Parallel and Distributed Computer Systems Third Project

Ιωάννης Παναγιώτης Μουτεβελίδης - AEM: 10442

Introduction

The present project implements an algorithm, which visualizes the evolution of an Ising Model in two dimensions. The current program is being analyzed, both from a sequential and parallel aspect using CUDA GPU programming. The C programming language was utilized for the algorithm development, taking advantage of its support for GPU parallelization techniques. The project was implemented in CUDA.

The project was developed using an AMD Ryzen 3700U CPU quad-core processor with a base clock speed of 2.30 GHz, as well as a Nvidia GeForce GTX 1050 GPU.

The algorithm experiments were undergone on the HPC Cluster ”Aristotle,” which provides superior GPU capabilities. The GPU card utilized was Nvidia RTX 6000. In this way, we conducted tests on a better GPU that exceeds by far the local GPU's computation capabilities.

Each implementation, including the data of each diagram, is located in the Project_3 folder on Github at the link below:

Github:
https://github.com/jomout/Parallel_Distributed_Systems

The Problem

The Ising Model is a mathematical model of ferromagnetism in statistical mechanics, which consists of discrete magnetic dipole moments of atomic spins. Each moment could be either one of the two states, +1 or -1. The moments are arranged in a square lattice of two dimensions. Each moment interacts with its four immediate neighbors and updates its state. The edge lattice points wrap around to the other side.

The magnetic moments is updated in discrete time steps, according to the following formula:

$$Sum = G[i-1,j]+G[i,j-1]+G[i,j]+G[i+1,j]+G[i,j+1]$$

$$N[i,j] = \text{sgn}(Sum) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$$

In this project, four C scripts are being utilized, `sequential.c`, `v1.c`, `v2.c`, `v3_1.c` and `v3_2.c`. Each one of them represent the following implementations accordingly:

I. Sequential Approach - V0

The analysis below focuses on the sequential algorithm.

A. Initialization

The program begins by defining the width of the squared 2D Ising Model's array (n), as well as the iterations (k) of the algorithm, which are given by the user.

We initialize two 2D arrays by dynamically allocating memory, so the size of each array is `n · n · sizeof(int)` bytes. These represent the input and the output arrays of the Ising Model, `lattice1` and `lattice2`. By using the `initilize()` method we set the `lattice1` array in a random initial state.

B. Main Loop

After the random initialization of `lattice1`, the process begins the stimulation of the Ising Model for k iterations. By calling the `stimulate()` method, we iterate over the moments of the `lattice1` matrix and we execute the former formula for each one of the moments.

In order to tackle the absence of if statements for the boundaries, we simply apply the mod (%) operation to the i and j of each moment. The value of the `lattice2`'s element in the i_{th} and j_{th} position is computed by interpreting the mathematical $sgn()$ function as followed:

$$Sum = G[i-1,j]+G[i,j-1]+G[i,j]+G[i+1,j]+G[i,j+1]$$

$$L[i,j] = (Sum > 0) - (Sum < 0)$$

In this way, we can obtain the sign of the `sum` variable without utilizing an if statement.

Listing 1. Sum

```c
int sum = current[((i - 1 + n)%n)*n + j] +
          current[((i + 1)%n)*n + j] +
          current[i*n + j] +
          current[i*n + (j - 1 + n)%n] +
          current[i*n + (j + 1)%n];

next[i*n + j] = (sum > 0) - (sum < 0);
```

At the end of the each loop iteration, we simply swap the pointers of the two lattices, `lattice1` and `lattice2`.

C. *Diagrams*

The diagrams, below, are representing the execution times of the program, depending on the width of the Ising Model n and the number of iterations k.

