Middle East Technical University
Multimedia Informatics Department

MMI 713

CONCEPTUAL DESIGN REPORT

Solving N-Queens Problem Using Cell Assemblies on GPU

Ugurcan Cakal

April 26, 2020

Contents

List of Figures

List of Tables

Preface

Hereby in this report, the conceptual design of the project ”Solving N-Queens Problem Using Cell Assemblies on GPU” has been declared. Algorithm descriptions, memory management, parallelization of the modules, and performance analysis plans have been examined. The main objective of the project is to design a biologically plausible constraint satisfaction problem solving neural network which is composed of cell assemblies. Since this report is a follow-up of the proposal report, problem statement and literature review parts are omitted. The reader may request[1] if desired.

1 Design Overview

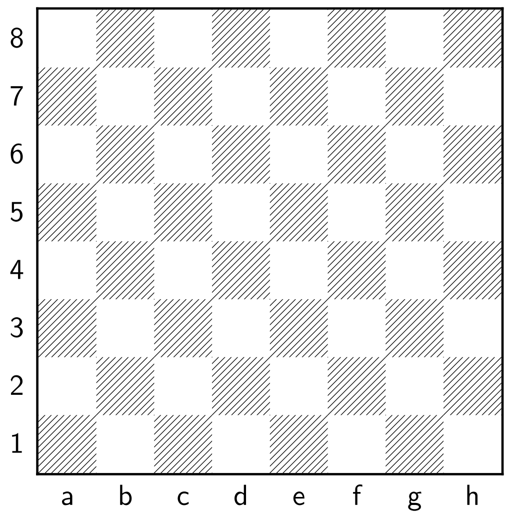
So far, it has been shown how many different solutions exist for n-queens problem up to n = 27. It is known that their solutions exist for n>27, but no one has ever proved the exact number of solutions. In the work [2], the numbers coming from different researches have been collected, and the results can be seen in Table 1.

Table 1: Number of Solutions for N-Queens Problem Given Different N Values

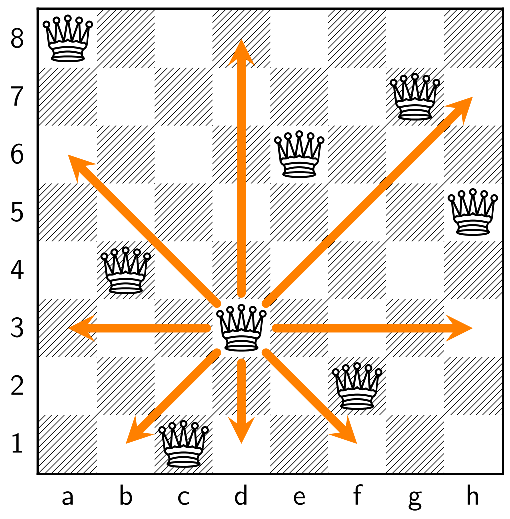
N	Number of Solutions
1	1
2	0
3	0
4	2
5	10
6	4
7	40
8	92
9	352
10	724
11	2.680
12	14.200
13	73.712
14	365.596
15	2.279.184
16	14.772.512
17	95.815.104
18	666.090.624
19	4.968.057.848
20	39.029.188.884
21	314.666.222.712
22	2.691.008.701.644
23	24.233.937.684.440
24	227.514.171.973.736
25	2.207.893.435.808.352
26	22.317.699.616.364.044
27	234.907.967.154.122.528

Nearly exponential and a-rhythmic increase in the number of solutions shows us that finding the exact number of solutions for any n requires infinite computational power. However, solving the problem for any n is possible. Since this project is more related to mimicking the human decision process, finding the exact number of different solutions is not a requirement. Investigating the synaptic connection pattern could be asserted as the primary target.

Let's examine the 8-queens problem, which is represented on a conventional chessboard. Each queen needs to be placed in a way that no queen attacks any other. Figure 1 shows an empty board and the a solution.



(a) Empty Chessboard



(b) 8-Queens Problem

Figure 1: Chessboard Representation

The first step through the overall design is to represent the chessboard using cell assemblies. For the sake of simplicity, each row of the chessboard is represented by a group of CA and each column number is encoded in binary. Since each row need to have exactly 1 queen, the place of the queen is represented by the column number in binary encoded form. For a cell assembly, 1 means ignition, and 0 means idle state. Therefore 3 consecutive CAs can represent 3 bit information. Figure 2 visualizes the CA Board design. Figure 2a is a translation of the chessboard in Figure 1a. In addition, Figure 2b represents the same solution in Figure 1b.

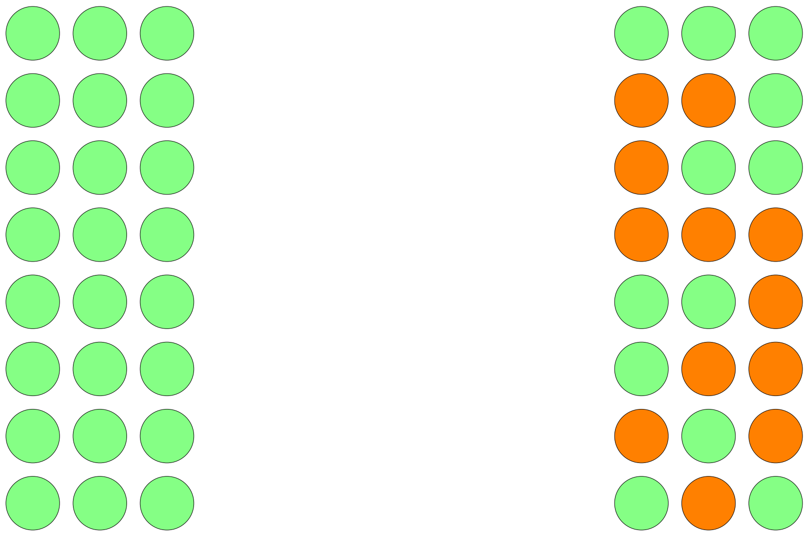

(a) Idle CA Board (b) CA Board with an 8 Queens Solution

Figure 2: CA Board Representation

The second step is to build the binary CA model using artificial spiking neurons. In a neural network, a group of neurons could be considered as cell assembly provided that they have a high mutual synaptic connectivity. For now, let's say a binary CA consists of 80% excitatory and 20% inhibitory neurons. The resulting model will look like the one in Figure 3.

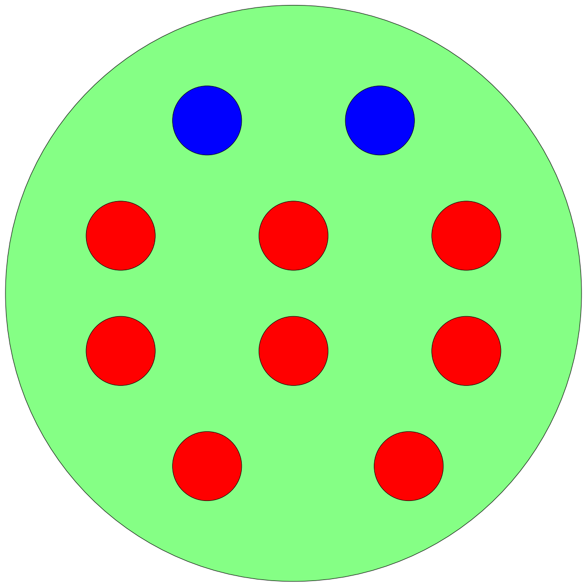


Figure 3: Binary CA (A simplified version of an actual CA in the brain) with blue circles representing inhibitory neurons and red ones representing excitatory neurons

The ignition activity of the cell assembly will be decided considering the collective activity of the neurons inside the CA. For a CA to be ignited, the average spiking activity is needed to be above a certain activation threshold. Also, the state needs to stay steady for a measurable time duration [3]. Figure 4 shows an imaginary raster plot representing the activity of the neurons inside a CA for 10 time units. The inhibitory neurons in Figure 3 have been assigned with blue ids and the excitatory neurons in Figure 3 have been assigned with red ids.

