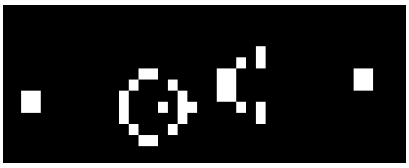
Instituto Tecnológico y de Estudios Superiores de Monterrey

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 [a]. 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 [b].

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 server as the base layer of current computing devices [c] 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 [a]), which takes advantage of GPU's and TPU's to perform high computing operations [d]. 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 the neutral pronoun "it" for simplicity. 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.

A visual representation of the gossiping people network.

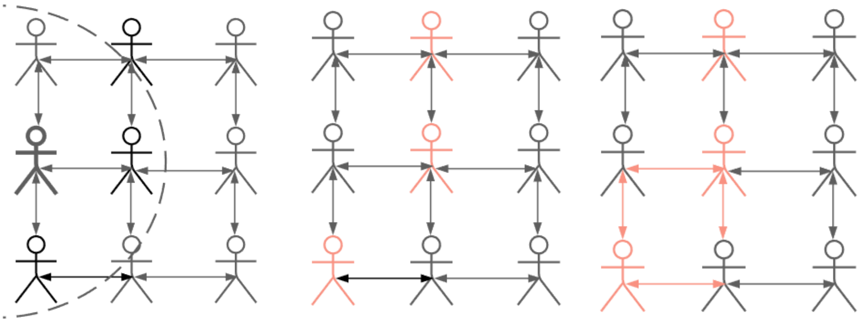


Fig 1.a), Neighborhood of **West** individual. Fig 1.b), People who know the gossip is colored with orage, othewise, black. Fig 1.c), Here, **West** and **South** individuals, will fully know the gossip after the state update because they have at least 2 neighbors who already know the gossip.

Formalisms

A cellular automaton is first defined by its **state** as a vector $X = (c_1, c_2..., c_n)$ of cells c_i. Each cell has a fixed amount variables that store the automaton information $c_i = \{u_{i1}, u_{i2}, u_{i3}, ..., u_{im}\}$. Next, we have a function ***V*** to get the **neighbourhood** of any cell c_i as followed: $V(c_i) = \{ \textit{every } c_j \textit{ that is connected to } c_i \}$. And finally we have the **update rule** defined as ***F***, such that $X_{t+1} = F(X_t)$. 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 $F(X) = (f_i(c_i) \textit{ for every } c_i)$ [a]. Computing the next state of a discrete model is known as the iteration step. Thus, from analyzing the update distributed formula composed of f_i, we can conclude that a single iteration process can easily be implemented in parallel, as updating a single cell only requires access to the previous state of its neighbours cells.

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.

Cellular Automata Usage

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

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

Programming language implementation:

Usually, the way to implementing a program that simulates a cellular automata is with matrices. Each cells' state values are stored in the corresponding cell of the matrix and then, repeatedly loop over the whole previous state matrix to compute the current states.

Additionally, updating rules and cell's for a real world application are often defined very simply. As a designer, you only need to think about what information is "known" by a single cell if you were to divide your problem into a collection of units, and how does these variables change in relation to the state of other cells [e]. This characteristic of "simple model-complex behaviour over time" has made them popular for many problem types. Cellular automata have a lot of presence in social science and biology domain, but many phenomena in physics, chemistry, medicine, computer science, among others has also been solved by CA.

However, research studies tend to implement a CA model that runs entirely sequential. A literature review was made on latest cellular applications, and it was found that, 3 out of the 4 recent articles reviewed concluded that, performance of the developed model could be significantly improve if the parallel paradigm was used, specially in the case of increasing the amount of variables involved [f] or, to solve large size problems [g]. One of them further indicates that increasing performance would enable "its ability to perform in real time" for production systems [f].

Given the interest on parallel cellular automata, we will prove their enhancement when they are done on parallel in a GPU with CUDA by assigning cells to CUDA threads, and in the following sections, we will propose a framework to easily implement, prototype, evaluate and benchmark parallel cellular automata.

III. Proof of Concept

Game Of Life

John Conway's Game of Life (GoL) is perhaps the most popular cellular automaton. Applications of Game of Life are only theorical to understand the complex behaviours than can arise from CA, and secondly, as an entertaining animation. However, its extreme simplicity makes it an excellent candidate for a first approximation and validation of concept.