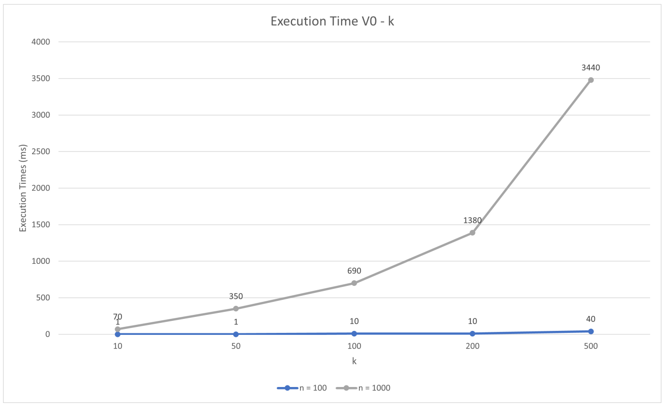

Figure 1. Time Executions of V0 - n = 100 and 1000

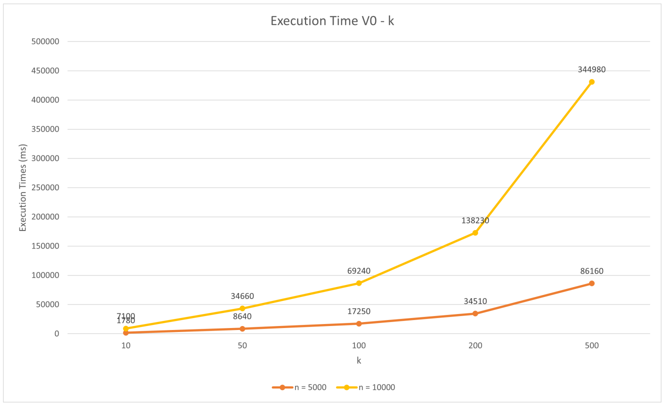

Figure 2. Time Executions of V0 - n = 5000 and 10000

The Figures 1 and 2 depict the impact of the Ising model's width n and the number of iterations k in the execution times. It is clear that as these two factors increase, the execution time is increased extensively. This happens, because the algorithm has time complexity $O(n^2)$.

II. CUDA WITH ONE THREAD PER MOMENT - V1

The analysis below focuses on the first CUDA version of the algorithm.

A. *Algorithm*

In this implementation, the user is asked todefine the width of the Thread matrix, `BLOCK_WIDTH`, that each Block will handle. So the number of threads per GPU block is equal to $\texttt{BLOCK\_WIDTH}^2$. Each thread is assigned to one moment in the 2D array, thus the number of threads per GPU block is equal to the number of moments each GPU block will handle.

The program computes exactly the number of GPU blocks needed for the the width of the Thread matrix, `BLOCK_WIDTH`, thus the number of threads per block that the user defined.

The algorithm calls the `stimulate()` method by defining the grid of GPU blocks as well as the grid of threads per GPU block. Each thread gets its own moment, thus, the (i, j) of each thread in the thread's grid corresponds to the moment's (i, j) in the lattice array:

Listing 2. Indexing

```
//Getting index
int j = blockIdx.x * blockDim.x +
        threadIdx.x;
int i = blockIdx.y * blockDim.y +
        threadIdx.y;
```

The rest of the code remains the same as with the sequential approach.

B. *Diagrams*

The diagrams, below, are representing the execution time of the program, depending on the width of the Ising Model n. Each number on the X-axis represents the width of the GPU block, in other words the width x of the thread matrix on a GPU block, as each thread has one moment. The number of the Threads per GPU block equals to x^2.