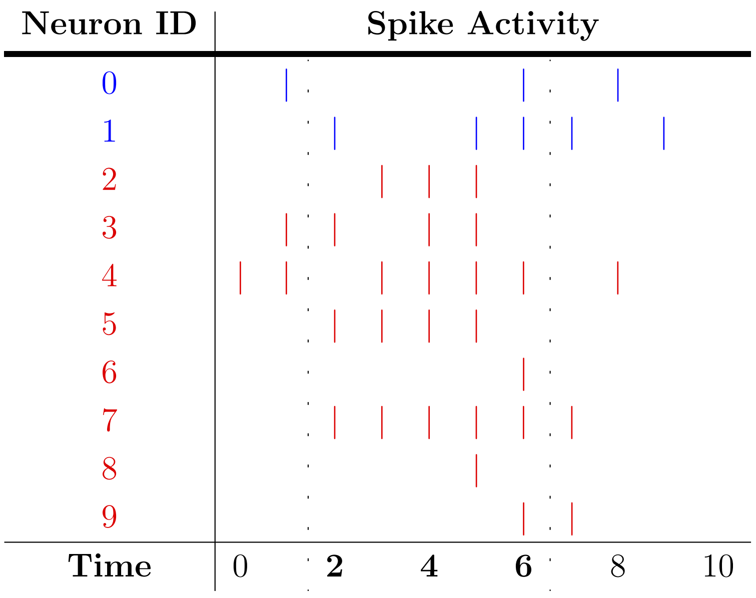


Figure 4: A Conceptual Raster Plot

It could be seen in the raster plot that the spiking activity of the inhibitory neurons results in fading out of some neurons. On the other side, the spiking activity of the excitatory neurons results in the firing of more neurons. The decision of the ignition is highly dependent on the threshold chosen. For example, if the threshold for ignition is set to be 30%, it could be said that the CA is active between t=2 and t=6.

The last step is to build a system to manage the operation. For this step, the stochastic meta control process implemented by Belavkin and Huyck for conflict resolution and learning probability matching in a neural cell-assembly architecture [4] has been inspired. The block diagram visualizing the overall system can be seen in Figure 5.

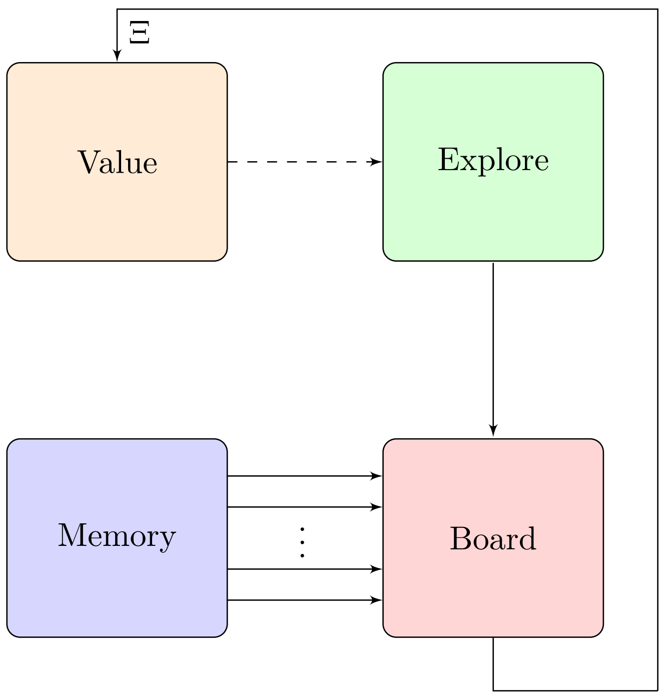


Figure 5: Overall Block Diagram of the System

There are four main modules in the system; value, explore, board, and memory. The memory and board modules can be considered as sets consisting of different number of CAs. Let's define memory as $M = \{m_0, m_1, ..., m_m\}$, and board as $B = \{b_0, b_1, ..., b_n\}$. Initially there are excitatory connections from every CA in M to all CAs in B, which means that all pairs (m, a) are equally preferred. The purpose of value and explore modules is to make the process selective according to the feedback and its fitness.

The value and explore modules are not sets of CAs but sets of neurons. The output activity of the value module represents the fitness value associated with the set of pairs, in other words, a synapse set $S = \{(m_0, b_0), (m_1, b0), (m_3, b_1), \ldots, (m_m, b_n)\}$ selected on previous step. On the other hand, the explore module aims to randomize the activity on the CA board. The explore module includes neurons which could fire independent of external stimulation.

The neurons in the explore module sends excitatory signals to all CAs in board module. This result in randomly triggered responses in the board module. However, the value module sends inhibitory connections to the explore module, so that high activity of the value cells shut down the activity of the explore cells. As a result, any response

CA that has been ignited in module B will persist longer, because it is less likely to be shut down by another CA.

Such a connectivity implements the following learning scheme: If a synapse set results in expressing an acceptable n-queen solution, then the high activity of the value module inhibits the explore module, the responsible synapse set is allowed to persist longer, and the $m \rightarrow b$ connections increase relative to others due to Hebbian Learning.

2 Algorithm Description

This section includes detailed explanations of the modules referred in Section 1 : binary cell assembly, value, explore, board, and memory. Here, the algorithmic frame has been drawn without considering parallelization and memory management issues. Separate sections have been allocated for parallelization and memory management, Sec 4 and 3.

2.1 Binary Cell Assembly

Implementation of the computational cell assembly model is the backbone of the project. The biological plausibility requirement leads the underlying neuron model to become computationally complex. In this project, fatigue leaky integrate and fire(FLIF)[5] model will be used because that Kaplan et al. have proposed FLIF as a neuron model for cell assembly construction.

In the project proposal report [1], the mathematical frame of binary cell assembly and FLIF neurons has been drawn. In this section, implementing an algorithm for the realization of the model is the main target. Since matrix operations are well suited for further parallelization, all necessary parameters and variables are represented in matrix or vector form.

Let's say C is a vector storing constant parameters:

- θ : Firing threshold
- d : Decay constant
- F^r : Recovery constant
- F^c : Fatigue constant

- $\alpha \in [0, 1]$: learning rate
- W_B : constant representing average total synaptic strength of the pre-synaptic neuron.
- W_i : current total synaptic strength

Φ, E, F, X, Y are vectors and Ω is the matrix storing:

- Φ : Firing flags
- E : Energy Levels
- F : Fatigue levels
- Ω : Connection weights
- X : Pre-synaptic Firing Flags
- Y : Post-synaptic Firing Flags

And parameters n, i, o defining the number of:

- n : neurons inside the cell assembly
- i : external neurons having connections from outside to inside of the CA
- o : external neurons having connections from inside to outside of the CA

Then the data segment of a CA can be presented as follows:

$$C = \begin{bmatrix} \theta & d & F^r & F^c & \alpha & W_B & W_i \end{bmatrix}$$
$$\Phi = \begin{bmatrix} \phi_0 & \phi_1 & \dots & \phi_n \end{bmatrix}$$
$$E = \begin{bmatrix} e_0 & e_1 & \dots & e_n \end{bmatrix}$$
$$F = \begin{bmatrix} f_0 & f_1 & \dots & f_n \end{bmatrix}$$
$$\Omega = \begin{bmatrix} \omega_{00} & \omega_{01} & \dots & \omega_{0(n+o)} \\ \omega_{10} & \omega_{11} & \dots & \omega_{1(n+o)} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_{(n+i)0} & \omega_{(n+i)1} & \dots & \omega_{(n+i)(n+o)} \end{bmatrix}$$

$$X = \begin{bmatrix} x_0 & x_1 & \dots & x_{n+i} \end{bmatrix}$$
$$Y = \begin{bmatrix} y_0 & y_1 & \dots & y_{n+o} \end{bmatrix}$$

Since Φ, E, F, X, Y vectors, and Ω matrix are dynamic structures, an update policy needs to be defined to apply in each iteration. Update policies are grouped in 4 considering dependencies. There are 4 update phases for Φ, E & F, Ω, and X & Y as follows:

Update Φ:

$$\phi_b(t) = \begin{cases} 1 \text{ if } E_b(t) - F_b(t) \geq \theta \\ 0 \text{ otherwise} \end{cases}$$

Update E and F:

$$\begin{aligned} &\textbf{if } \phi_b(t) = 0 \ : \\ &E_b(t+1) = \frac{1}{d}E_b(t) + \sum_{a=1}^{n} \omega_{ab} \times \phi_a(t) \\ &F_b(t) = max\{0, F_b(t-1) - F^r\} \end{aligned}$$

$$\begin{aligned} &\textbf{if } \phi_b(t) = 1 \ : \\ &E_b(t+1) = \sum_{a=1}^{n} \omega_{ab} \times \phi_a(t) \\ &F_b(t) = F_b(t-1) + F^c \end{aligned}$$

Considering $||$ as the concatenation operator, update X and Y:

$$X = \Phi_n(t)||\Phi_{in}(t)$$
$$Y = \Phi_n(t+1)||\Phi_{out}(t+1)$$

Update Ω :

$$\Delta\omega_{ij} = \begin{cases} \alpha(1-\omega_{ij})e^{W_B - W_i} & \text{if } x_t = 1,\ y_t = 1 \\ -\alpha\omega_{ij}e^{W_i - W_B} & \text{if } x_t = 1,\ y_t = 0 \end{cases}$$

Using all together, whether the CA is in ignition or not is decided. The rate of total number of firing neurons over all neurons in the CA need to be passed in order to be in ignition state.

