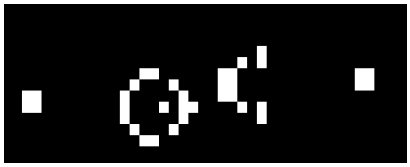
Instituto Tecnológico y de Estudios Superiores de Monterrey

A Proposed Framework for the Implementation and Prototyping of Parallel Cellular Automata with CUDA

Programming Languages Aug-Dec 2020

Author:
Juan José Olivera Loyola

Professor:
Benjamín Valdés Aguirre



Index

I. Introduction

A common traditional problem across all computer science fields has been improving that of improving performance. We seek it continously today, and sought too back in the early years of computer science. By the end of the 1940's, computing opioneering John Von Neumann proposed the development of the theory of automata-networks as an effort to design and build multiprocessor machines. Then, in 1966, he introduced the concept of "cellular automata", a simple mathematical model capable of producing complex results from simple designed rules and is intrinsically parallel.

Automata-networks was developed along, and as a response to, the development of the second and third generation of computers. While its true that third generation technology servers as the base layer of current computing devices such as CPUs, GPUs, FPGs, etc., modern architectures were not really tested decades ago due to the difficulty of fabrication, power consumption and economical costs.

Now on days, dedicated GPU's are frequently found in user consumer PC's, and have reenabled the development of applications foundend on old computing models. One example is deep learning, (an older automata-network based model), which takes advantage of GPU's and TPU's to perform high computing operations. In this project we seek to see if cellular automatas could benefit as well from the exploitment of GPU's resources, as it was originally conceived, and if indeed they can, what could be done to motivate and help reasearchers to increasingly apply them in state-of-the-art problems.

II. Some Theory on Cellular Automatas

First, an intuitive presentation of automatas will be given to grasp the general idea, and then, we will formalize some components to clearly identify them later.

Cellular Automata

Let's learn what is a cellular automata by considering an example. We will refer to persons with neutral "it" pronoun. Suppose a world where each person always remains at home and the only inter-personal contant with other people is the its surrounding neighbors, (such a world during a global pandemic…). Each person only gets out once a day to recieve the bills, water the garden and on the way, salute the neighbors. It is only at this time of the day that people have opportunity to talk between each other so of course they use it to gossip. As they are gossiping, communication is discrete and quiet, so a person can only fully understand a rumor if 2 or more of its neighbors tell the rumor the same day. Nonetheless, every morning

each person has a conversation with every of its neighbors to hear about or spread the rumor. We can picture from the description a network of people where each individual is connected to its neighbors. People can either know the rumor or not know the rumor, and if they know it, they will try to spread it to its neighbors. In such a network, we would refer to every person as “cells”, the information that each person knows as the cell’s state and, the process of talking to the neighbors to transmit information is called, the update rule. The whole structure consisting of the cells and relationships, their states, and the update rule is known as a cellular automaton.

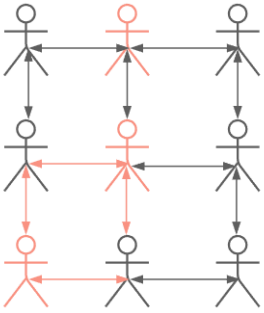


Fig 1. A visual representation of the gossiping people network. Each person is colored as orange if it fully knows the gossip. Here, west and south individuals, will fully know the gossip after the state update because they have at least 2 neighbors who already know the gossip.

Formalisms

A cellular automaton is first defined by its state as a vector X = (c1, c2, …, cn) of cells ci. Each cell stores a fixed amount variables that store the automaton information ci = {vi1, vi2, vi3… vim }. Next, we have a function V(ci) = { cells connected/related to ci }. And finally we have the update rule defined as F, such that Xt+1 = F(Xt). This definition could be seemed as a linear algebra transformation for example y = T(x). However, the update function f can be broken down into F(X) = { fi(V(ci)) for every ci in the network }. Computing the next state of a discrete model is known as the iteration step. Thus, from analyzing the update distributed formula, we can conclude that a single iteration process can easily implemented in parallel, as updating a single cell only requires access to the previous state of its neighbours cells.