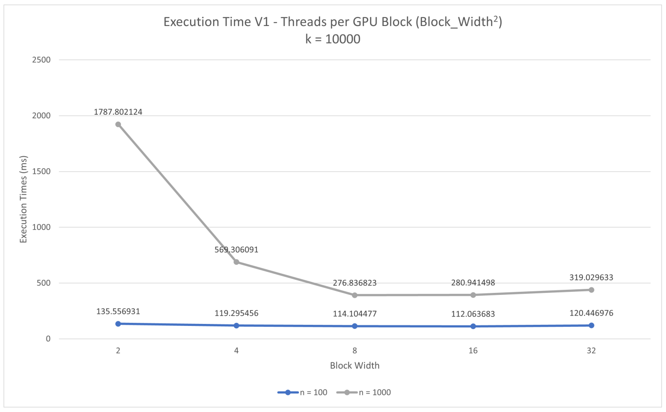

Figure 3. Time Executions of V1 for k = 10000 - n = 100 and 1000

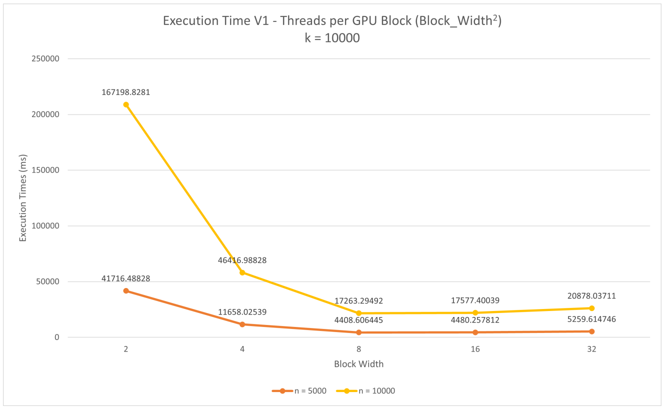

Figure 4. Time Executions of V1 for k = 10000 - n = 5000 and 10000

The Figures 3 and 4 depict the impact of the number of threads per GPU block used for different values of n. We can safely assume that increasing the number of threads per GPU block significantly reduces program execution time.

III. CUDA with one Thread computing a Tile of Moments - V2

The analysis below focuses on the second CUDA version of the algorithm.

A. Algorithm

In this V2 implementation , the user is asked define the width of the moments chunk of the lattice, `BLOCK_WIDTH`, that each Block will handle, as well as the tile width `TILE_WIDTH` of the moments that each thread will get assigned to.

The width of the Thread Matrix per GPU Block is equal to:

$$M = \lceil \frac{BLOCK\_WIDTH}{TILE\_WIDTH} \rceil$$

The number of threads per GPU block is equal to M^2. Each thread is assigned to one tile formed in the moments chunk of each GPU Block.

The algorithm calls the `stimulate()` method by defining the grid of GPU blocks as well as the grid of threads per GPU block. Each thread gets its own tile, thus, the (k, l) of each thread in the thread's grid corresponds to the starting moment's (i, j) in the tile:

Listing 3. Indexing
```
//Getting index
int k = blockIdx.y * BLOCK_WIDTH +
        threadIdx.y * TILE_WIDTH;
int l = blockIdx.x * BLOCK_WIDTH +
        threadIdx.x * TILE_WIDTH;
```

In a nested for loop we iterate over the moments of each tile, by calculating the (i, j) position values of each moment in the tile accordingly. The rest of the code remains the same as with the sequential approach.

B. Diagrams

The diagrams, below, are representing the execution time of the program, depending on the width of the Ising Model n. Each number on the X-axis represents the Tile width of each Thread depending on the current n value. In this way, we can achieve same total number of threads among the experiments, thus and same number of threads per GPU block.

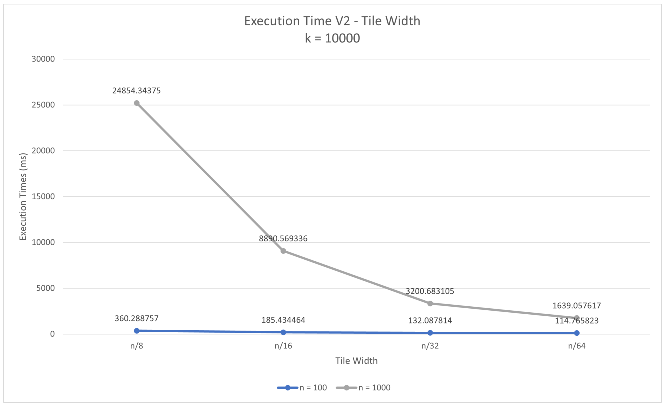

Figure 5. Time Executions of V2 for k = 10000 - n = 100 and 1000

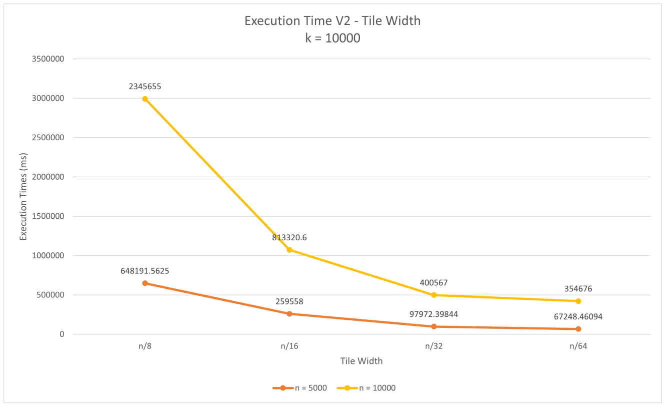

Figure 6. Time Executions of V2 for k = 10000 - n = 5000 and 10000

The Figures 5 and 6 depict the impact of the Tile width, each thread is assigned to. The way that is represented in the diagram, is such so the same total number of threads are used in each experiment. For instance, we have Tile width = $\frac{n}{4}$, which means that we have total number of threads equal to 16, etc. The Tile width is in fact in descending order, so for larger Tile widths, we have increased execution times.

