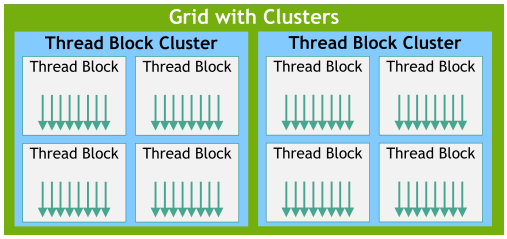

Figure 1: Grid of Thread Block Clusters

In the context of GPU programming we will utilize three essential concepts used to organize and execute parallel computations efficiently; Threads, Blocks and the Grid.

Threads just like typical CPUs are the smallest unit of execution in a GPU. Thousands of concurrent threads allows massive parallelism. Threads are organized into blocks. Blocks are groups of threads that themselves are organized into a grid. The blocks of a GPU provide a way to partition the overall computation into smaller manageable units that can be executed in parallel. Threads withing a block cooperate and synchronize using shared memory between them.

Grids unlike blocks, can be broke down to collections of blocks that execute independently from each other.

In this assignment we are called to investigate various implementation strategies in the overall attempt of creating a faithful Ising model. The parallel capacities of a GPU are especially valuable in our case since

it provides legit mapping of our modeling problem on both a physical case in case of the GPU topology, and more thoroughly in its powers to keep up with the changes of the model on real time. Essentially, since our model consist of numerous processes interacting with themselves simultaneously, common computational methods, in the case of a sequential CPU core, are met with a bottleneck due to the volume of on time computational threads.

Here we are meant to try the three following parallel strategies to handle the problem; Using one threads per moment, i.e. spreading across the grid only one thread per moment, the use of one thread computing a block of moments and to investigate an optimal block size. Finally we are called to create multiple threads sharing common input moments in our blocks.

3 Code Design

We conducted the given experiment, not on the provided HPC grid 'Aristotelis', but on a local computer equipped with an NVIDIA RTX 4070 GPU with 12 GB GDDR6X VRAM, a 16 core i7-10700K CPU and 16 GB of RAM. RTX 4070 has 5888 CUDA cores. The Ising model simulation was implemented in CUDA, and we measured the execution time for different problem sizes and over a different number of iterations

3.1 V0. Sequential Design

The fist attempt to simulate the Ising model was made on a sequential C script. The main body of the design which inspires the following CUDA scripts derives from here. The C script itself works as follows; from a given seed (that the reader is encouraged to change), and two integer arguments in the main function (or three), of n, the size of the Ising square grid, and k the number of iterations to follow, a randomly generated grid of nc of 1 or -1 values is distributed across the lattice. Then with the appropriate memory being allocated on the host memory, the update step mentioned in the introduction is iterated a number of k times. The update function since it is called to read all four neighboring sides of each point, for border cases, makes use of the torodial nature of the lattice by making the appropriate jumps across opposite point to be read, with no if statements involved. Thus, The newly generated grid is renewed from the new one with a grid pointer. The timing of the whole processes is also displayed.

3.2 V1 GPU with one thread per moment

Here the design structure is overall the same with exception of the conversion of the `__global__` update function and the changes between host and device memory allocation. More clearly, two allocations are made in the device memory of the two interchangeable grid states and only one from the host memory to carry the random initial and the final state back to the host. The device itself is distributed over its grid of n^2 one dimensional blocks that make use of only one thread. Thus every point on the Ising lattice is allocated a single thread to compute and update its future state, thus making use of n^2 separate threads.

We can tell that this might not be the most computationally efficient way to approach the problem, which is further derived on the following submissions.

3.3 V2 GPU with one thread computing a block of moments

Here we improve the previous installation by assigning more work per thread by computing a whole block of moments of a mutable size we are meant to investigate in its optimal outputs. Before we can find an appropriate size of the blocks, we must first design a pattern of placing the device blocks of size b, across the grid. This is a packaging problem of placing a number of boxes of size b inside a larger box of size n which is the grid space. To deal with squeezing of sometimes an odd number of blocks to boost productivity, we take the cost of changing the allocated grid size to; $\left(\frac{n+b_x-1}{b_x}, \frac{n+b_y-1}{b_y}\right)$.

This should deal with an appropriate number of odd cases, even countering non square shapes, for packaging of the largest possible number of blocks

across the device grid. Then b^2 threads inside each block cooperate if allowed to trade their (shortly provided) shared memory.

For evidence that are mentioned on the results section, by fixing every other variable of the script and changing the b value for multiple iterative runs, it is decided that an arbitrary value of $b = 5$ is fitting for our needs.

3.4 V3 GPU with multiple thread sharing common input moments

Finally the last implementation of the Ising model using CUDA, calls for the use of shared memory between threads, such that the number of read commands for the main memory be minimized. On technical terms, this is achieved by extending our `__global__` update function with `__shared__` memory array of `int` elements. On parallel each participating thread in the joint processes, loads the centered lattice point its found data into the shared memory. Then after a synchronization between them to ensure that no thread is left behind, the known summation takes place.

4 Results

Here are the reports of execution times and their speedups for different values of n and k. Each version of code is tested and compared with each other

4.1 V1 Results

On the first stress test for V1 we fix all other variables and iterate over a number of different n values. The k value for all of them is fixed at 20, because, after many experiments, we can typically see the model reach a state of equilibrium around 5 to 6 iterations in. We choose a round 20 as a k value since it is a fair value for any size of n to have reached its own equilibrium.