Lets say:

- A : CA
- |A| : number of neurons in A
- Θ : activation threshold for A
- t : time step

$$\Theta < \frac{\sum_{a\in A} \phi_a(t)}{|A|}$$

Define Θ' for the sake of simplicity as :

$$\Theta' = \Theta - \frac{\sum_{a\in A} \phi_a(t)}{|A|} > 0$$

Flowchart in Figure 6 summarizes the CA algorithm.

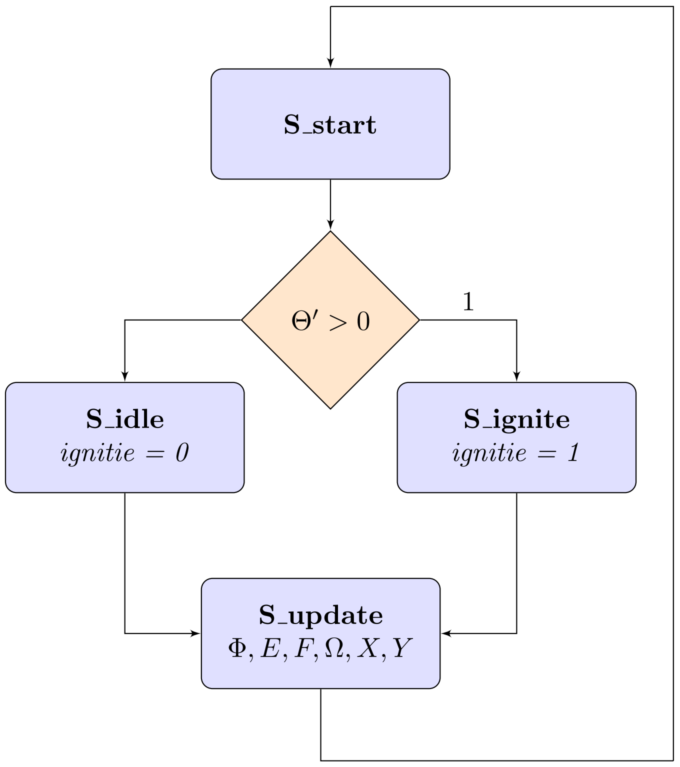


Figure 6: Algorithmic Flowchart of CA Model

If the Θ' value is greater than 0, then ignite, else stay idle. After each decision, update firing flags, energy levels, fatigue levels, connection weights, pre-synaptic firing flags and post-synaptic flags. Then go to the starting state and check again in an endless loop.

2.2 Value

The value module does the evaluation of the last chessboard appearance. It's composed of connection neurons and a calculation unit. The module takes the chromosome as an input from the board module, calculates the fitness value, and represents how much the system is satisfied with the placement via its output activity. For example, if the encoded n-queen solution satisfies the constraints, then the neurons of the value module fire a relatively high rate. This results in inhibition of the explore module neurons and leads the memory and board synapses to be stronger. On the other hand, if the encoded n-queen solution is far from satisfying the constraints, then the neurons of the value module fire a relatively low rate. This results in a more random activity of the explore module neurons and leads the memory and board synapses to be weaker.

The operation of the value module could be more realistic if the calculation would also be embedded in the neural network and the fitness function would be replaced by a utility function that would be learned through the iterations. However, since the purpose of this project is to exploit the parallel nature of the GPUs and to optimize memory management, the calculation unit is defined externally. Pseudocode of the fitness calculation algorithm is given in Algorithm 1.

Algorithm 1 Fitness

```
Require: $\Xi = \begin{bmatrix}\xi_0 & \xi_1 & \dots & \xi_n\end{bmatrix}$
1: procedure NQueenFitness
2:     $collision \leftarrow 0$
3:     for $i \leftarrow n \; ; \; i \geq 0 \; ; \; i--$ do
4:         for $k \leftarrow i-1 \; ; \; k \geq 0 \; ; \; k--$ do
5:             $d \leftarrow |\xi_i - \xi_k|$
6:             if $(d = 0) \; || \; (d = i - k)$ then
7:                 $collision++$
8: return $collision$
```

The rest of the value module is a network of sparsely, recursively, and randomly connected artificial neurons. Different from the original cell assembly model, weights are not subject to change because the weight update is not necessary. The activity control is done by feeding the network with a dummy input vector.

$$\Phi_{dummy} = \begin{bmatrix}\Phi_0 & \Phi_1 & \dots & \Phi_m\end{bmatrix}$$

The output of the fitness function will be between $\left[0, \frac{n \cdot (n-1)}{2}\right]$ and 0 means best fit. According to this, if the fitness value is 0, then an input vector including all ones $\Phi_{dummy} = \begin{bmatrix}1 & 1 & \dots 1\end{bmatrix}$ will be fed to the network. On the contrary, if the fitness value is $\frac{n \cdot (n-1)}{2}$, then an input vector including all zeros $\Phi_{dummy} = \begin{bmatrix}0 & 0 & \dots 0\end{bmatrix}$ will be fed to the network. For the outputs between minimum and maximum possible values, a vector with corresponding fill rate:

$$\Upsilon_{\Phi} = 1 - \frac{2 \cdot \text{fitness}(\Xi)}{n \cdot (n-1)}$$

will be generated randomly.

Since the neuron level configuration is the same as in the binary cell assembly model, the equations and policies are not repeated. The reader may revisit the Section 2.1 for equations. Different from the binary CA, the weight update will be skipped. Flowchart in Figure 7 summarizes the value module operation.

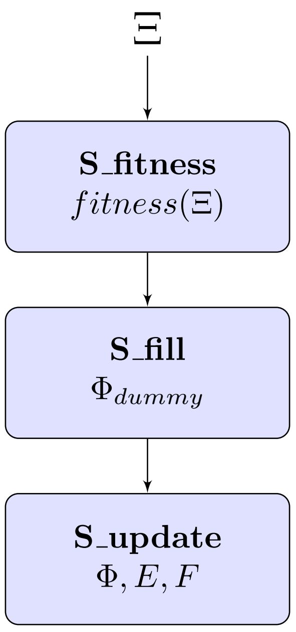


Figure 7: Algorithmic Flowchart of Value Module

Asynchronous chromosome feed initiates the operation. S_fitness state calculates the fitness value of the current chessboard. Using the fitness(Ξ) value, Φ_{dummy} vector is filled according to fill rate, Υ_{Φ}. After filling the input vector, the update state takes place and result in the transmission of the signals internally and through the explore module.

2.3 Explore

The explore module helps the board module represent random states in the case that it has not been suppressed by the value module. The module has incoming inhibitory synapses coming from the value module and outgoing excitatory connections to the board module. The firing activity depends highly on the firing flags resides in the value module. If there are too many active flags at the value side, the explore module cannot fire enough to stimulate the board CAs.

The operation of the explore module is similar to the operation of the value module. The explore module is also a network of sparsely, recursively, and randomly connected artificial neurons. Weights are not subject to change because the weight update is not necessary. The activity control is done by feeding the network with a dummy input vector.

$$\Phi_{dummy} = \begin{bmatrix} \Phi_0 & \Phi_1 & \dots & \Phi_m \end{bmatrix}$$

Different from the value module, the dummy vector is filled using a random fill rate. The neuron-level configuration is the same as in the binary cell assembly model. The reader may revisit the Section 2.1 for equations. Flowchart in Figure 8 summarizes the explore module operation.