IV. CUDA WITH MULTIPLE THREAD SHARING COMMON INPUT MOMENTS - V3

The analysis below focuses on the third CUDA version of the algorithm. We need to have the minimum amount of shared memory registration and thus the maximum amount of moments used from the shared memory for the computation of the next state of each moment. This leads us to organize the moments in block groups, thus the threads of GPU blocks will be in a thread matrix. For this algorithm, two approaches were implemented:

A. *Each Thread has its own moment*

This implementation is similar to the V1 second approach version of the algorithm. The following were added:

- The program defines the size of the shared memory which is equal to the size of the block of moments that each GPU block will handle, plus 2 rows and 2 columns for the neighbors of each moment.
- If the thread - moment is on the border of the thread matrix, then will have to register more moments to the memory than itself alone.
- We use the `__syncthreads()` method to wait for the moment registration. After the registration of each moment needed is done by the threads, each thread calculates the next state of its moment, using the shared memory.

B. *Each Thread has its own tile of moments*

This implementation is similar to the V2 second approach version of the algorithm. The following were added:

- The program defines the size of the shared memory which is equal to the size of the block of moments that each GPU block will handle, plus 2 rows and 2 columns for the neighbors of each moment.
- The moments assigned to each block are organized in smaller tiles, which are assigned to the corresponding thread of the GPU block.
- If a thread - tile is on the border of the thread matrix, then for each moment on the border of the block chunk, will have to register more moments to the memory than itself. This process is done in a nested for loop.
- After the nested for loop, we use the `__syncthreads()` method to wait for the moment registration.
- After the registration of each moment needed is done by the threads, each thread calculates the next state of moments in its tile, using the shared memory, in another nested for loop.

This implementation is not as optimal as the first one, as it uses two for nested loops, but is a great approach nevertheless.

C. *Diagrams*

*1) **First approach of the V3 - Each Thread per moment**:* As previously on V1, the diagrams, below, are representing the execution time of the program, depending on the width of the Ising Model n. Each number on the X-axis represents the width of the GPU block. The number of the Threads per GPU block equals to x^2.

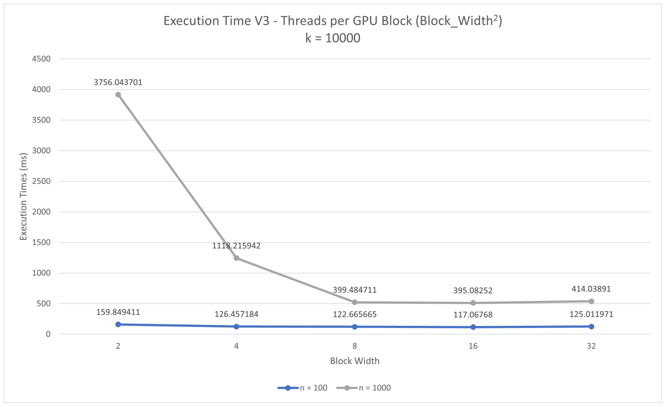

Figure 7. Time Executions of V3 for k = 10000 - n = 100 and 1000

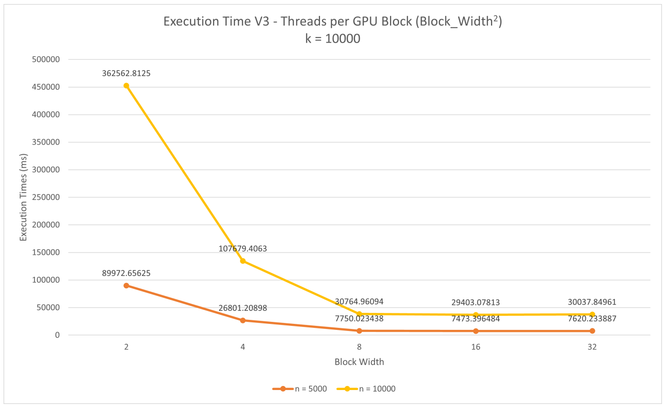

Figure 8. Time Executions of V3 for k = 10000 - n = 5000 and 10000

The Figures 7 and 8 depict the impact of the number of threads per GPU block used for different values of n. We can safely assume that increasing the number of threads per GPU block significantly reduces program execution time. The performance of using shared memory, compared to the V1 implementation, remains the same.

