A Representation of Membrane Computing with a Clustering Algorithm on the Graphical Processing Unit

Authors:

Ravie Chandren Muniyandi, Ali Maroosi

Date Submitted: 2021-04-16

Keywords: CUDA, GPU multiprocessor occupancy, GPU kernel execution, parallel computing, membrane systems

Abstract:

> Long-timescale simulations of biological processes such as photosynthesis or attempts to solve NP-hard problems such as traveling salesman, knapsack, Hamiltonian path, and satisfiability using membrane systems without appropriate parallelization can take hours or days. Graphics processing units (GPU) deliver an immensely parallel mechanism to compute general-purpose computations. Previous studies mapped one membrane to one thread block on GPU. This is disadvantageous given that when the quantity of objects for each membrane is small, the quantity of active thread will also be small, thereby decreasing performance. While each membrane is designated to one thread block, the communication between thread blocks is needed for executing the communication between membranes. Communication between thread blocks is a time-consuming process. Previous approaches have also not addressed the issue of GPU occupancy. This study presents a classification algorithm to manage dependent objects and membranes based on the communication rate associated with the defined weighted network and assign them to sub-matrices. Thus, dependent objects and membranes are allocated to the same threads and thread blocks, thereby decreasing communication between threads and thread blocks and allowing GPUs to maintain the highest occupancy possible. The experimental results indicate that for 48 objects per membrane, the algorithm facilitates a 93-fold increase in processing speed compared to a 1.6-fold increase with previous algorithms.

Record Type: Published Article

Submitted To: LAPSE (Living Archive for Process Systems Engineering)

Citation (overall record, always the latest version):	LAPSE:2021.0191
Citation (this specific file, latest version):	LAPSE:2021.0191-1
Citation (this specific file, this version):	LAPSE:2021.0191-1v1

DOI of Published Version: https://doi.org/10.3390/pr8091199 *processes* MDPI

Article

A Representation of Membrane Computing with a Clustering Algorithm on the Graphical Processing Unit

Ravie Chandren Muniyandi * and Ali Maroosi

Research Center for Cyber Security, Faculty of Information Science and Technology, National University of Malaysia, Bangi 43600, Selangor, Malaysia; ali.maroosi@gmail.com

* Correspondence: ravie@ukm.edu.my

Received: 30 July 2020; Accepted: 18 September 2020; Published: 22 September 2020

Abstract: Long-timescale simulations of biological processes such as photosynthesis or attempts to solve NP-hard problems such as traveling salesman, knapsack, Hamiltonian path, and satisfiability using membrane systems without appropriate parallelization can take hours or days. Graphics processing units (GPU) deliver an immensely parallel mechanism to compute general-purpose computations. Previous studies mapped one membrane to one thread block on GPU. This is disadvantageous given that when the quantity of objects for each membrane is small, the quantity of active thread will also be small, thereby decreasing performance. While each membrane is designated to one thread block, the communication between thread blocks is needed for executing the communication between membranes. Communication between thread blocks is a time-consuming process. Previous approaches have also not addressed the issue of GPU occupancy. This study presents a classification algorithm to manage dependent objects and membranes based on the communication rate associated with the defined weighted network and assign them to sub-matrices. Thus, dependent objects and membranes are allocated to the same threads and thread blocks, thereby decreasing communication between threads and thread blocks and allowing GPUs to maintain the highest occupancy possible. The experimental results indicate that for 48 objects per membrane, the algorithm facilitates a 93-fold increase in processing speed compared to a 1.6-fold increase with previous algorithms.

Keywords: membrane systems; parallel computing; CUDA; GPU multiprocessor occupancy; GPU kernel execution

1. Introduction

Membrane systems are a class of computational models that inspired their computation from cell biology. Membrane systems have been applied in different areas [1], including zooplankton migrate vertically in system biology [2], image processing [3,4], robot path planning problem [5], power systems as well as address highly-complex computational problems such as traveling salesman, knapsack, Hamiltonian path and satisfiability [6–10]. The main elements of a membrane system include (i) structure, including its delimiting compartments; (ii) multisets of objects; (iii) biochemically-inspired rules. Membrane systems are based on the parallel and distributed systems found in biological cells, with both the objects and membranes located within the system capable of being processed by the same set of rules [11,12].