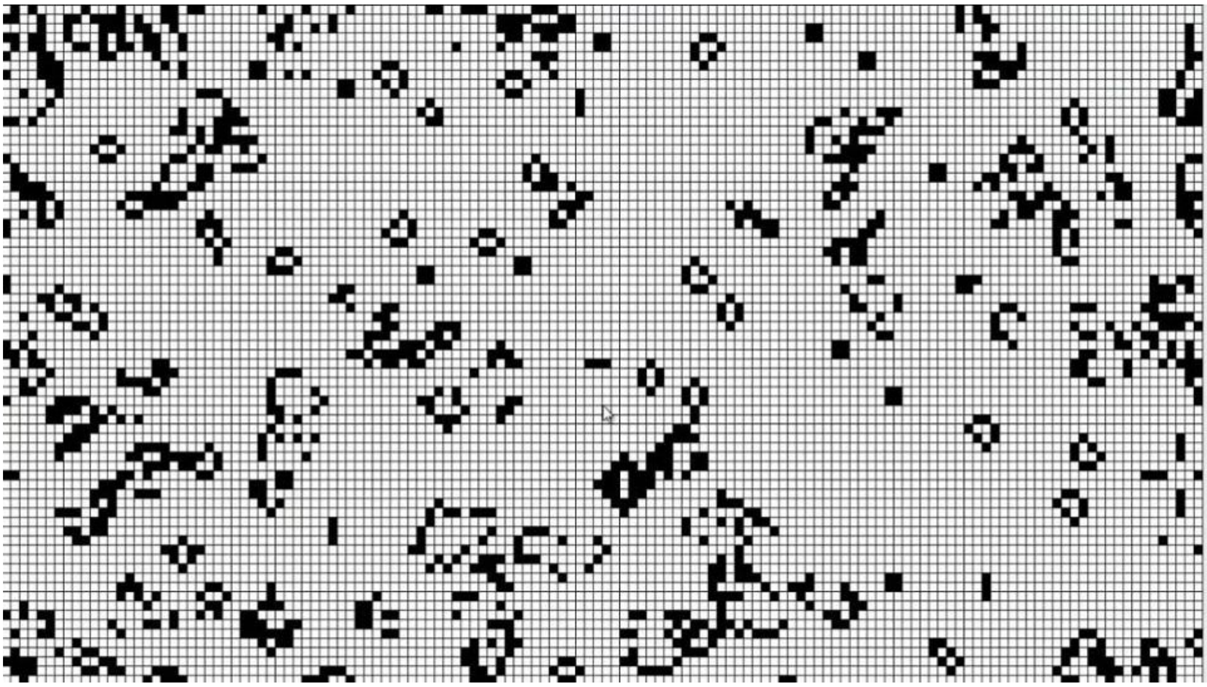


Fig 3. A medium sized Game of Life cellular automaton.
[It From Bit: Is The Universe A Cellular Automaton? | by Paul Halpern | Starts With A Bang! | Medium]

Definition:

The CA consists of a 2 dimensional grid of cells that can be either "dead" (represented by 0) or alive (represented by 1). Each cell will check its 8 surroding cells (Moore's neighborhood), to decide if it should come to live or die.

1. If it's dead, and there are precisely 3 alive neighbor cells, it will come alive.
2. If it's alive, and there are precisely 2 or 3 alive neighbors, it will remain alive, otherwise die.

In the figure below we can see the evolution of a (GoL) 6x6 configuration that starts as 2 alive squares. Then an update rule is applied and the middle 2 cells die as they are the only ones with more than 3 neighbors. A following update rule brings them to life as each of them has 3 alive neighbors.

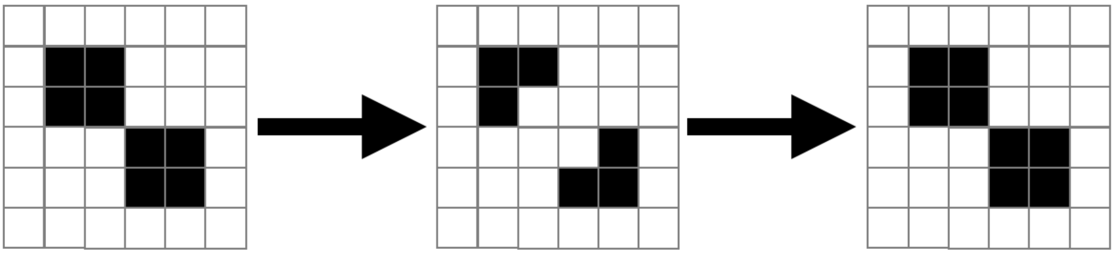


Fig 2., A simple evolution of a Game of Life 6x6 cellular automaton.
Conway's Game of Life in python. A simple implementation of Conway's… | by Robert Andrew Martin | Medium

CUDA Solution Design

As a first approach, we related (GoL) to a common large sized problem already optimized with GPU's, image convolution processing. Image convolution is a technique for processing images or signals with a filter named formally as the kernel. For computing the output of a specific pixel x_i, we multiply a small fixed kernel to the pixels around the input pixel x_i.

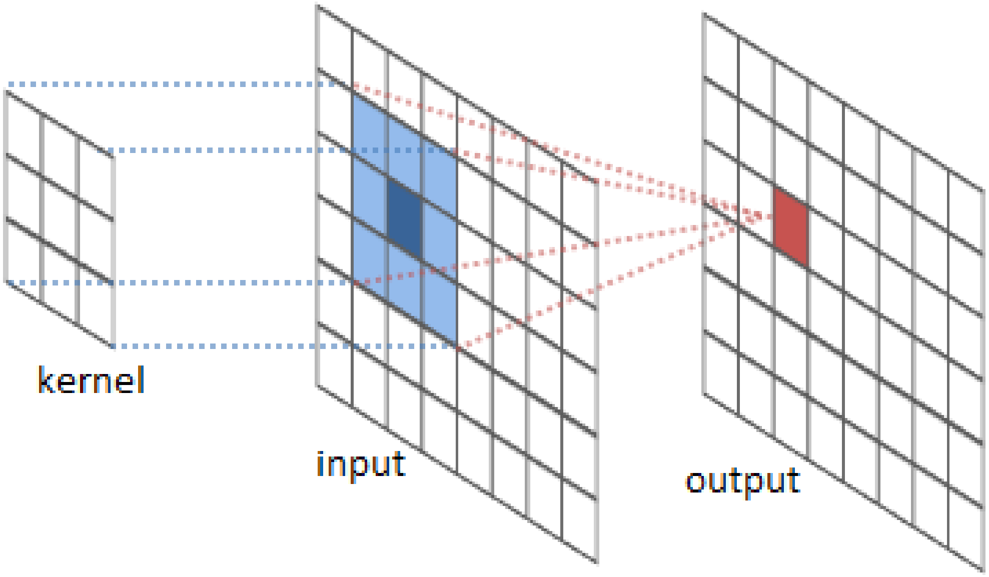
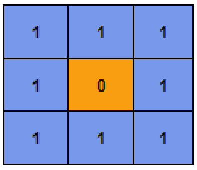


Fig 4. Computation of a convolution around a pixel.
Image Convolution From Scratch. Mathematical operation on two functions… | by Sameeruddin Mohammed | Analytics Vidhya | Medium

If we where to consider the kernel a neighborhood, and the convolution operator of the kernel over the input data an update rule, we can easily describe (GoL) in these terms. Applying a 3x3 kernel of 1's and a center 0, would be equivalent to counting the amount of live neighbor cells. We would then just need to update the cell based on a condition of the computed alive neighbors count.



1	1	1
1	0	1
1	1	1

Fig 5. Proposed convolution kernel for Moore's neighborhood.

$Y(x,y) = w * X(x,y)$, where the convolution operator $*$ is defined as the sum of the element-wise multiplication of the kernel and the cell grid, and a condition for updating the cell's state based on the resulting sum.