*2) **Second approach of the V3 - Each Thread gets a Tile of Moments**:* As previously on V2, the diagrams, below, are representing the execution time of the program, depending on the width of the Ising Model n. Each number on the X-axis represents the Tile width of each Thread depending on the current n value. In this way, we can achieve same total number of threads among the experiments, thus and same number of threads per GPU block.

The shared memory size, however, cannot be larger than $100 \cdot 100 \cdot sizeof(int)$, a feature that was observed, while setting different values for the block of moments, each GPU block gets. This means that the block of moments cannot have width larger than 100 moments, which restricts the values of tile widths to be smaller than 100 moments.

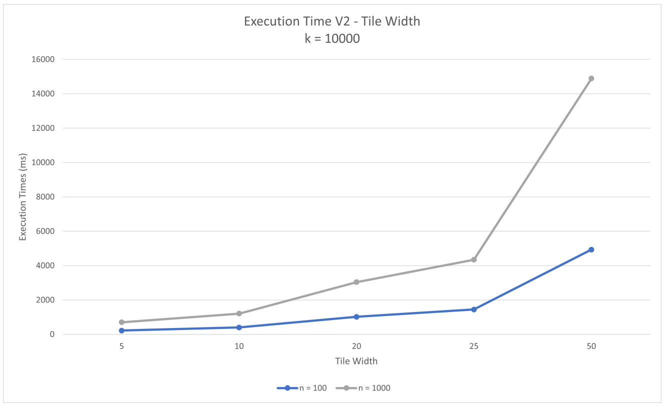

Figure 9. Time Executions of V3 for k = 10000 - n = 100 and 1000

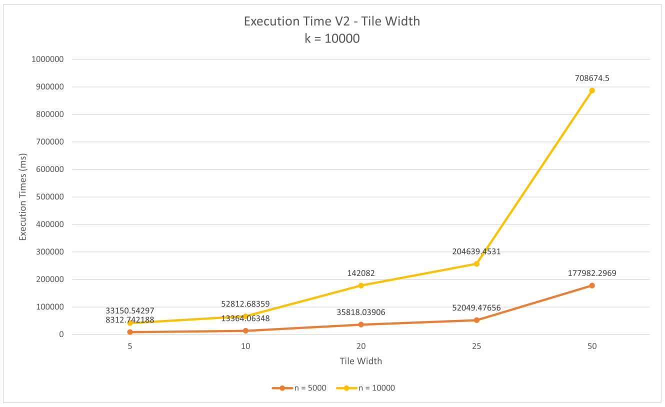

Figure 10. Time Executions of V3 for k = 10000 - n = 5000 and 10000

The Figures 9 and 10 depict the impact of the Tile width, each thread is assigned to. As we see, the smaller the Tile width, the better the performance of the algorithm. For larger Tile widths, we have increased execution times.

V. Results - Conclusion

In the diagrams above, we measure different execution times with respect to the Ising model's width n, the number of iterations k and the Tile width in the V2 and V3b implementations, in order to visualize the effect of each factor to the final result.

The increase of the Width of the moments chunk that each GPU block is assigned, clearly, enhances the performance of the program. More threads per GPU block has much better performance than more GPU blocks and less threads per block, for the same amount of total threads.

References

[1] https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/
[2] https://edoras.sdsu.edu/ mthomas/sp17.605/lectures/CUDA-Mat-Mat-Mult.pdf
[3] https://docs.nvidia.com/cuda/archive/9.1/pdf/CUDA_C_Programming_Guide.pdf
[4] https://developer.nvidia.com/blog/easy-introduction-cuda-c-and-c/
[5] https://gist.github.com/raytroop/120e2d175d95f82edbee436374293420
[6] https://github.com/lzhengchun/matrix-cuda
[7] https://github.com/CoffeeBeforeArch/cuda_programming/tree/master