While various attempts have been carried out and are currently ongoing to imitate membrane systems, some of these had been implemented on single-processor computers [13], neglecting the intrinsic parallelism of the systems being simulated. Efforts are now focused on increasing membrane-system simulation speed through the use of multicore processors [14–17], cluster of computers [10,18], designs of parallel hardware and the field programming gate arrays (FPGA) [19,20], and graphics processing units (GPU) [21–26]. Installation and employment of cluster computing are cost-prohibitive, making this option feasible only for large companies. Additionally, implementation and modification of the FPGA code are very time-consuming. However, thousands of threads able to be executed in parallel using one low-cost platform of GPU. Different variants of P systems have been simulated on GPU [27], including enzymatic numerical P system [28], spiking neural P system [3], population dynamics P systems [29], etc. This study used active membrane systems, another variant of the membrane system for GPU simulation with Compute Unified Device Architecture (CUDA). CUDA is a platform for parallel computing using GPU to develop highly parallel algorithms on graphic units.

The first implementation of an active membrane system was reported in [30]. Each membrane was assigned to one GPU thread block, the CPU accomplished the selection process, and the GPU undertook the execution of rules. A subsequent study in [31] assigned each membrane to one thread block with both selection and execution procedures performed by the GPU. Active membrane systems were used to resolve the satisfiability (SAT) investigation by using a cluster of GPUs and by designating a particular active membrane where the number of objects in every membrane approached the highest achievable number of threads per thread block in the GPU to escalate the number of active threads in every thread block, thereby increasing effectiveness [32,33]. The maximum quantity of threads available per thread block is 512 for version 1.x GPUs and 1024 for versions 2.x and 3.0 GPUs [34]. The method in [32,33] was useful in situations that can modify the number of objects while the number of objects in the membrane was problem-dependent and not modifiable.

Here, we represent active membrane systems as matrices. In our previous study, active membrane systems were expressed as a matrix [21]. This study generalized concepts in [21] and assigns matrices to each thread block instead of assigning a matrix. This approach would increase the thread block occupancy of the GPU. As a result, the efficiency of the GPU would be increased. The object-weighted network and membrane-weighted network have been introduced to automatically organize dependent objects and membranes within the same groups according to their dependencies, allowing them to be executed by the same threads or thread blocks. The algorithm presented here tests communication rates between membranes. It determines whether assigning sub-matrices, including additional membranes to each thread block, will cause a decrease in communications between threads and increase GPU occupancy. Matrices are apportioned into sub-blocks to fully utilize the maximum possible quantity of threads in each thread block [35,36]. This method eliminates shortcomings associated with previously implemented methods that applied one of the ensuing two notions: (i) allocating any quantity of objects in every membrane to every thread block or (ii) first designating an active membrane system in which the quantity of objects in every membrane achieves the highest quantity of threads in a GPU thread block.

When the quantity of objects in one membrane is large, performance drops due to the limitations of registers and the local and shared memory available for each thread or thread block. In our previous study, active membrane systems were represented as a matrix [21]. This study used a matrix representation to assign an object to a thread. This would decrease the usage of a thread capability. Our algorithm addresses these problems by representing membrane systems as matrices. When a membrane contains many objects, this enables its efficient distribution among different thread blocks to improve communication between objects.

Conversely, when the total number of objects is low, the algorithm assigns objects from different membranes to a single thread block. These efforts allow maximizing using of GPU computational resources. Additionally, mapping membrane objects to different matrices enable the assignment of dependent objects to the same thread, thereby decreasing unnecessary communication between threads and to enhance performance.

Therefore, the main contribution of this paper is to develop a classification algorithm based on the communication rate to manage dependent objects and membranes in membrane computing implemented on GPU. This study defined a weighted network based on communication rate and assigned them to sub-matrices for execution by the same threads and thread blocks to improve the communication between thread blocks, reduce kernel invocations, and maximize GPU occupancy.