Implementation

In our program we will have the representation of a 2 dimensional matrix of int values to store GoL state, and a 3x3 matrix to store the kernel. We will initialize the grid with simple data, and compute a iteration step sequence applying the convolution method.

Two programming tricks which facilitated this labor where:

1. Expand the original input state grid of size nxn into an extended (n+2)x(n+2) grid with padding state 0's on the borders. This was done to avoid the annoying cases of computing the kernel over the locations that are not defined when the input cell is at the border.
2. Always have 2 matrices **A** and **B** of the extended grid to avoid copying each step the grid. We first have the initial configuration in **A.** When we compute the first iteration, we use **A** as the previous state and store the results in **B**. Then, thanks to the memory reference representation of arrays and matrices, we can create a temporal pointer **C** to **B.** Then we make variable **B** point to **A**, and finally we update **A** by pointing into **C.** In simple terms, we just swap matrices **A** and **B,** (to make previous state **A** be the latest computed state **B**), without actually copying every single data cell.

Two GoL C/C++ functions were programmed. One to run this algorithm sequentially using a single CPU thread, and the other, to do it on parallel using CUDA. This was done in order to compare time performance between the sequential CPU and parallel GPU implementations.

CUDA kernel threads where organized into 2 dimensional blocks, and blocks where organized as 2 dimensional grid. Each thread then would start by computing the update rule of the corresponding cell in the state grid. If not all the cells were initially mapped into threads, then the thread would advance horizontally with a blockDim.x*gridDim.x step to process remaining horizontal cells. Aftewards, it will try to advance vertically with a blockDim.y * gridDim.y step, and repeat the previous step to process the new row. This process is repeated in all of the launched CUDA threads.

Code: GameOfLife.sni {GoL.cu, gpuGoLSim.cu, cpuGoLSim.cu, mat_utils.h}

Benchmarks

Each experiment consist of a run Game of Life over a cell matrix of N*N, repeated N_STEPS iterations on both the sequential and parallel implementations.

System Specs:
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: 41.426s
        ii. GPU Total: 11.245s
        iii. GPU Iterations: 10.185s
        iv. Speedup: 3.683
        v. 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: 40.829s
        ii. GPU Total: 5.731s

            iii. GPU Iterations: 4.68s
            iv. Speedup: 7.124
            v. 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: 40.863s
        ii. GPU Total: 6.917s
        iii. GPU Iterations: 5.913s
        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: 41.02s
        ii. GPU Total: 3.961s
        iii. GPU Iterations: 3.833s
        iv. Speedup: 10.355
        v. 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

Experiment 2

Experiment Parameters:

- N: 4000
- N_STEPS: 1000

1. SubExperiment
    A. Parameters:
        1. ThreadsPerBlock dim3(16,16)
        2. BlocksPerGrid dim3(128,128)
    B. Results:
        1. CPU Time: 779.993s
        2. GPU Total: 59.759s
        3. GPU Iteration: 58.573
        4. Speedup: 13.0523
        5. Implications:

Proof of Concept Conclusions

Results show clearly that GPU parallel computation of large scaled GoL cellular automata produces significant performance enhancement, even with mid-range GPU's such as NVIDIA's GTX 1050. Some interesting results are:

1. The GPU parallel speedup increases as problem size increases as we expected.
2. The GPU on small cases will be equivalent or even slower, to the CPU due to initial memory device allocation and transfer.
3. Suprisingly, the GPU implementation performs better, when thread blocks with smaller launching dimensions are used. We could hypothesize that the phyisical GPU architecture prefers have to process just a fixed number of long threads similar to its internal simultaneos thread capacity, rather than a amount of small threads larger to its internal capacity.

IV. Analyisis of Real World Cellular Automata

Haven proven the performance potential of GPU parallelized cellular automata, we proceed to review recent cellular automata applications in different scientic fields to develop a framework that could be used by researchers to implement their ideas.

Article selection method

First, a query over cellular automata was performed on the ScienceDirect database. From the articles given, only the ones concerning the development of an applied cellular automaton were read. From these read articles, only the ones who explicitly defined the cellular automata operation (variables involved and update rule formulas) and had test parameters and results where kept. From this method we ended up with 3 articles: "Cloud Simulation"[f], "Peat Wildfires" [h] and "Struss structure optimization"[g].

Applied cellular automata generality

Each research used a different set of formulas and logic for the updating step according to their very own problem domain. And their cellular automaton algorithm had different variations to the traditional formal description. Therefore, in order to be able to compare the differences and extract commonalities between them, we represented each of their algorithms in a "normalized" flowchart with additional symbols to represent parallel steps.

Besides, common known flowchart symobology, as defining process executing or variable declarations within boxes and arrows denoting the sequence. We have added a dotted box outline to those sections that will be performed on parallel in each cell. Furthermore, the main sequence is always in one direction, vertical or horizontal. When additional space is needed because a procedure is complex or composed of different procedues, we wrote it to the "left" of the main sequence, and we used an arrow to indicate what item is being expanded and what is its body.