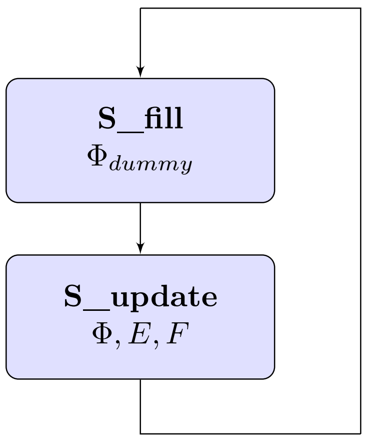
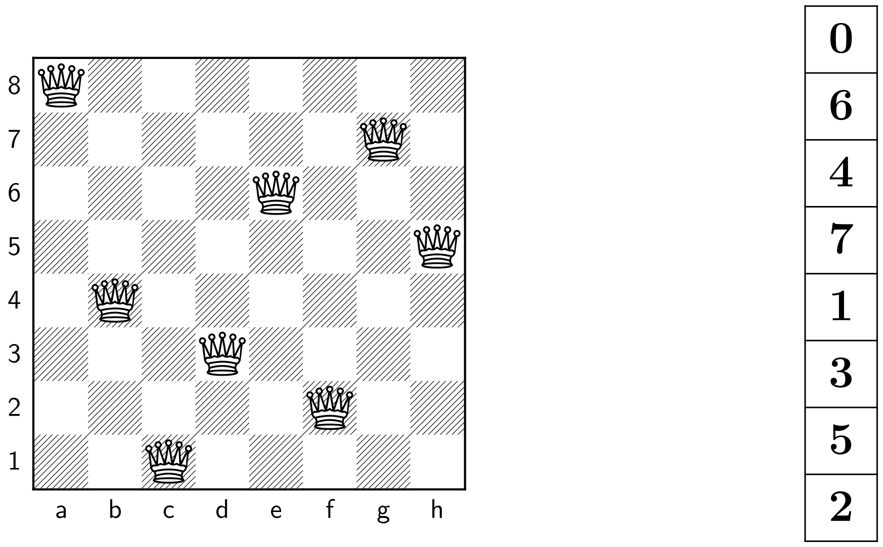


Figure 8: Algorithmic Flowchart of Explore Module

The dummy vector is filled using a random fill rate $\Upsilon_\Phi \in [0,1]$. After filling the input vector, the update stage takes place and firing flags, energy levels, and fatigue levels vectors are updated. This results in random stimulations in the board module if there is enough firing.

2.4 Board

Board module represents a candidate solution for the n-queens problem. Instead of representing each square with a CA, the board module uses a more lightweight and compressed version. Each row is represented by a group of CAs. On the other hand, each column is defined by $\lceil log_2 n \rceil$ number of bits in the CA form. Let's look at Figure 9. An 8-queen solution is encoded in a vector Ξ, called chromosome. While each cell of the chromosome holds the column number, the row information of the chessboard is preserved by using the exact amount of cells in the same order.



(a) An 8-queen Solution (b) Representative Chromosome

Figure 9: Encoding an 8-queen Solution

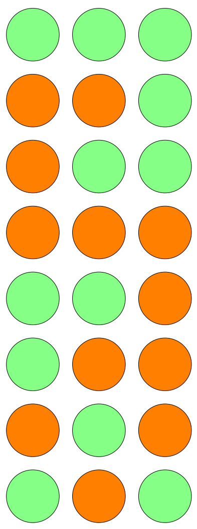
Decimal representation cannot be expressed by binary cell assemblies directly. Therefore, the decimal chromosome needs to be converted to a binary equivalent. Then the binary chromosome can be expressed by a group of CAs directly. Each cell assembly stands for one bit in the binary chromosome. Considering the ignition situation of a cell assembly as 1 and idle as 0, the binary chromosome can be represented by a group of CA as in Figure 10.

0
6
4
7
1
3
5
2

(a) Decimal Chromosome

000
110
100
111
001
011
101
010

(b) Binary Chromosome



(c) Binary Encoded CA Board

Figure 10: Representing an 8-queen Solution on CA Board

Let's say B is a matrix storing ignition status of the board CAs:

$$B = \begin{bmatrix} b_{00} & b_{01} & \dots & b_{0\lceil log_2 n \rceil} \\ b_{10} & b_{11} & \dots & b_{1\lceil log_2 n \rceil} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n0} & b_{n1} & \dots & b_{n\lceil log_2 n \rceil} \end{bmatrix}$$

In each cycle of a CA unit, the ignition flag will be updated. Along with this, the B vector needs to be updated though. B update is straightforward because each CA has already been storing their ignition status. Also, in each iteration, the decimal coded chromosome needs to be fed to the value module for feedback. The chromosome Ξ is a vector holding the decimal equivalents of the binary encoded values in the CA board.

$$\Xi = \begin{bmatrix} \xi_0 \\ \xi_1 \\ \vdots \\ \xi_n \end{bmatrix}$$

Considering all these, Figure 11 summarizes the board module operation.

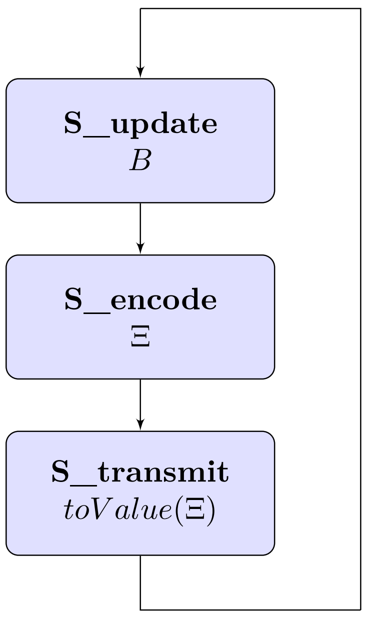


Figure 11: Algorithmic Flowchart of Board Module

The loop starts with update state. This state is depended on ignition flags coming from different CAs. Then the outlook of the CA board need to be encoded in a chromosome to be transmitted to the value module. The loop never ends unless externally forced to.

2.5 Memory

The memory module consists of a number of cell assemblies more than the board module has. The main purpose of the existence of the memory module is to sustain the connectivity which results in true choices. Initially there exist excitatory connections from every CA in memory to all CAs in board. As time went by, some connections get stronger due to the correlation between the ignition of a CA in M and the ignition of a CA in B. Similarly, some connections get weaker due to the anti-correlation. The value module promote the connections which lead to an 8-queen solution by suppressing the explore module. Observation of decreased activity of explore module implies that the system has developed useful synapses.

Since the synapses and signal transmission are handled in the CA level, memory module holds only the ignition status of the CAs. Let's say M is a vector storing ignition flags of the memory CAs.

$$M = \begin{bmatrix} m_0 & m_1 & \dots & m_m \end{bmatrix}$$

In each cycle of a CA unit, the ignition flag will be updated. Along with this, the M vector needs to be updated though as it is the case in the board module. M update is straightforward because each CA has already been storing their ignition status. Figure 12 summarizes the board module operation.

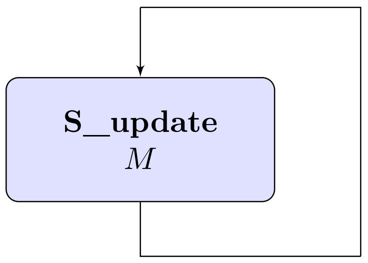


Figure 12: Algorithmic Flowchart of Memory Module

Memory module has a simpler algorithmic flow of operation considering the previous modules. It updates the M vector in an endless loop. The update state is depended on the ignition flags coming from different CAs inside the module. Therefore, data dependencies need to be handled carefully.

3 Memory Management

Before making an effort to parallelize the algorithms, it's worth spending some time on building an organized memory scheme. This section focuses on allocating memory space in an effective way for the data segment of the project. The memory usage hierarchy can be represented as in Figure 13

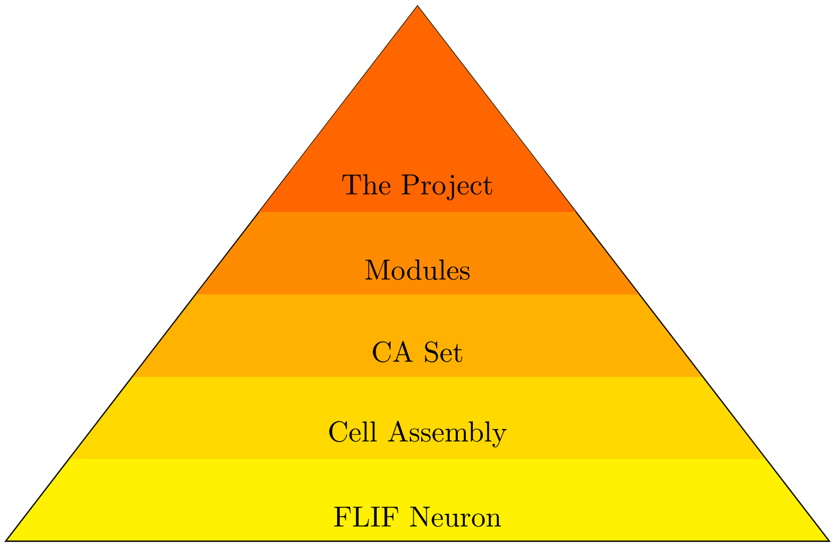


Figure 13: Memory Hierarchy of the Project

3.1 FLIF Neuron

FLIF neurons stay at the lower level since they only need 3 values: firing flag, energy level, and fatigue level; and a vector holding incoming connection weights.

$$\phi_x \ , \ e_x \ , \ f_x \ , \ \Omega_x = \begin{bmatrix} \omega_{0x} \\ \omega_{1x} \\ \vdots \\ \omega_{(n+i)x} \end{bmatrix}$$

During the update operation, these parameters will be used by a single GPU block. However, even if the referred values are enough to characterize a single neuron, not sufficient for doing any update operation. Those operations cannot be done without information coming from other neurons. Therefore both the shared memory and global memory will need to be accessed.