We can see from figure 2 that the median execution time has a positive rate of change. Although not following a $O(n^2)$ complexity rate, it clearly isn't a

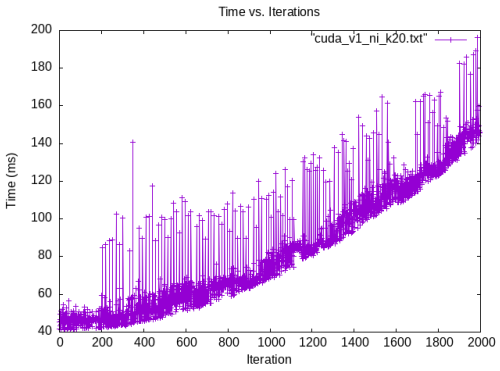

Figure 2: V1 time scores for values of $n = 3 \dots 2000$ and $k = 20$

linear rate. This could be explained by the ever increasing volume of data being crossed between host and device, since by the capabilities of the device itself the shouldn't explain this rate of change.

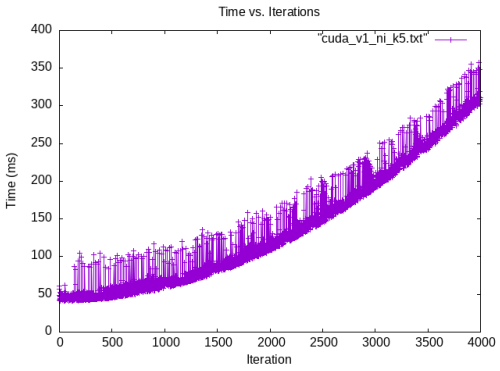

Figure 3: V1 time scores for values of $n = 3 \dots 4000$ and $k = 5$

We can tell the same for figure 3 with the bigger $n = 4000$ upper size and a less stressful $k = 5$ number of iterations.

4.2 V2 Results

Before we make the similar n k reviews we must find the appropriate b value for of block size. Here we fixed an iteration of runs to the values of $n = 2000$

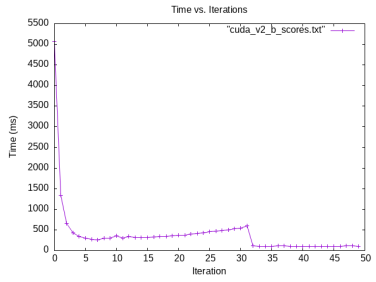

Figure 4: V1 time scores for different values of $b = 1 \dots 50$ and $n = 2000$ and $k = 2000$

and $k = 2000$ for different choices of b from 1 to 50. We can see a figure 4 that increasing our block size we reach a local minimum at $b = 6$ before dropping to a homogeneous minimal area for values greater that 32. For reasons explained by our choice of n & k ,we will fix for now on the general b value to $b = 5$ to cover a larger possible area of different arguments.

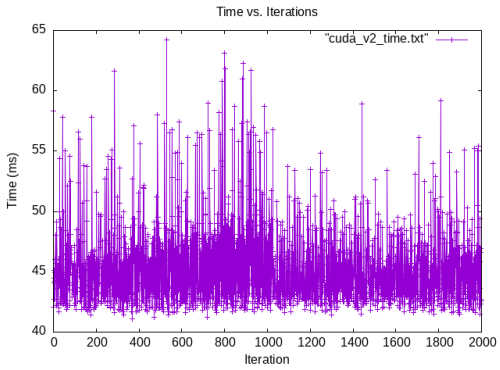

Figure 5: V2 time scores for values of $n = 3 \dots 2000$, $k = 20$ and $b = 5$

Now with a known block size we compare the V1 and V2 methods. As shown in figure 5, V2 clearly (despite a frequent amount of lags) is a huge improvement from the V1 method. For the same range of inputs, V2 manages to keep its median range of values constant.

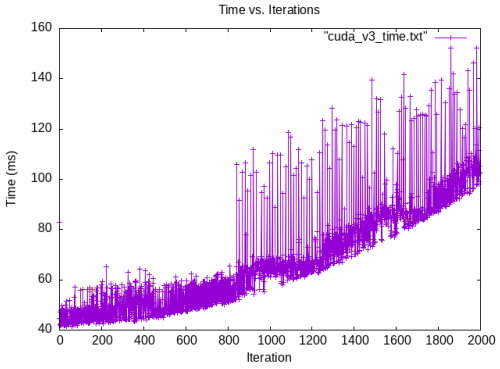

Figure 6: V3 time scores for values of $n = 3 \dots 2000$, $k = 20$ and $b = 5$

4.3 V3 Results

Finally, for the same block size of $b = 5$ we will stress V3 and find out how much the shared memory between n threads improves the time performance.

Unfortunately, looking at figure 6 we can tell that the current implementation is slower than the previous one.

5 GitHub link

The source code mentioned in this paper is accessible here

6 Usefull References

- https://stanford.edu/ jeffjar/statmech/intro4.html
- https://docs.nvidia.com/cuda/cuda-c-programming-guide/contents.html
- https://developer.nvidia.com/