A basic algorithm for a traditional cellular automata can be given as:

1. Set initial configuration state X and relations.
2. Make a copy of state X
3. Update each cell c computing a function based on its neighbours previous states stored in the copy of X
4. Repeat from step 2 until desired

There is more into the theory of cellular automata such as iteration graphs, state attractos, cycle paths, state derivates but, only with the first general definitions given, numerous CA models have been developed in scientific fields to simulate real world phenomena, such as cloud dynamic simulation, wildfire propagation and structural struss member optimization for example.

Programming language implementation:
A usual way of implementing cellular automata is to store cells' state values in a matrix and repeatedly loop over the whole matrix to compute the following states.

We will prove, that this can be done in parallel in a GPU with CUDA by assigning cells to CUDA threads.

Cellular Automata Usage

III. Proof of Concept

Game Of Life Experiment

Code: GameOfLife.sni {GoL.cu, gpuGoLSim.cu, cpuGoLSim.cu, mat_utils.h}
We perform Game of Life over a cell matrix of N*N, repeated N_STEPS iterations.
Computation Strategy: Convolutions

<u>System Specs:</u>
Host RAM: 16GB
Host Disk: SSD
Host CPU: Intel Core i5 8300
- Clock Speed: 2.3Ghz [Max 4Ghz]
- Number of Cores: 4 Cores (2 Threads per core)

Device GPU: NVidia GTX 1050
- Clock Speed: 1455 MHz
- RAM: 2GB
- Cores: 640

Experiment 1:
Experiment Parameters:
- N: 1000
- N_STEPS: 1000

1. SubExperiment
    a. Parameters:

        i. ThreadsPerBlock: dim3(32,32)
        ii. BlocksPerGrid: dim(128,128)
    b. Results:
        i. CPU Time: 41.426segs
        ii. GPU Total: 11.245segs
        iii. GPU Iterations: 10.185segs
        iv. Implications:
            1. It takes 10ms to process 1 iteration for the GPU
2. SubExperiment
    a. Parameters:
        i. ThreadsPerBlock: dim3(16,16)
        ii. BlocksPerGrid: dim(128,128)
    b. Results:
        i. CPU Time: 40.829segs
        ii. GPU Total: 5.731segs
        iii. GPU Iterations: 4.68segs
        iv. Implications:
            1. It took ~4.68ms to process 1 iteration for the GPU with a quareter the size of maximum available threads poer block
            2. 16x16 = 256
3. SubExperiment
    a. Parameters:
        i. ThreadsPerBlock: dim3(32,16)
        ii. BlocksPerGrid: dim(128,128)
    b. Results:
        i. CPU Time: 40.863segs
        ii. GPU Total: 6.917segs
        iii. GPU Iterations: 5.878segs
        iv. Implications:
            1. It took ~5.6, ~6ms to complete 1 iteration, greater than SubExperiment 2 but less than SubExperiment1
4. SubExperiment
    a. Parameters:
        i. ThreadsPerBlock: dim3(8,8)
        ii. BlocksPerGrid: dim(128,128)
    b. Results:
        i. CPU Time: ----
        ii. GPU Total: 3.961segs
        iii. GPU Iterations: 3.833segs
        iv. Implications:
            1. It took ~3.833, ~4ms to complete 1 iteration
            2. Seems that the lesser the #threads per block, greater the performance for 1000x1000 matrices

TODO: Reducing #Threads Per Block increases performance even for 4000x4000 case??

TODO: Does Mantaining balance has same effects? Performance ~= BLOCKSXTHREADS_PER_BLOCK for some size?
TODO: Do reducing the number of threads while increasing thread internal iteration results in an improvement?

Experiment 2
Experiment Parameters:

- N: 4000
- N_STEPS: 1000

IV. SubExperiment
    A. Parameters:
        1. ThreadsPerBlock dim3(16,16)
        2. BlocksPerGrid dim3(128,128)
    B. Results:
        1. CPU Time: 779.993
        2. GPU Total:59.759
        3. GPU Iteration: 58.573
        4. Implications:

V. Analyisis of Real World Cellular Automata

VI. PUCCA Framework

VII. Integration with external MASON Tool

VIII. Results

IX. Discussion

X. Conclusion and Future Work

XI. References

Bibliography
https://web.archive.org/web/20100718140020/http://www.swarm.org/images/1/10/How-to_set_up_%26_use_Eclipse_with_Mason.pdf