3.2 Cell Assembly

The second layer is stated as the cell assembly layer. Encapsulating multiple neurons, this layer holds single neuron related and inter-neuron connection related parameters and the ignition flag I. These parameters have been explained in Section 2.1 in detail.

$$I$$

$$C = \begin{bmatrix} \theta & d & F^r & F^c & \alpha & W_B & W_i \end{bmatrix}$$

$$\Phi = \begin{bmatrix} \phi_0 & \phi_1 & \dots & \phi_n \end{bmatrix}$$

$$E = \begin{bmatrix} e_0 & e_1 & \dots & e_n \end{bmatrix}$$

$$F = \begin{bmatrix} f_0 & f_1 & \dots & f_n \end{bmatrix}$$

$$\Omega = \begin{bmatrix} \omega_{00} & \omega_{01} & \cdots & \omega_{0(n+o)} \\ \omega_{10} & \omega_{11} & \cdots & \omega_{1(n+o)} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_{(n+i)0} & \omega_{(n+i)1} & \cdots & \omega_{(n+i)(n+o)} \end{bmatrix}$$

$$X = \begin{bmatrix} x_0 & x_1 & \dots & x_{n+i} \end{bmatrix}$$

$$Y = \begin{bmatrix} y_0 & y_1 & \dots & y_{n+o} \end{bmatrix}$$

Vectors and matrices represented above are necessary and sufficient for doing update operations. Since CAs are presented as binary decision units, firing flags of the neurons associated with the same CA reside in the same block in order to decrease the global memory access during the ignition decision. Although this approach reduces the global memory access, it could not stop completely. The reason for that is, there are incoming and outgoing connections of neurons associated with the CA from outside of the CA as in Figure 14. In addition, the ignition flag needs to be stored at the global memory to be able to be used reached by upper layers.

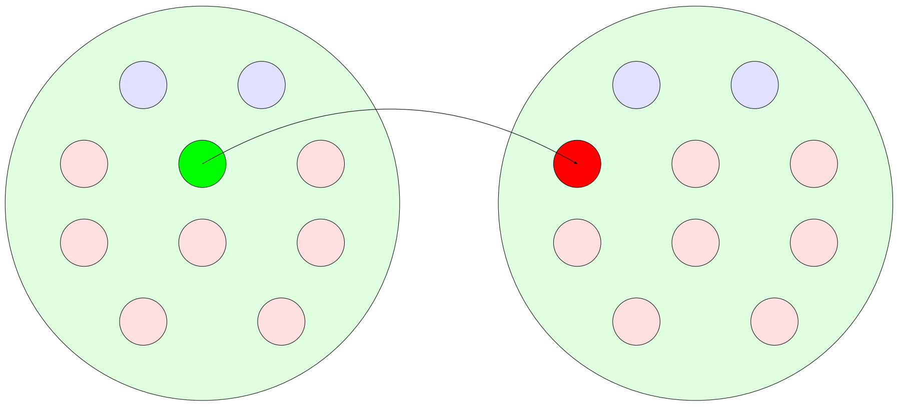


Figure 14: External Connection Case

The weight update requires the post-synaptic and pre-synaptic(green one) firing activity and done at the post-synaptic(red one) neuron side. In the case that a post-synaptic neuron has an external incoming connection and performs a weight update operation, it will need to reach the global memory to get the last pre-synaptic firing flag. In addition, it will notify the weight change to the pre-synaptic neuron which is outside of the CA via global memory.

Due to the small shared memory capacity, it's impossible to hold the global synaptic connection map in the block local memory. Only the CA associated neurons, along with their incoming and outgoing weights, can be stored in a matrix. Table 2 lists the CA related structures which are processed in global and shared memory. C, E, F, and Ω are stored in shared memory since they do not need to be in connection with outside of the CA. Φ, X, Y, and I have been placed to the global memory since upper layers, or other CAs will require the information presented.

Table 2: Shared and Global Memory Usage of a Cell Assembly

Shared	**Global**
C	Φ
E	X
F	Y
Ω	I

Since the connectivity regime is known to be sparse, the synaptic map can be stored in a sparse matrix form. For sparse data storage, Compressed Sparse Column (CSC) format will be used as illustrated below.

Dense Case:

$$\text{Matrix} = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

Sparse Case(CSC):

$$\text{column offsets} = \begin{bmatrix} 0 & 2 & 5 & 7 & 9 \end{bmatrix}$$

$$\text{row indices} = \begin{bmatrix} 0 & 2 & 0 & 1 & 3 & 1 & 2 & 2 & 3 \end{bmatrix}$$

$$\text{values} = \begin{bmatrix} 1 & 5 & 7 & 2 & 6 & 8 & 3 & 9 & 4 \end{bmatrix}$$

In this format, only the nonzero values of the matrix is stored along with column pointers and row indices. Even though it adds extra computational overhead to all operations, it uses much less memory if the matrix has a relatively high number of zeros. Since reducing memory usage is more beneficial than adding computational overhead while computing with GPUs, sparse matrix storage has been chosen over the dense storage.

3.3 CA Set

A CA set holds the ignition flags of a group of cell assemblies. Since the ignition flag is stored in the global memory at the previous step, the duty of a CA set is to find the flags and store them in an organized form. Since the host connection is not required, the set will stay at global memory.

$$S = \{s_0, s_1, \ldots, s_k\}$$

3.4 Modules

Two of the modules, value and explore, bypass the CA set layer in the hierarchy, and directly use neurons. Therefore, their memory management is similar to a Cell Assembly. However, there are two main differences. The first is that they do not need to store X and Y vectors since weights are not updated. Also, ignition status is irrelevant. Table 3 lists the related structures which are processed in global and shared memory.

Table 3: Shared and Global Memory Usage of Value and Explore Modules

Shared	**Global**
C	Φ
E	$\cdot$
F	$\cdot$
Ω	$\cdot$

The rest of the modules, board, and memory, use CA sets in different formats. The board module transforms the set to a 2-D matrix structure and the memory module transforms it into a 1-D vector format. The transformation does not have an effect on choosing a location for the sets. The formatted sets can also stay at the global memory.

$$B = \begin{bmatrix} b_{00} & b_{01} & \dots & b_{0\lceil log_2 n\rceil} \\ b_{10} & b_{11} & \dots & b_{1\lceil log_2 n\rceil} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n0} & b_{n1} & \dots & b_{n\lceil log_2 n\rceil} \end{bmatrix}$$

$$M = \begin{bmatrix} m_0 & m_1 & \dots & m_m \end{bmatrix}$$

Board module encodes the overview of the board in a chromosome Ξ structure to be shared with the value module.

$$\Xi = \begin{bmatrix} \xi_0 \\ \xi_1 \\ \vdots \\ \xi_n \end{bmatrix}$$

In order to make the Ξ vector reachable by the value module, it needs to be stored in the global memory. Moreover, since the chromosome representation is a lightweight way to represent the chessboard, it may also be used by a human observer to monitor the operation. In this case, unified memory, which is announced along with CUDA 6 is a better option.

4 Parallelization

Looking at the big picture, the whole system seems to be a neural network that has some specialized units. These specialized units are called cell assemblies and most of the computational work is in the responsibility of these structures. Since the neurons inside a CA and different CAs together can make progress simultaneously, the problem suits well for parallelization. However, considering the nature of the n-queens problem, data dependencies need to be handled carefully. This section deals with the parallelization of the system and the issues that need to be taken into consideration while doing parallelization, in 2 branch: update operations and fitness calculation.