This approach can be applied to solve NP-hard problems such as traveling salesman, knapsack, and satisfiability. The traveling salesman problem is one of the combinatorial optimization problems and applied to the design of network structure, machine scheduling system, the manufacturing of cellular, and the assignment of frequency. Boolean satisfiability problem is used to solve problems such as in mathematics, artificial intelligence, data mining, and circuit design. The knapsack problem is another combinatorial optimization problem that aims to maximize the value of items under backpack capacity constraints, and it is being applied to solve decision problems.

With the increase in the number of inputs, the time complexity of solving these problems also will increase significantly. With the proposed classification algorithm, the efficiency and effectiveness of the parallelism of GPU will reduce the computational complexity. It is feasible to generate all solutions and then screen out the suitable solutions in polynomial time or even linear time.

2. Background

2.1. Active Membrane Systems

Multiple parts, such as a skin and the membranes, are compartmentalized inside an active membrane system. In each membrane, various sets of objects (comparable to biochemical materials) and groups of evolutionary rules (identical to responses) exist inside of membranes' compartments (Figure 1).

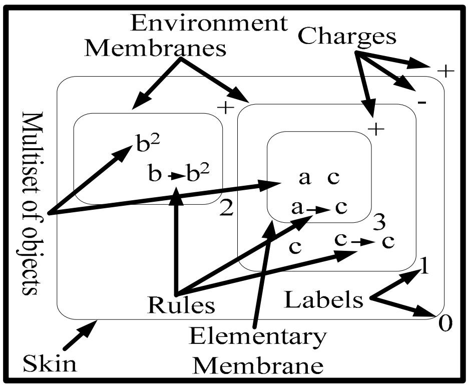


Figure 1. P-system structure. This system consists of membranes, objects, rules, and charges. Rules react to evolve objects and membranes. Charges respond to control processing.

Active membrane systems are described as $\pi = (O, H, \mu, \omega_0, \ldots, \omega_m,\ R)$, where,

(1) m is the preliminary degree of the membrane system, equivalent to the number of membranes, $m \geq 1$;

(2) O is the object alphabet;

(3) H is a predetermined group of membrane labels;

(4) μ is a membrane arrangement that includes m membranes labeled *1, … ,m*, each having a preliminary neutral polarization and tagged with components from H;

(5) $\omega_0, \ldots, \omega_m$ are strings over O, designating the multisets of objects positioned in the m compartments of μ;

(6) R is a predetermined group of rules described as:

(a) Evolution rules for the object, $[a \rightarrow u]_h^\alpha$, in which $h \in H$; $\alpha \in \{+, -, 0\}$ are electrical charges, $a \in O$, and u is a string over O that designates a multiset of objects connected with membranes that depend on the label and the charge related with the membranes;

(b) “In” communication rules for the object from an environment entering into a membrane, $a[\ ]_h^{\alpha} \rightarrow [b]_h^{\beta}$, in which $h \in H;\ \alpha, \beta \in \{+, -, 0\}; a, b \in O$, when an object enters a membrane, it is likely that this object modifies, where the preliminary charge, α, is transformed to β;

(c) “Out” communication rules for the object from a membrane entering into an environment, $[a]_h^{\alpha} \rightarrow [\ ]_h^{\beta} b$, in which $h \in H;\ \alpha, \beta \in \{+, -, 0\}; a, b \in O$, when an object is discharged from a membrane, it is likely that this object modifies, where the preliminary charge, α, is transformed to β;

(d) Dissolving rules for membrane, $[a]_h^{\alpha} \rightarrow b$, in which $h \in H;\ \alpha \in \{+, -, 0\}; a, b \in O$, a membrane with a particular charge is dissolved in a reaction with a probably altered object;

(e) Division rules for membrane, $[a]_h^{\alpha} \rightarrow [b]_h^{\beta}[c]_h^{\gamma}$, in which $h \in H;\ \alpha, \beta, \gamma \in \{+, -, 0\}; a, b, c \in O$, in response to an object, the membrane is divided into two membranes; the label remains unceasing, but the charge could modify, and the objects inside the membrane are duplicated, except for a, which may be altered in each membrane.