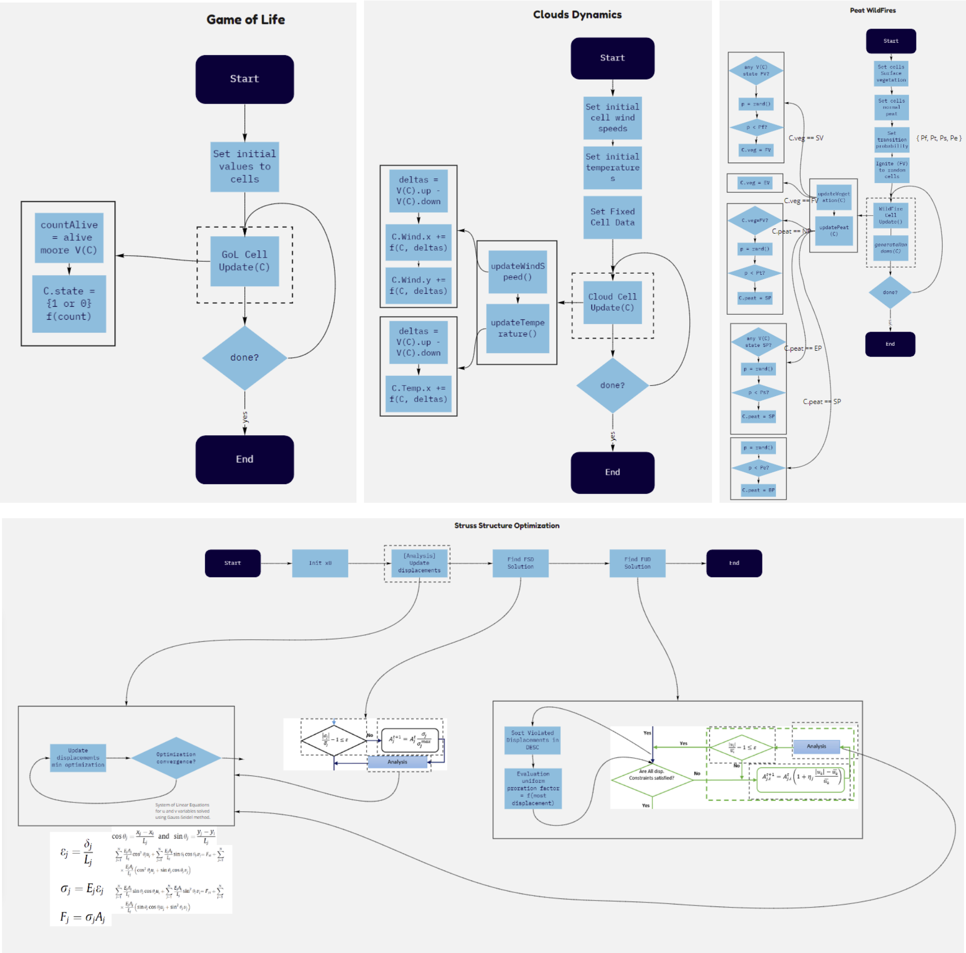

Game of Life
Start
Set initial values to cells
GoL Cell Update(C)
countAlive = alive moore V(C)
C.state = {1 or 0} f(count)
done?
yes
End
Clouds Dynamics
Start
Set initial cell wind speeds
Set initial temperature s
Set Fixed Cell Data
Cloud Cell Update(C)
updateWindSpeed()
updateTemperature()
deltas = V(C).up - V(C).down
C.Wind.x += f(C, deltas)
C.Wind.y += f(C, deltas)
deltas = V(C).up - V(C).down
C.Temp.x += f(C, deltas)
done?
yes
End
Peat WildFires
Start
Set cells Surface vegetation
Set cells normal peat
Set transition probability
{ Pf, Pt, Ps, Pe }
Ignite (FV) to random cells
WildFire Cell Update()
generateRandoms(C)
done?
yes
End
updateVegetation(C)
updatePeat (C)
any V(C) state FV?
p = rand()
p < Pf?
C.veg = FV
C.veg == SV
C.veg = EV
C.veg == FV
C.veg=FV?
p = rand()
p < Pt?
C.peat = SP
C.peat == NP
any V(C) state SP?
p = rand()
p < Ps?
C.peat = SP
C.peat == EP
C.peat == SP
p = rand()
p < Pe?
C.peat = BP
Struss Structure Optimization
Start
Init x0
[Analysis] Update displacements
Find FSD Solution
Find FUD Solution
End
Update displacements min optimization
Optimization convergence?
System of Linear Equations for u and v variables solved using Gauss-Seidel method.
$\frac{|\sigma_j|}{\bar{\sigma}_j} - 1 \leq \varepsilon$
No
$A_j^{t+1} = A_j^t \frac{\sigma_j}{\sigma_j^{max}}$
Analysis
Sort Violated Displacements in DESC
Evaluation uniform proration factor = f(most displacement)
Yes
Yes
$\frac{|u_i|}{\bar{u}_i} - 1 \leq \varepsilon$
Analysis
No
Are All disp. Constraints satisfied?
No
$A_{j,k}^{t+1} = A_{j,k}^t \left(1 + \eta_j \frac{|u_k| - \bar{u}_k}{\bar{u}_k}\right)$
Yes
$\varepsilon_j = \frac{\delta_j}{L_j}$
$\sigma_j = E_j \varepsilon_j$
$F_j = \sigma_j A_j$
$\cos\theta_j = \frac{x_j - x_i}{L_j}$ and $\sin\theta_j = \frac{y_j - y_i}{L_j}$

Fig 6. Side by side comparison of the different CA algorithms from our literature review. We included Conway's Game of Life on the top left for reference. For full page sized pictures, consult Appendix A.

The algorithms are presented in increasing order of complexity from left to right, up to bottom. As we can see Game of Life (a) is farily straighforward, with just a parallel update rule with a single computation over the neighbors and a condition. The cloud simulation automata (b) is very similar to GoL with the distintion that it has more state variabes (horizontal and vertical wind speed, position x and y, pressure and temperature, and thus the update rule performs a couple more of calculations. Peat wildfire (c) has as well a single update rule, however, it 2 "parallel" state variables, so it has more discrete state combinations, resulting in a branch. Nonetheless, this 2d state can be mapped into a single 1d state of more values. Additionally, it makes use of random numbers to decide if an update should be taken in place or not. Finally, Struss Structure Optimization CA (d) is severly more complex than the previous, both in formula definitions as well as step logic. This is because it performs multiple methods, for optimizing and evaluating the automaton. However, it can be divided into single state grid, with sequential CA runs over it, each of them performing a different method.

V. PUCCA Framework

GPU CA Implementation

From the previous analysis we can identify 2 differences between each automaton: cell state defintion variables & field global data, and update rule logic.