4.1 Single Neuron Update

In Section 2.1, the update policy of a neuron has been depicted by using matrix and vector notations. Considering this, the reader can realize that all update operations can be reduced to matrix operation. This makes it easier to parallelize.

In Figure 15, a simplified connectivity map of a neuron inside a binary CA has been drawn. This the figure, incoming connections are colored with brown and outgoing connections are colored with blue. In the naming of the weights, the neurons inside the CA has been prioritized. That means, first 10 weight names are allocated for internal connections because there are 10 neurons inside the CA. Now, let's examine all update operations which requires a computational work: Φ, E, F, Ω one by one.

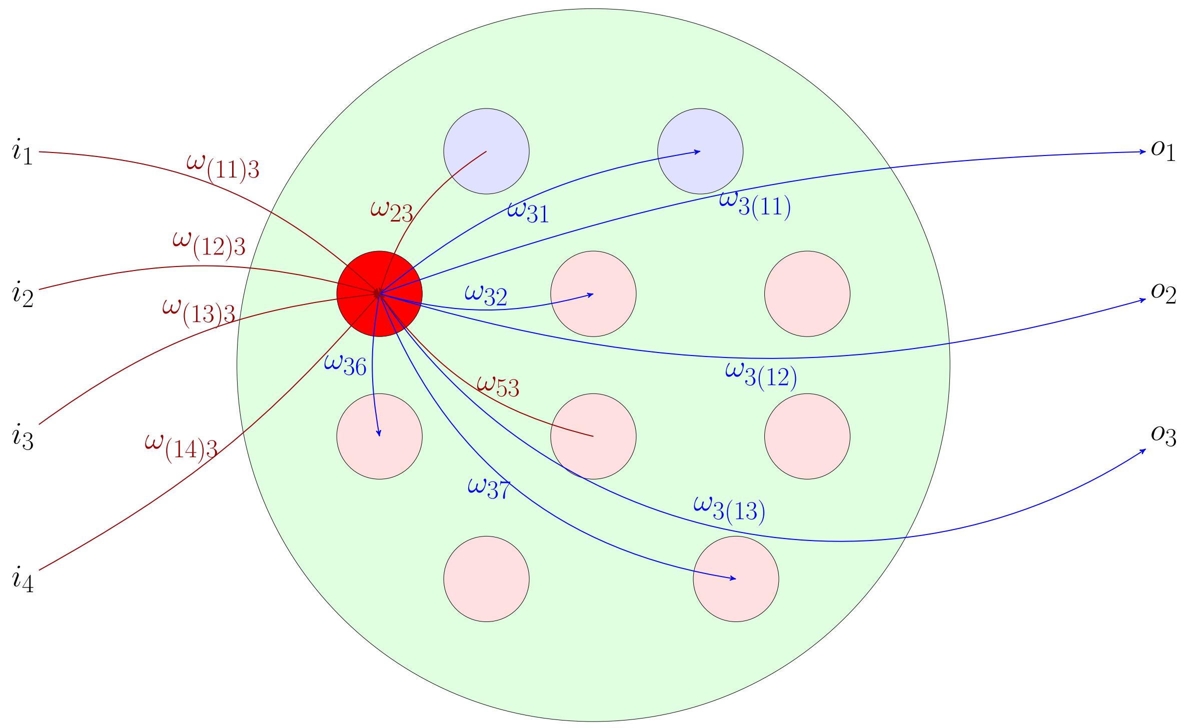


Figure 15: Connectivity Map of a Neuron Inside CA

4.1.1 Update Φ

Φ update requires only matrix addition operation and sign comparison.

$$\phi_b(t) = \begin{cases} 1 \text{ if } E_b(t) - F_b(t) \geq \theta \\ 0 \text{ otherwise} \end{cases}$$

Considering data dependencies of the other CAs, Φ vector need to be copied to global memory after the calculation phase done. Otherwise the rest of the network cannot complete the weight updates properly. Pseudocode of the updatePHI algorithm is given in algorithm 2.

Algorithm 2 Update Φ Parallel

```
Require: $\vec{E}$, $\vec{F}$, $\theta$, $n$
 1: procedure UPDATEPHI
 2:     i ← blockDim.x · blockIdx.x + threadIdx.x
 3:     stride ← blockDim.x · gridDim.x
 4:     while $i < n$ do                    ▷ Insufficient thread condition considered
 5:         $temp \leftarrow E[i] - F[i]$
 6:         $i \leftarrow i +$ stride
 7:         if $temp > \theta$ then
 8:             $\phi_i \leftarrow 1$
 9:         else
10:             $\phi_i \leftarrow 0$
```

4.1.2 Update E

Energy level update is the most computationally costly procedure among all update procedures. The cost is a consequence of the sparse dot product operation.

$$\textbf{if } \phi_b(t) = 0 \ :$$

$$E_b(t+1) = \frac{1}{d}E_b(t) + \sum_{a=1}^{n} \omega_{ab} \times \phi_a(t)$$

$$\textbf{if } \phi_b(t) = 1 \ :$$

$$E_b(t+1) = \sum_{a=1}^{n} \omega_{ab} \times \phi_a(t)$$

The dot product operation requires only a column of the Ω matrix. Since CSC format has been used for sparse data storage, extracting the required column from the matrix can be done by taking two consecutive column offsets as split indices. Let's say operation needs the 2^{nd} column matrix of the A matrix.

$$\text{A} = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

CSC Storage Format

$$\text{column offsets} = \begin{bmatrix} 0 & 2 & 5 & 7 & 9 \end{bmatrix}$$

$$\text{row indices} = \begin{bmatrix} 0 & 2 & 0 & 1 & 3 & 1 & 2 & 2 & 3 \end{bmatrix}$$

$$\text{values} = \begin{bmatrix} 1 & 5 & 7 & 2 & 6 & 8 & 3 & 9 & 4 \end{bmatrix}$$

The extraction operation can be done as follows:

1. Mark the n^{th} and $(n+1)^{th}$ elements of the column offset vector with start and end flags.
   $CO = \begin{bmatrix} 0 & 2 & \underbrace{5}_{\text{start}} & \underbrace{7}_{\text{end}} & 9 \end{bmatrix}$

2. Use the start and end index to cut the row indices and values vectors.
   $RI = \begin{bmatrix} 0 & 2 & 0 & 1 & 3 & |\mathbf{1} & \mathbf{2}| & 2 & 3 \end{bmatrix}$
   $V = \begin{bmatrix} 1 & 5 & 7 & 2 & 6 & |\mathbf{8} & \mathbf{3}| & 9 & 4 \end{bmatrix}$

3. Then those sub-vectors can be used in 1-D coordinate(COO) format for further calculations.
   $RI = \begin{bmatrix} 1 & 2 \end{bmatrix}$
   $V = \begin{bmatrix} 8 & 3 \end{bmatrix}$

For the sake of simplicity, dot product operation is done by a separate kernel. The algorithm has been designed in a way that even if one block with one kernel has been allocated for the operation, the dot product can be found correctly. Moreover, to reduce the cost, parallel reduction has been used for summation operation. The pseudocode of the dot product operation has been given in Algorithm 3.

Algorithm 3 Parallel Dot Product

```
Require: $\vec{RI}$, $\vec{V}$, $\vec{\Phi}$, $b$, $n$
 1: procedure UPDATEE
 2:     i ← blockDim.x · blockIdx.x + threadIdx.x
 3:     tid ← threadIdx.x
 4:     stride ← blockDim.x · gridDim.x
 5:     $v_{off} \leftarrow CO[b]$
 6:     $p_{off} \leftarrow RI[i + v_{off}]$
 7:     end ← $CO[b+1]$
 8:     while $i <$ end do                    ▷ Insufficient thread condition considered
 9:         temp ← temp + $(V[i + v_{off}] \times \Phi[i + p_{off}])$
10:         $i \leftarrow i +$ stride
11:     cache[tid] ← temp
12:     __syncthreads()
13:     $j \leftarrow \frac{\text{blockDim.x}}{2}$
14:     while $j > 0$ do                      ▷ Parallel Reduction
15:         if tid $< j$ then
16:             cache[tid] ← cache[tid] + cache[tid + j]
17:         __syncthreads()
18:         $j \leftarrow \frac{j}{2}$
19:     if tid = 0 then
20:         atomicAdd(product, cache[0])
```

After the dot product has been calculated, the problem reduces to vector summation problem. The pseudocode of the updateE algorithm has been given in Algorithm 4.