These rules are implemented in accordance with the ensuing conceptions: Every rule is implemented in a maximally parallel mode, denote that one membrane object is consumed by at the most one rule in every phase. If an object is able to be consumed by beyond one rule, it must be picked in a non-deterministic means and then consumed by that rule. Whichever object that able to evolve based on any rule should evolve in one step. Rules (b) through (e) are not able to be implemented concurrently in one computational step, and the rules connected with membranes with the label, h, are implemented merely to those membranes. Active membrane systems are explained in detail by Rozenberg et al. [37].

2.2. Compute Unified Device Architecture (CUDA)

Processes on threads, thread blocks, and kernels were programmed by using CUDA [34]. Cores are consumed to execute threads, in which groups of thread blocks apportioned to one streaming multiprocessor (SM) and are considered a kernel to be performed on the SMs. Every thread block is implemented on a single SM. The same shared memory able to be consumed by all threads within a thread block. In comparison, mutual access to shared memory is unacceptable between thread blocks. Kernel re-invocation is ensued between thread blocks to apply thread synchronization. Meanwhile, thread synchronization within a thread block transpires through barrier synchronization.

Atomic operations in the same kernel are applied for synchronization between thread blocks [21,38,39] and demonstrated ineffectiveness. Every thread acquires fast memory across registers and slow memory across local memory. Global memory is also slow, though it is reachable through threads in every part of the thread block.

2.3. Applied Methods in Previous Research

Suppose that there are m membranes and n objects. Assume that a_{ij} signifies the ith object in the jth membrane. Previously the methods to GPU simulation of active membrane systems represented every membrane to a single thread block and every membrane object to related threads within that thread block (Figures 2 and 3, line 0) [31–33]. If the quantity of objects within every membrane is insignificant, then the quantity of active threads in every thread block is negligible relative to the quantity of available threads. This leads to the inefficient use of computational resources and minimizes GPU occupancy.

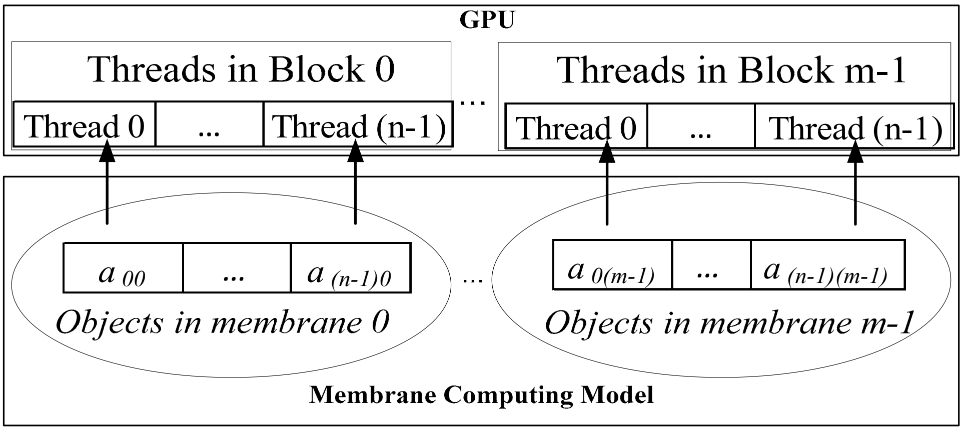
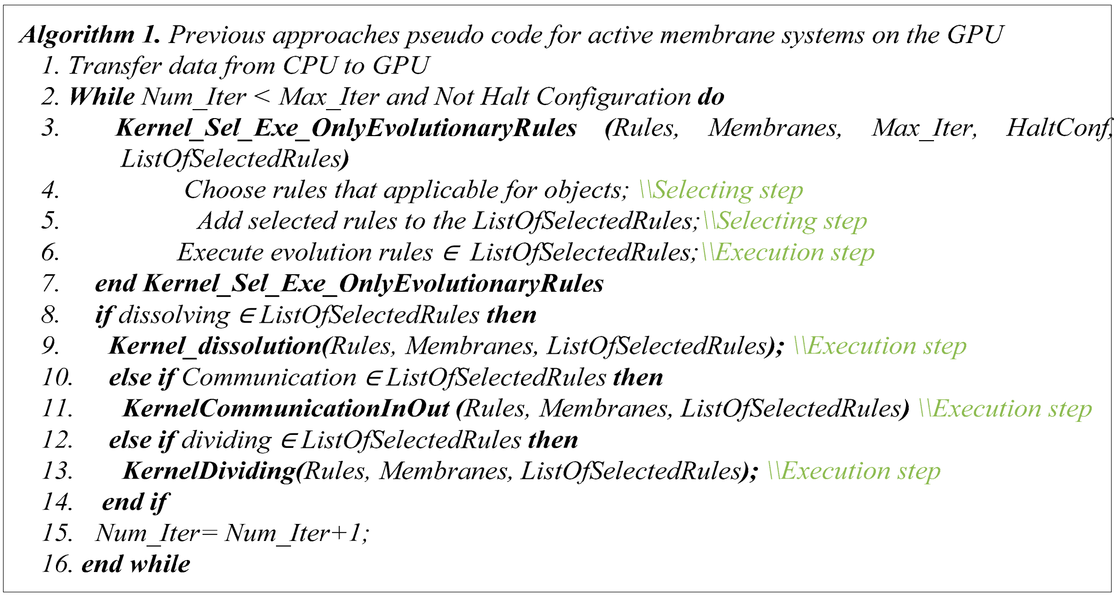


