Exercise III

Parallel & Distributed Systems

An implementation of the Ising Model in CUDA

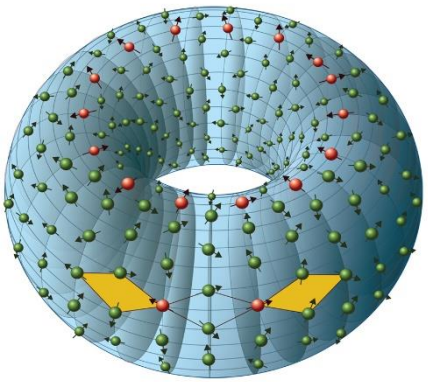
Christos Papakonstantinou 8531 papachri@auth.ece.gr

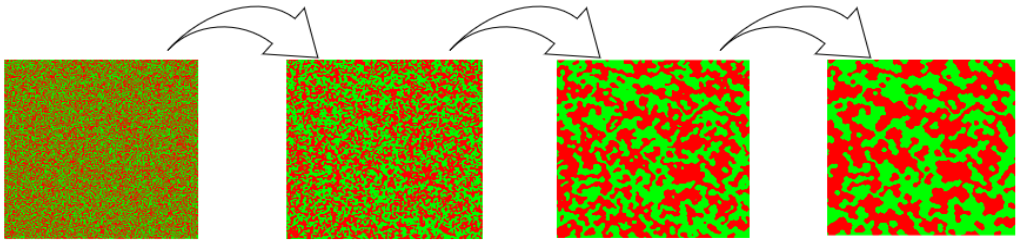
Evripidis Chondromatidis 8527 eurichon1996@gmail.com

Github Code & Graphical Demo: https://github.com/eurichon/ISING-MODEL-IN-CUDA.git

Problem

The Ising model is a mathematical model of ferromagnetism in statistical mechanics. The model consists of discreate magnetic dipole moments of atomic "spins" that can be in one of two states (+1 or -1). The spins are arranged in a square lattice with periodic boundary conditions, forming a torus in space and allowing each of the spins to interacts with its neighbors. Finally, the dipole moments update in discrete time steps according to the majority of the spins within an m-by-m windows. Each of the neighbor spins contributes to the resulting spin with a weight which is inversely analogous to the distance between the two.





Our Graphical Demo displays the evolution of the model until convergence

Sequential

The sequential implementation is pretty straightforward. We first allocate memory for three matrices: **Model**, **Up_Model** and **Mask** which contain the model at the current state, the model state after one update and the weight coefficients which affects the neighbors spin moments. We then initialize the «**Model**» matrix with the initial configuration given by the «conf-init.bin» file which is provided. It is crucial to write the result in a different matrix as the moments are interdependent and a write operation in the original matrix during the update will result in a different outcome.

The whole calculation is made by function «Ising» which calculates the k-updates by calling the «IsingOnce». IsingOnce iterates throw every point of the «Up_Model» and request the resulted spin moment by the «updateSpin» function. The last one is responsible for the computation of the weighted average using the «Mask» and the neighbor spin moments of the «Model» matrix for a

given moment point. However, in order to take the toroidal boundary conditions into consideration the element access in done throw the «posRoll» function which takes the two coordinates of a point and if is out of bound it shift the about the reverse side. Then it adds the two coordinates with the formula $Index = i * n + j$ in order to match the row-major indexing of the matrix «Model». Lastly, after its update is completed, we swap the pointer of matrices «Model» & «Up_Model».

CUDA Implementation (V1 & V2)

As a first approach of parallelizing the Ising model in CUDA we launch a kernel with a combination of Grid size and Block size so as the resulted threads match the number spin moments. The block size was kept constant «32,32» while the grid size was depending upon the n size of input model. This tactic as simple as is it, is not practical nor scalable.

In order to tackle the scalability issue in version II we assign more job per block reducing the number of them. This resulted in performance degradation because each thread has to access global memory a huge amount of times as well as we have poor GPU occupancy!

Cuda Optimized Implementation (V3)

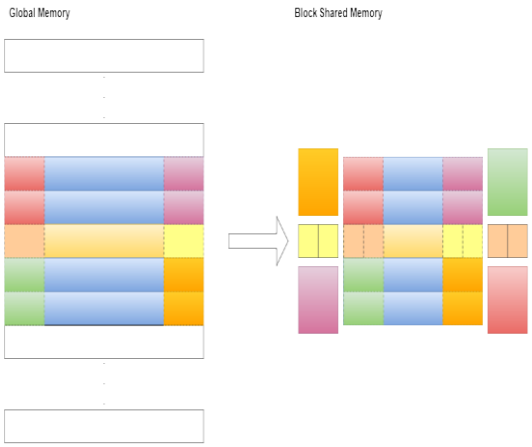
In this version we aimed to achieve optimal performance by utilizing three key concepts alongside with a much better blocks per SM & thread per Block ratio which result in better GPU occupancy!

- ***Coalesced access of Global memory***

  In order to transfer the data in shared memory we will have to access them from global memory at least once. Because threads in a block are grouped in «Warps» which means that are completely synchronous (SIMD) if we access continuous memory chunks, we can reduce the number of read operations and as a result the transfer time taking into advantage the GPU architecture. For this reason, the row-major array is split into «1xn» stripes and each of them is assigned to one block.