Algorithm 4 Update E Parallel

```
Require: $\Omega$, product, $n$
 1: procedure UPDATEE
 2:     i ← blockDim.x · blockIdx.x + threadIdx.x
 3:     stride ← blockDim.x · gridDim.x
 4:     while $i < n$ do                      ▷ Insufficient thread condition considered
 5:         if $\Phi[i] = 1$ then
 6:             $e_i \leftarrow$ product
 7:         else
 8:             $e_i \leftarrow \frac{1}{d} \cdot e_i +$ product
 9:         $i \leftarrow i +$ stride
```

4.1.3 Update F

Fatigue level update is similar to a conditional vector summation problem.

$$\textbf{if } \phi_b(t) = 0 \ :$$
$$F_b(t) = max\{0, F_b(t-1) - F^r\}$$

$$\textbf{if } \phi_b(t) = 1 \ :$$
$$F_b(t) = F_b(t-1) + F^c$$

In the implementation of the algorithm, insufficient number of thread condition has been considered. The pseudocode of the algorithm is given in Algorithm 5.

Algorithm 5 Update F Parallel

```
Require: $F$, $F^c$, $F^r$, $n$
1: procedure UPDATEF
2:     i ← blockDim.x · blockIdx.x + threadIdx.x
3:     stride ← blockDim.x · gridDim.x
4:     while $i < n$ do                    ▷ Insufficient thread condition considered
5:         if $\Phi[i] = 1$ then
6:             $f_i \leftarrow f_i + F^c$
7:         else
8:             if $f_i > F^r$ then
9:                 $f_i \leftarrow f_i - F^r$
10:            else
11:                $f_i \leftarrow 0$
12:        $i \leftarrow i +$ stride
```

4.1.4 Update Ω

Weight update is done at the post-synaptic neuron side. In order to determine the type of the update(potentiation or depression), it requires both the pre-synaptic and post-synaptic firing activity.

$$\Delta\omega_{ij} = \begin{cases} \alpha(1-\omega_{ij})e^{W_B - W_i} & \text{if } x_t = 1,\ y_t = 1 \\ -\alpha\omega_{ij}e^{W_i - W_B} & \text{if } x_t = 1,\ y_t = 0 \end{cases}$$

It's important to note that, since the energy level update uses ω vector, E and Ω update need to be done one after the other. If they tried to be done in parallel, updateE process can access t and t-1 time-step results at the same time. This situation does not result in a crash or an error code but, generate false results. The pseudocode of the parallel weight update procedure is given in Algorithm 6.

Algorithm 6 Update Ω Parallel

```
Require: $X$, $Y$, $W_B$, $W_i$
1: procedure UPDATE$\Omega$
2:     i $\leftarrow$ blockDim.x $\cdot$ blockIdx.x + threadIdx.x
3:     stride $\leftarrow$ blockDim.x $\cdot$ gridDim.x
4:     while $i < n$ do                    ▷ Insufficient thread condition considered
5:         if $X[i] = 1$ then
6:             if $Y[i] = 1$ then
7:                 $\Delta\omega \leftarrow \alpha \times (1 - \Omega[i]) \times exp(W_B - W_i)$
8:             else
9:                 $\Delta\omega \leftarrow -\alpha \times \Omega[i] \times exp(W_i - W_B)$
10:        $\Omega[i] \leftarrow \Omega[i] + \Delta\omega$
11:        $i \leftarrow i + $ stride
```

4.2 Fitness Calculation

The calculation of the fitness value as in Algorithm 1 requires $\frac{n \cdot (n-1)}{2}$ steps. Therefore the computational complexity is $O(n^2)$ in a sequential calculation. The parallelization of this algorithm may result in a less time-consuming version. Therefore, algorithm 7 have been implemented for GPU calculation.

Algorithm 7 ParallelFitness

```
Require: $\Xi = \begin{bmatrix}\xi_0 & \xi_1 & \dots & \xi_n\end{bmatrix}$
 1: procedure NQueenFitness
 2:     collision ← 0
 3:     tid ← threadIdx.x
 4:     bid ← blockIdx.x
 5:     temp ← Ξ[bid]
 6:     if tid<bid then
 7:         $d \leftarrow |\text{temp} - \Xi[\text{tid}]|$
 8:     if $(d = 0)||(d = \text{bid} - \text{tid})$ then                ▷ Collision
 9:         cache[tid] ← 1
10:     else
11:         cache[tid] ← 0
12:     __syncthreads()
13:     $j \leftarrow \frac{\text{blockDim.x}}{2}$
14:     while $j > 0$ do                                        ▷ Parallel Reduction
15:         if tid < $j$ then
16:             cache[tid] ← cache[tid] + cache[tid + j]
17:         __syncthreads()
18:         $j \leftarrow \frac{j}{2}$
19:     if tid = 0 then
20:         collision ← cache[tid]
```

This version of the fitness calculation requires n blocks and n threads while n is equal to number of elements in the chromosome. This limits the n-queen problem by n¡1024 because the limit of the blocksize = 1024. However, the expected region of operation in this project is $n \in [1, 10]$. Therefore, the limit is much more greater than the expected maximum value.

5 Performance Analysis

In order to evaluate the performance of the project, success rate over cycles, and time spent vs. board size have been determined as two main aspects. The effort will be made on improving these two aspects by parameter tuning and by searching for best thread/block/grid size for kernels.

First of all, success rate vs. cycles plot is expected to seem as in Figure 16 roughly. If the overall system works well, the success rate will start near 50 % and increase with an exponentially decreasing acceleration. The reason behind is that at first, in the random board generation case it's very unlike that all queens are colliding or the board represents an acceptable solution. That means that the initial point should be far from 0

Also, while value module activity is increasing, the success rate needs to increase at the same time. However, no matter how powerful the value module is, there is always a chance for explore module to excite the board. This situation will result in wrong queen placements at a gradually decreasing rate.

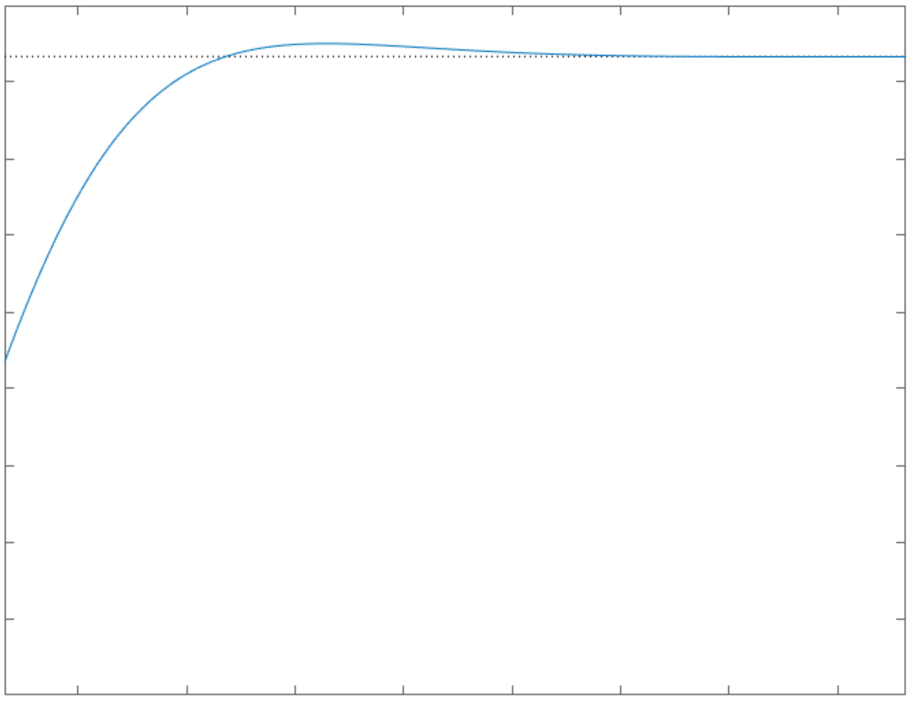


Figure 16: Expected Success Rate vs. Cycles Characteristic

Characteristic of the expected time passed until convergence vs. board size is expected to be similar to Figure 17. The reason behind the exponential function guess is based on Table 1. In this table, number of solutions for the n-queens problems given different n values have been listed. The solutions tend to increase exponentially. This means that the algorithm faces a much bigger search space while n is increasing. The author has deduced that in a bigger search space, modules will spend exponentially more time before reaching the steady-state.