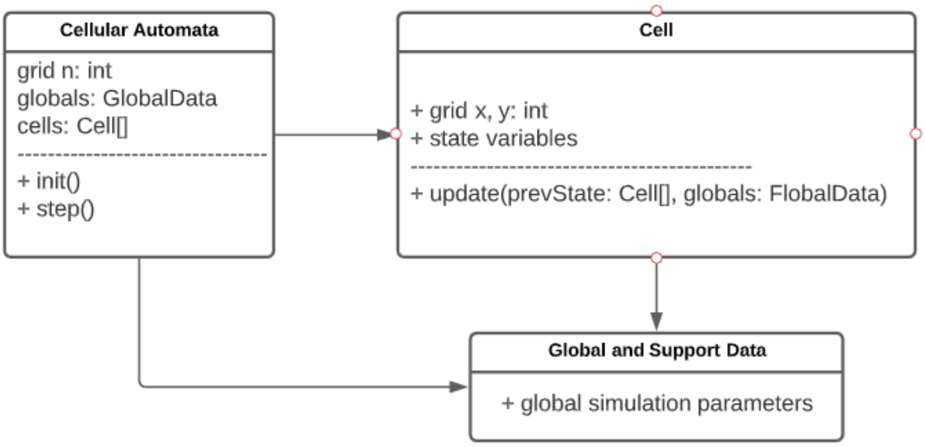
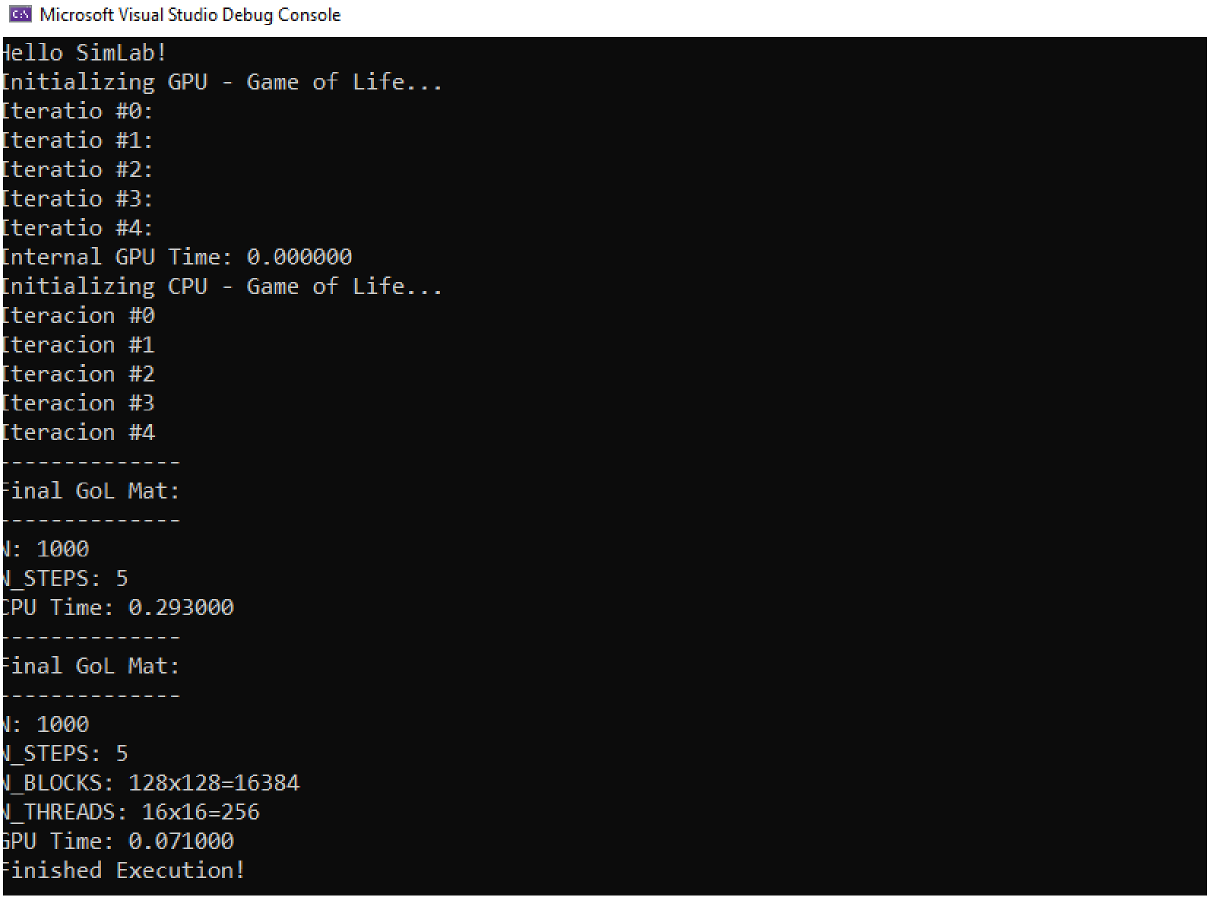


Fig 7. A UML Class diagram to represent a Cellular Automata. On the left is common cellular automata operations that could be implemented identically to work with CUDA, on the right we have user defined cell data, global data and rules.

An object oriented approach was first tried to develop a small CUDA C/C++ library as classes are supported within device code as it is demonstrated in [i]. However, using OOP techniques like polymorphism is generally not allowed due to the implementations details of C++ that stores a virtual member funtion table in memory of the CPU for CPU instantiated objects, thus it can't be accesed from device code. Also there are severly performance hits when using class casting and method invocations. Our cellular automata's performance was degraded by a factor of ⅓,

For these reasons, a simpler approach was taken, to provide a code template implementing a cellular automata, and guide the user to modify only those sections concerning state definition and update logic. The implementation of GoL was done a served as a sample to implement new automata.

The model was loaded and executed in the file "TestSimLab.cpp". To compile it is recommended to use Visual Studio with the NVIDIA extension, as it handles every object compilation and linking automatically. However, if it were to proceed manually, c++ files would have to be partially compiled into .obj files with C/C++ system compiler. And CUDA .cu files have to be compiled to .obj files with CUDA *nvcc* compiler. Then we can link all the produced .obj files again with *nvcc* into our final executable.



```
Hello SimLab!
Initializing GPU - Game of Life...
Iteratio #0:
Iteratio #1:
Iteratio #2:
Iteratio #3:
Iteratio #4:
Internal GPU Time: 0.000000
Initializing CPU - Game of Life...
Iteracion #0
Iteracion #1
Iteracion #2
Iteracion #3
Iteracion #4
-------------
Final GoL Mat:
-------------
N: 1000
N_STEPS: 5
CPU Time: 0.293000
-------------
Final GoL Mat:
-------------
N: 1000
N_STEPS: 5
N_BLOCKS: 128x128=16384
N_THREADS: 16x16=256
GPU Time: 0.071000
Finished Execution!
```

Fig 8. Output results from the TestSimLab.cpp program evaluating GoL automaton.