Figure 2. A membrane is assigned to one thread block, and objects are assigned to threads within each thread block [31–33].



```
Algorithm 1. Previous approaches pseudo code for active membrane systems on the GPU
 1. Transfer data from CPU to GPU
 2. While Num_Iter < Max_Iter and Not Halt Configuration do
 3.       Kernel_Sel_Exe_OnlyEvolutionaryRules (Rules, Membranes, Max_Iter, HaltConf,
          ListOfSelectedRules)
 4.              Choose rules that applicable for objects; \\Selecting step
 5.                Add selected rules to the ListOfSelectedRules;\\Selecting step
 6.              Execute evolution rules ∈ ListOfSelectedRules;\\Execution step
 7.    end Kernel_Sel_Exe_OnlyEvolutionaryRules
 8.    if dissolving ∈ListOfSelectedRules then
 9.      Kernel_dissolution(Rules, Membranes, ListOfSelectedRules); \\Execution step
10.      else if Communication ∈ListOfSelectedRules then
11.        KernelCommunicationInOut (Rules, Membranes, ListOfSelectedRules) \\Execution step
12.      else if dividing ∈ListOfSelectedRules then
13.        KernelDividing(Rules, Membranes, ListOfSelectedRules); \\Execution step
14.     end if
15.    Num_Iter= Num_Iter+1;
16. end while
```

Figure 3. Algorithm 1 for active membrane systems on the GPU [31–33].

Mapping individual membranes to single thread blocks will develop a mechanism in which the communication among membranes requires interaction among thread blocks. Thread-block synchronization necessitates the invocation of separate kernels. Inter-block synchronization without launching a second kernel has been attempted using atomic operations and the *threadfence()* function; however, the overhead of this approach is high [21,38,39]. Therefore, distinct kernel invocation is required for thread-block synchronization, making it impossible to perform division and communication rules in a single kernel with no invocation of a second (Figure 3, lines 9, 11, and 13).

Previously, division rules could not be executed without the invocation of a separate kernel, given that a newly produced membrane can be allocated to another thread block (Figure 3, line 13). The algorithm presented here introduces a new representation of membrane systems in GPUs. The dissolution, communication, and dividing rules are executable within the same kernel without the necessity of invoking a separate one. A comparison between the two algorithms can be found in Section 3.2.

3. Methods

3.1. Proposed Approach

Here, we represent membrane systems as matrices that can be divided into sub-blocks to balance the number of threads used in GPU thread blocks [35,36]. The objects in the membranes are subsequently assigned to matrix entries (Figure 4), thereby increasing the efficiency with which the matrix allocates the threads in the thread blocks. In this condition, the quantity of threads in each thread block is not reliant on the quantity of objects inside each membrane. Still, rather objects in a single membrane can be allocated to threads in dissimilar thread blocks, and objects from distinctive membranes can be allocated to threads in one thread block. The new representation and the proposed method have the following advantages:

(1) The computational workload associated with the membranes is distributed between thread blocks such that the workload is balanced, and higher performance is achieved.

(2) It is possible to decrease communication between threads within one thread block.