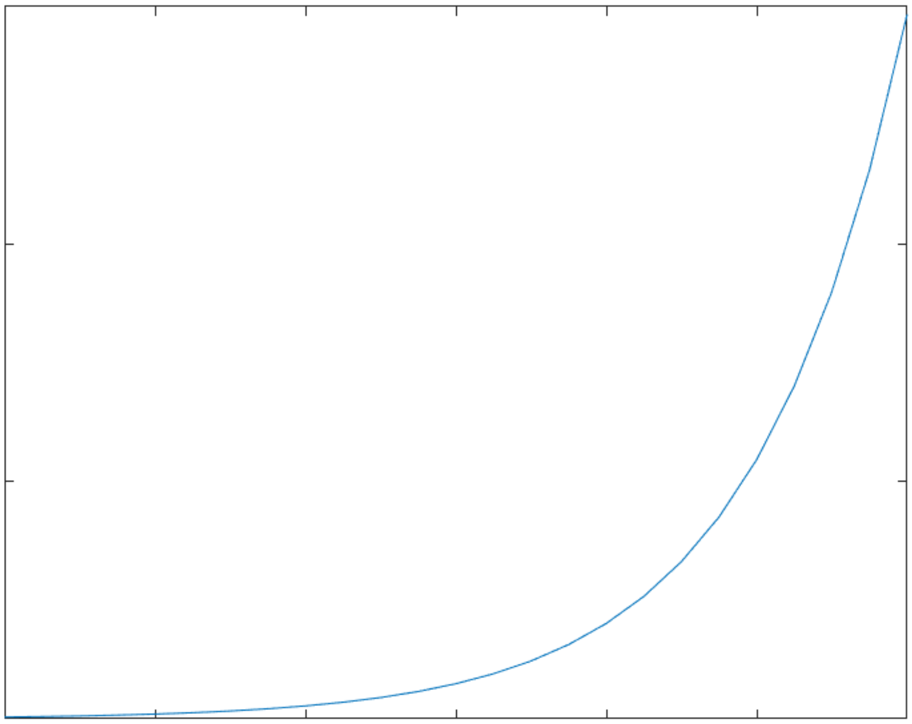


Figure 17: Expected Time Spent vs. Board Size Characteristic

From this point on, candidate modifications to reduce the time spent given the board size and increasing the success rate will be discussed.

5.1 Parameter Tuning

So far, a number of parameters have been introduced without assigning any default value. In this stage, default values will be determined and methods to find better ones will be discussed. To begin with, the n value of the n queen problem can be chosen. Table 1 shows that there is no possible solution for n¡4. Moreover, there is too much solution for a human controller to check the results by hand for n¿6. Therefore, n = 5 has been chosen. Also, the number of neurons inside a CA has been drawn as 10 but never said that it has to be 10. Nevertheless, $|CA|$ has been chosen as 10 for convenience.

$$n = 8$$
$$|CA| = 10$$

After deciding the n and $|CA|$ values, the first thing to consider should be the module setups. Suppose that $|M|$ stands for the number of CAs in the memory module, and $|B|$ for the number of CAs in the board module. In Section 2.4, $|B|$ has been assigned by $n \cdot \lceil log_2 n \rceil$ CAs. Also, the memory module has been noted in Section 2.5 to have more CAs than the board module. For now, the number of $|M|$ has been chosen as $n \cdot n$. Increasing and decreasing $|M|$ will be tested while the rest of the system stays the same. This test is named as $|M|$ Test. Table 4 has been given as a reference.

Table 4: $|M|$ test

$\|M\|$	Final Success Rate	Time Spent
$\frac{n}{4}$		
$\frac{n}{2}$		
n		
$n \cdot \lceil log_2 n \rceil$		
$2n$		
$n \cdot n$		
$\vdots$		

CAs represent the chessboard and construct the memory of the system by representing bits through their ignition status. Θ parameter is the threshold for the decision of the ignition. For now, it has been chosen as 20%, but it requires fine-tuning. Therefore, it will be tested in a one against all manner as number of CAs. However, for now, there is no reason seen to keep the memory and board threshold different. Therefore, at first, a universal increase and decrease will be tested and then module-specific increase and decreases will be tested. This test is named as Θ test.Table 5 has been given as a reference.

Table 5: Θ test

Θ_M	Θ_B	Final Success Rate	Time Spent
5%	5%		
5%	10%		
$\vdots$	$\vdots$		

Number of cells for the memory and board modules are dependent on previously chosen parameters. However, value and explore modules are not. These are selected as 500 and need to be tested. This test is named as the cell number test. Table 6 has been given as a reference.

Table 6: Cell Number Test

$\#_M$	$\#_B$	Final Success Rate	Time Spent
100	100		
100	200		
$\vdots$	$\vdots$		

Neuron activation threshold, decay constant, recovery constant, and fatigue constant are neuron-specific parameters. Experimenting with these can be handled in one test. For the sake of simplicity, they all have assigned to the same values. However, the first thing is to be done is to make the system work well. Therefore, their region of operation will be learned after the implementation. One against all test procedures will be applied to those too. This test is named as neuron parameter test.

Learning rate is only meaningful for memory module since only the neurons in this module learn. It is expected that the learning rate will have a huge effect on the convergence characteristic of the success rate. Therefore, the values that it can have from 0 to 1 will be investigated studiously. This test is named as learning rate test. Table 7 has been given as a reference.

Table 7: α test

α	Final Success Rate	Time Spent
0.05		
0.1		
$\vdots$		

Table 8 has been filled with default values of the parameters that has been stated above.

Table 8: Default Values to Assign Parameters to be Tuned

Parameter	**Module**			
	Memory	Board	Value	Explore
Number of CA	$n \cdot n$	$n \cdot \lceil log_2 n \rceil$	-	-
CA Activation Threshold Θ	0.2	0.2	-	-
Number of Cells	$\|CA\| \cdot \|M\|$	$\|CA\| \cdot \|B\|$	500	500
Neuron Activation Threshold θ	4.0	4.0	4.0	4.0
Decay Constant d	1.0	1.0	1.0	1.0
Recovery Constant F^r	1.0	1.0	1.0	1.0
Fatigue Constant F^c	1.0	1.0	1.0	1.0
Learning Rate α	0.2	**0**	**0**	**0**

The tests mentioned for parameter tuning is listed as follows:

- $|M|$ test
- Θ test
- Cell number test
- Neuron parameter test
- Learning Rate Test

5.2 Thread/Block/Grid Size Search

Experience coming from the previous assignments tells that thread/block/grid size choice has an effect on time spent by the kernel functions. Therefore, the best configuration will be searched for all parallel algorithms explained in Section 4. The least time consuming configuration will be chosen as the best. Table 9 shows an example table which can be used as a guide in searching.

Table 9: Thread/Block/Grid Size Search

Configuration			**Time Spent**
Thread	Block	Grid	
1	1	1	
1	1	4	
1	1	16	
1	1	32	
⋮	⋮	⋮	

5.3 Environment

The implementation and development of the system will be done using the computer environment in Table 10.

Table 10: Computer Environment for Experiment Setup

Feature	**Specification**
GPU	NVIDIA GeForce 940 MX
Compute Capability	5.0
GPU Memory	2 GB
GPU Bandwidth	40.10 GB/s
GPU TDP	23 W
GPU Cores	384
CPU	Intel Core i7 - 7500
CPU Speed	2.7 GHz - 2.9 GHz
RAM	8 GB
OS	Windows 10 Home
System	64 - bit
Programming	Visual Studio(CUDA)

6 Acknowledgments

The author would like to thank Dr. Chris Huyck for helpful discussions and valuable suggestions.

7 References

[1] U. Cakal, “Solving n-queens problem using cell assemblies on gpu : Project proposal report,” 2020.

[2] N. J. A. Sloane, “Number of ways of placing n nonattacking queens on an n x n board.” 2018. [Online]. Available: https://oeis.org/A000170

[3] E. Byrne and C. Huyck, “Processing with cell assemblies,” *Neurocomputing*, vol. 74, no. 1, pp. 76 – 83, 2010, artificial Brains. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925231210002572

[4] R. V. Belavkin and C. R. Huyck, “Conflict resolution and learning probability matching in a neural cell-assembly architecture,” *Cognitive Systems Research*, vol. 12, no. 2, pp. 93 – 101, 2011, the 9th International Conference on Cognitive Modeling. Manchester, UK, July 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389041710000598

[5] S. Kaplan, M. Sonntag, and E. Chown, "Tracing recurrent activity in cognitive elements (trace): A model of temporal dynamics in a cell assembly," *Connection Science*, vol. 3, no. 2, pp. 179–206, 1991.