We will refer to as this process of isolating only the specific state data and update rule of a cellular automata and coding into into CUDA using a code template file as PUCCA's Parallel Cellular Automaton Implementation phase.

Integration with external MASON visualization tool

Now, this is not very useful for researchers as none of them want to only see how much time it took to simulate a phenomena, or watch a console output representation of such simulation. That's why it's enhanced with the integration of the external simulation visualization tool MASON.

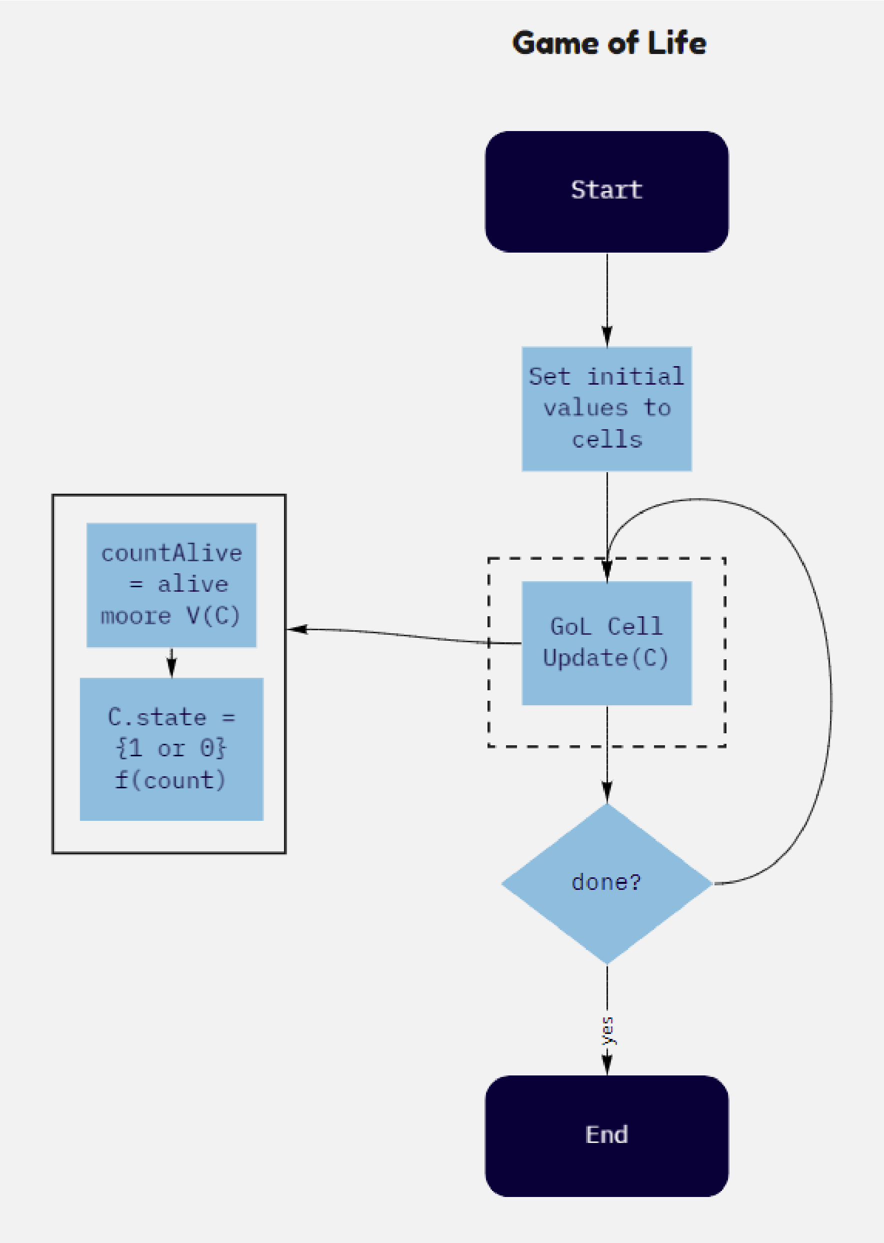
MASON is a jave framework that provides many programming structures and functions to easily model a phenomena and run, control, evaluate, and extract results from it in a visual manner. It is highly extensible and very powerful. However, it does not have support using CUDA. Also, C/C++ and Java are two completely different programming languages. Luckly, Java has JNI (Java Native Interface) for programming bridges between code executing in the JVM and machine code compiled functions.

