Ising Model - CUDA Implementation

Aimilia Palaska 10453, aimiliapm@ece.auth.gr

Introduction and Problem Description

The Ising Model, a canonical model in statistical mechanics, serves as a fundamental paradigm for understanding the behavior of physical systems composed of discrete interacting entities, often employed to simulate phenomena such as ferromagnetism. Originating from the realm of theoretical physics, the Ising Model encapsulates the discrete spins of individual particles within a lattice, exhibiting binary values (+1 or -1) to denote their orientations. The inherent complexity of the Ising Model necessitates multiple independent computations for each moment, rendering it an ideal candidate for parallelization on Graphics Processing Units (GPUs) through the utilization of CUDA (Compute Unified Device Architecture) programming.

In this context, the implementation of the Ising Model algorithm in CUDA not only aligns with the inherent parallel nature of the problem but also capitalizes on the computational power of GPUs, thereby facilitating the efficient simulation and analysis of large-scale systems with substantial implications for diverse scientific domains.

Code: GitHub, Google Drive

Input Handling

In relation to handling the grid's initial state, a file denominated as "input.c" was created to generate a text file containing a predetermined quantity of +1 and -1 values. The selection of these values was achieved randomly through the seeding of the srand() function. The user has the flexibility to modify the desired quantity of values (denoted as 'n' and taken as an input to the executable) within the source code. Additionally, each CUDA implementation incorporates a function titled "takeInput()" responsible for parsing the contents of the aforementioned file and loading the host array with its data before beginning the core algorithm. Although an attempt was made to parallelize the input loading process, notable enhancements in execution time were not observed.

V0. Sequential

In order to gain a more profound comprehension of the algorithm's nature and to establish a preliminary framework, it was imperative to construct a sequential version of the Ising Model. This particular implementation abstains from the utilization of CUDA functions, as reflected in the file extension being denoted as ".c". In the context of the grid structures, the strategy employed involved the allocation of a two-dimensional integer array.

V1. GPU with one thread per moment

For the initial experiments in CUDA and its functionalities, the simplest possible approach was adopted. This entailed assigning a single element to each thread, retrieved from CPU memory, in order to compute a singular moment. It is evident that the number of threads is always equal to n^2 in this case.

It is important to highlight that, due to challenges associated with two-dimensional arrays within the kernel function, it was rather preferred to convert the plane into a vector. Necessary adjustments concerning the borders were implemented accordingly. The ensuing results were validated against version 0 (v0) through the utilization of the "compareFiles.c" script, thereby guaranteeing the accuracy of the outcome.

V2. GPU with one thread computing multiple moments

A more complex implementation involved increasing the computational workload allocated to each thread. Specifically, a block of elements (with predetermined size that can be modified through the variable “threadElements”) was assigned to individual threads. These threads executed computations similar to those in version 1 (v1), in regards to the subsequent state of the grid. The challenge in this context resided in the correct division of the workload, prompting the decision to allocate a slightly larger block of elements to the last thread when their number was not perfectly divisible by the block size. Various block sizes were subjected to testing, and the conclusions are analyzed in the benchmark section of the report. Here, only n^2/threadElements threads were employed.

V3. GPU with multiple thread sharing common input moments

In the ultimate version, an attempt was made to optimize the number of main memory accesses by employing shared GPU memory. This consisted of creating “__shared__” matrices within each block, denoted as "blockMemory", which were loaded in parallel with the corresponding data from the main memory. Following necessary synchronization, the threads determined whether the data essential for calculations could be sourced from the shared memory or not. In instances where values were situated near the block borders, some accesses to the main memory were inevitable. Similar to the initial version, the computation focused on a singular moment, although it is noteworthy that this version could be easily extended to accommodate a multi-moment approach.

Benchmarks and Tests

Many tests were run remotely on Aristotle’s HPC, with their corresponding results and graphs in this google sheets file. Below are the derived conclusions:

- The optimal block size for the second version (v2) is two moments per block, although one and three moments per block were almost as good.
- Regarding the execution time, the sequential approach takes approximately 200 times more than the parallel one, showcasing the benefits of the CUDA implementation.
- Minimizing the memory accesses in version 3 did not beat the execution time of the other two versions, however it is undoubtedly more memory-friendly.

Other observations and comments

Please feel free to reach out to me if any of the links is not functioning or you have any questions about the implementations, I will be more than happy to help.

Sources and Citations

- ➔ The ChatGPT 3.5 language model was employed for the purposes of text formatting and the generation of certain portions of the code.
- ➔ The wikipedia article on the Ising Model
- ➔ Documentation on the use of shared memory in CUDA