- ***Use of shared memory in each block***

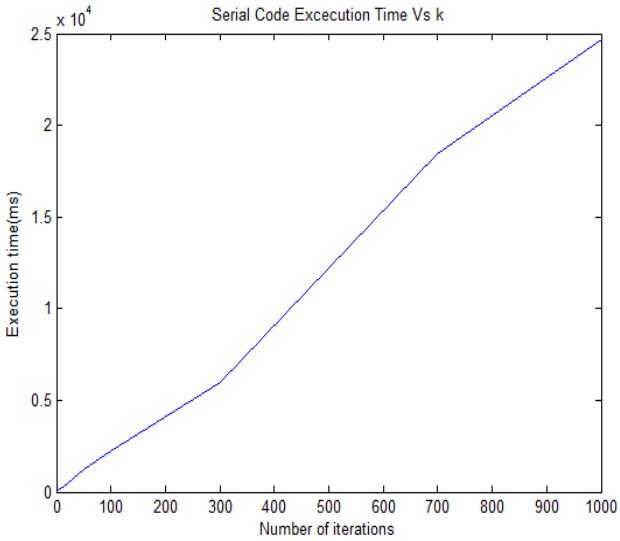
  The implementation of this algorithm is a classic case of 2-d stencil problem. That means for each spin's moment calculation we will have to access memory multiple times (in this case x25). Therefore, shared memory provides a significant advantage due to its superior access time. However, while faster than global, it is very limited in size and local to each block. That mean each block must run independently and for that reason each block has to copy both his own and the adjacent stipes in order to satisfy the boundary conditions as shown in the figure.

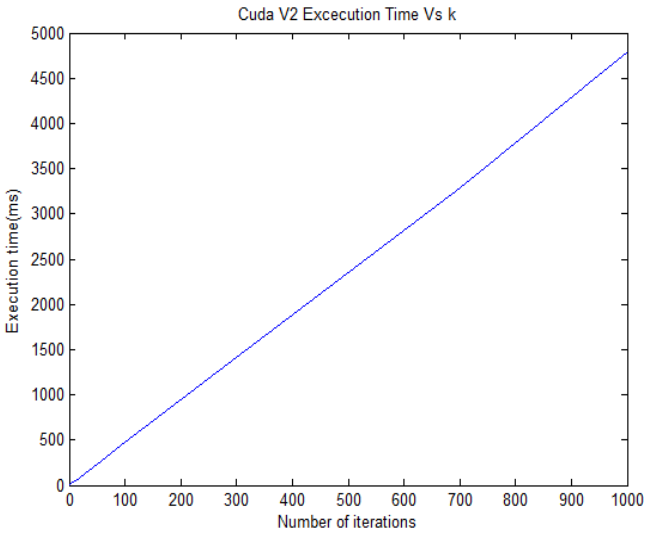


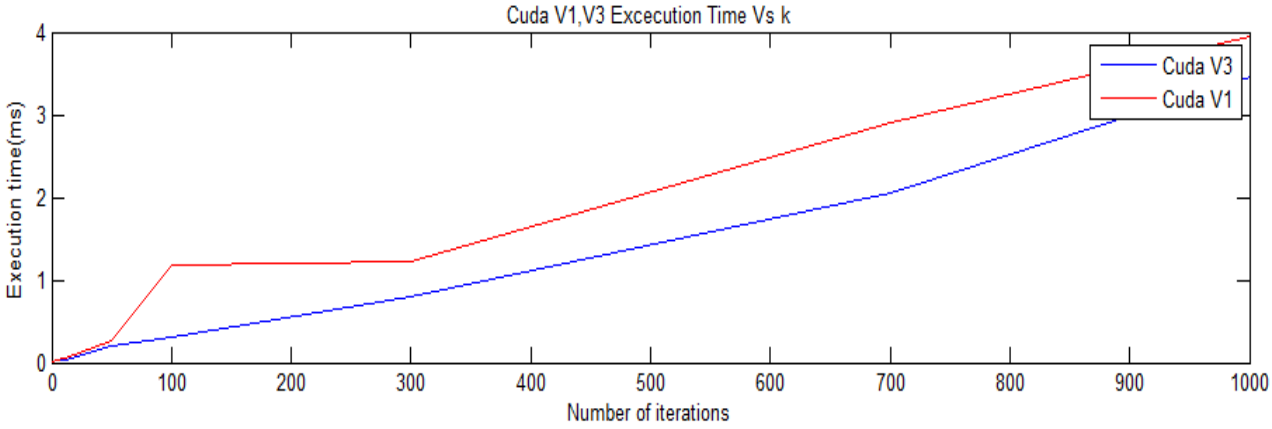
- ***Avoid shared memory bank conflicts***

  After the five rows are copied to shared memory each block has all the points to fill the boundary conditions. So, we use a simple switch statement in order to manually assign one of these last copy operations to a thread and avoid back conflicts which result is serialization and reduced performance.

RESULTS







Final Comments

- As expected, the Execution Time of the all the versions is approximately a linear function of the number of iterations(k).
- The CUDA V1 implementation is faster that the sequential by 4 orders of magnitude.
- The CUDA V2 implementation is slower a lot slower than V1, since we assigned a lot more work to each thread.
- Finally, we observe that the CUDA V3 implementation is the fastest, since we utilized the block shared memory and used coalesced memory access of the Global memory in order to achieve optimal performance while using significantly less threads than V1. This resulted in better resource management and higher GPU occupancy.

Machine Specifications

CPU: Intel Core i7-7700HQ	GPU: GeForce GTX 1050 Ti	RAM:16 GB