VI. Results

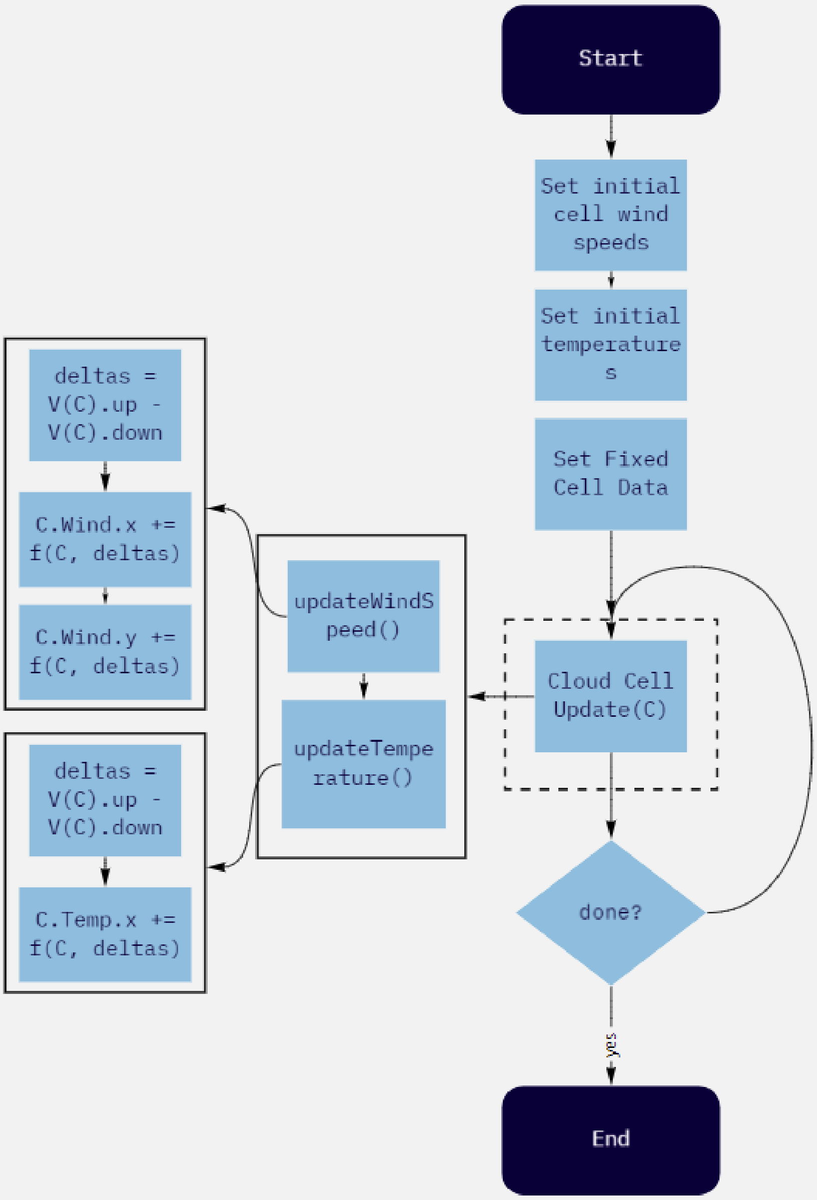
VII. Discussion

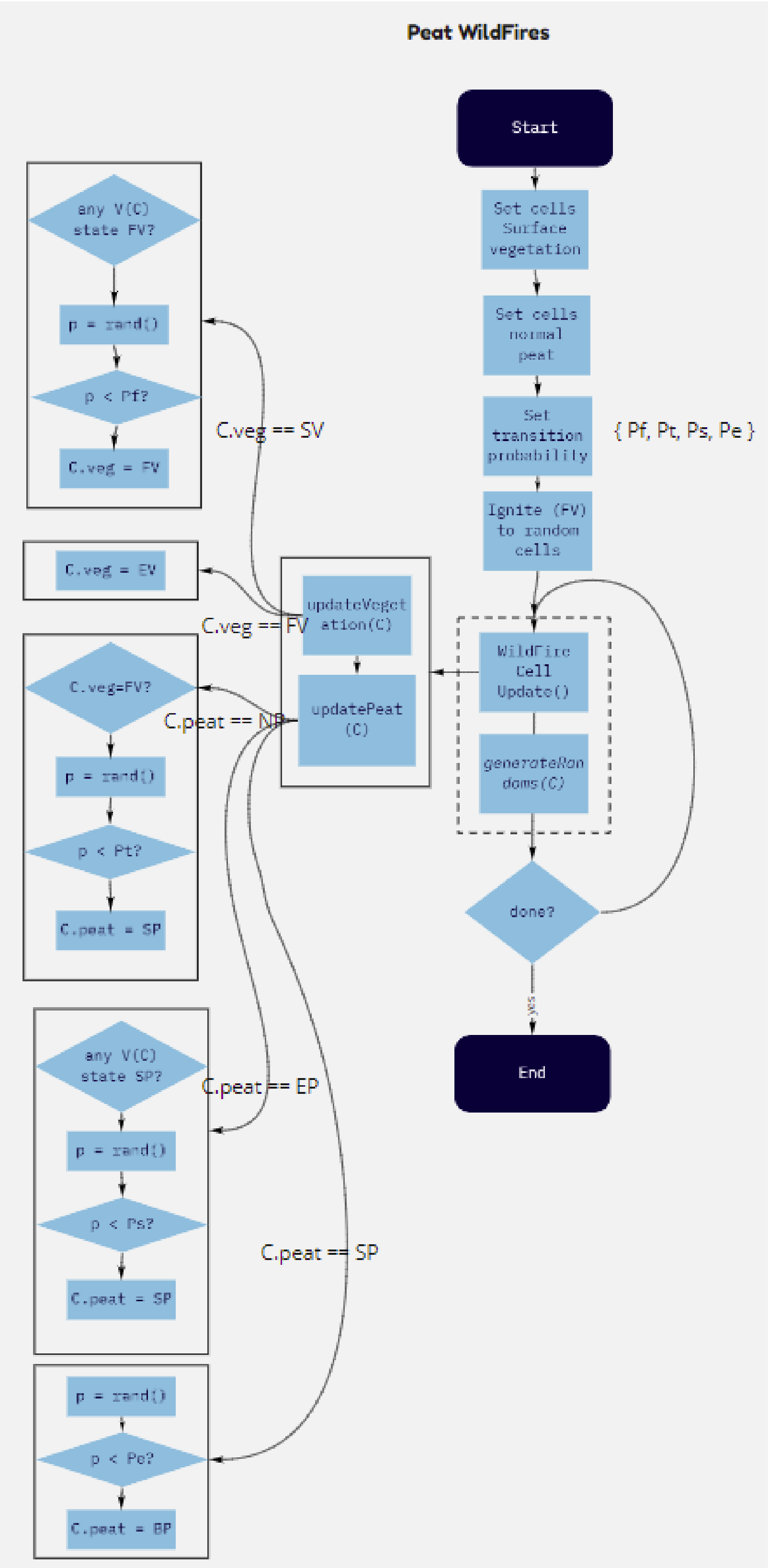
VIII. Conclusion and Future Work

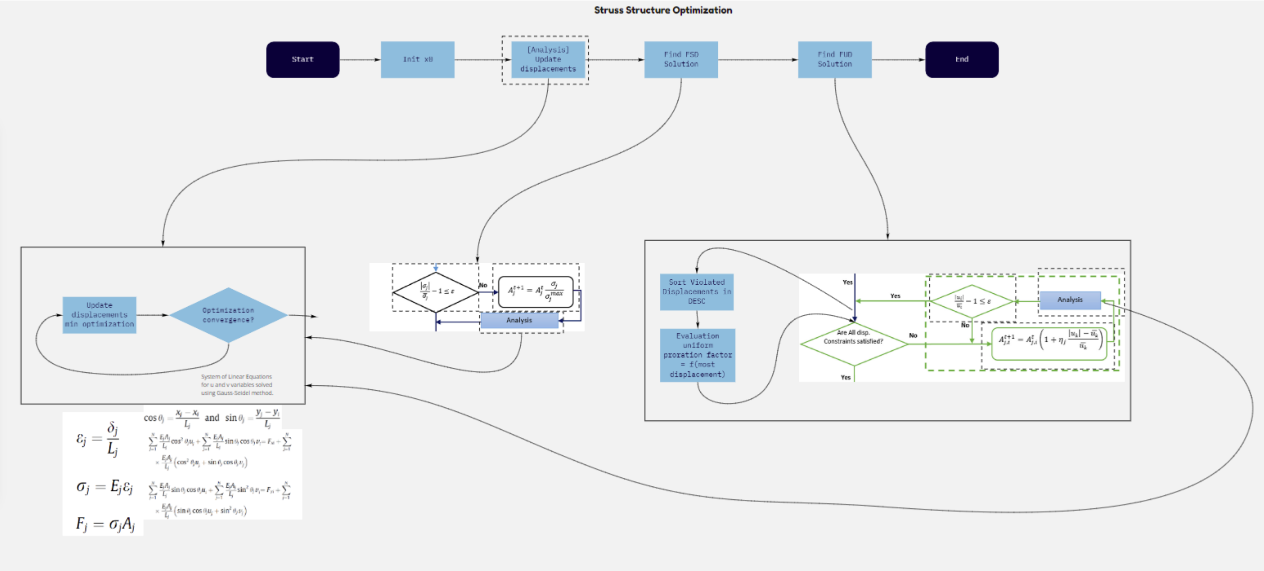
IX. Appendix A


Game of Life
Start
Set initial values to cells
countAlive = alive moore V(C)
C.state = {1 or 0} f(count)
GoL Cell Update(C)
done?
yes
End

Clouds Dynamics




Peat WildFires
Start
Set cells Surface vegetation
Set cells normal peat
Set transition probability
{ Pf, Pt, Ps, Pe }
Ignite (FV) to random cells
WildFire Cell Update()
generateRandoms(C)
done?
yes
End
updateVegetation(C)
updatePeat(C)
C.veg == SV
C.veg == FV
C.peat == NP
C.peat == EP
C.peat == SP
any V(C) state FV?
p = rand()
p < Pf?
C.veg = FV
C.veg = EV
C.veg=FV?
p = rand()
p < Pt?
C.peat = SP
any V(C) state SP?
p = rand()
p < Ps?
C.peat = SP
p = rand()
p < Pe?
C.peat = BP


Struss Structure Optimization
Start
Init x0
[Analysis] Update displacements
Find FSD Solution
Find FUD Solution
End
Update displacements min optimization
Optimization convergence?
System of Linear Equations for u and v variables solved using Gauss-Seidel method.
$\frac{|\sigma_j|}{\bar{\sigma}_j} - 1 \leq \varepsilon$
No
$A_j^{t+1} = A_j^t \frac{\sigma_j}{\sigma_j^{max}}$
Analysis
$\varepsilon_j = \frac{\delta_j}{L_j}$
$\sigma_j = E_j \varepsilon_j$
$F_j = \sigma_j A_j$
$\cos\theta_j = \frac{x_j - x_i}{L_j}$ and $\sin\theta_j = \frac{y_j - y_i}{L_j}$
Sort Violated Displacements in DESC
Evaluation uniform proration factor = f(most displacement)
Yes
Yes
Are All disp. Constraints satisfied?
No
Yes
$\frac{|u_i|}{\bar{u}_i} - 1 \leq \varepsilon$
No
Analysis
$A_{j,i}^{t+1} = A_{j,i}^t \left(1 + \eta_j \frac{|u_k| - \bar{u}_k}{\bar{u}_k}\right)$

X. References

Bibliography

[1] Automata networks in computer science
[2] Cellular Automata DEVS: A Modeling, Simulation, and
Visualization Environment
[3]A novel model to simulate cloud dynamics with cellular automaton
[4]Using cellular automata to simulate field-scale flaming and smouldering wildfires in
tropical peatlands
[5] A non-uniform cellular automata framework for topology and sizing
optimization of truss structures subjected to stress and displacement
constraints
[6] The Java™ Native
Interface
[7] Programmer’s Guide and Specification
Multiagent Simulation
And the
MASON Library
[8]
https://web.archive.org/web/20100718140020/http://www.swarm.org/images/1/10/How-to_set_up_%26_use_Eclipse_with_Mason.pdf

[9] Best Practices Guide :: CUDA Toolkit Documentation (nvidia.com)
[10] CUDA By Example An Introduction to General-Purpose GPU Programming
[11] Separate Compilation and Linking of CUDA C++ Device Code Separate Compilation and
[12] Linking of CUDA C++ Device Code | NVIDIA Developer Blog
[c] Computer & Generations.pdf (dspmuranchi.ac.in)
[d] What is a GPU and do you need one in Deep Learning? | by Jason Dsouza | Towards Data Science

Image Convolution From Scratch. Mathematical operation on two functions… | by Sameeruddin Mohammed | Analytics Vidhya | Medium
It From Bit: Is The Universe A Cellular Automaton? | by Paul Halpern | Starts With A Bang! | Medium
Conway's Game of Life in python. A simple implementation of Conway's… | by Robert Andrew Martin | Medium