(3) It is probable to automatically allocate membranes that demand to communicate with each other to the same thread block. In some instances, membranes with more significant dependencies are assigned to the same thread blocks to decrease inter-block communication, despite the possibility that this could decrease GPU occupancy.

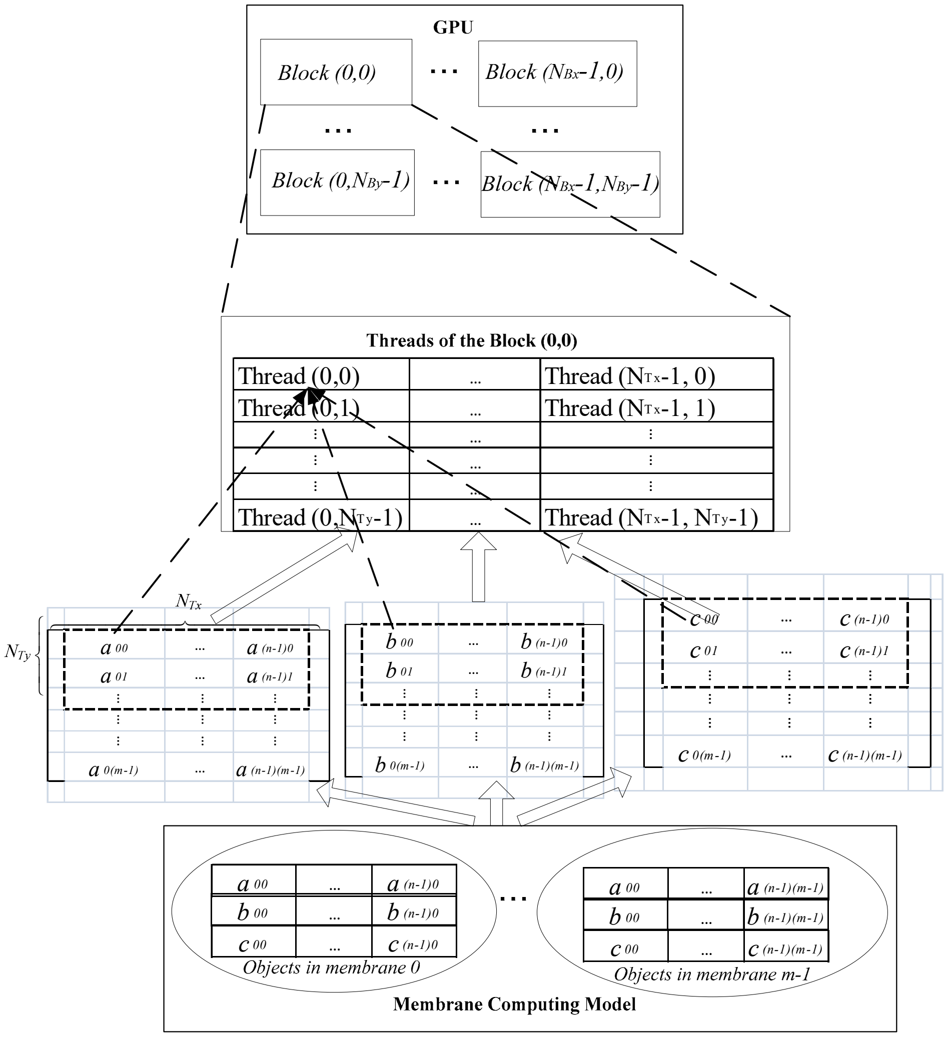


Figure 4. Representation of a membrane system as matrices.

This paper proposes mapping membrane objects to different matrices (Figure 4), allowing several matrices (sub-matrices) to be assigned to threads in thread blocks. It is more efficient to map objects that share dependencies to the same entry positions within different matrices, given that matrix entries at the same position are assigned to the same thread. Therefore, objects that share dependence are mapped to the same thread, decreasing the communication between threads and reducing synchronization time.

An example of a mapping involving three matrices is illustrated in Figure 4 and described as follows. Threads in a thread block can be represented as a two-dimensional matrix with $NTx \times NTy$ threads within $N_{Bx} \times N_{By}$ thread blocks. Objects $a_{i,j}$, $b_{i,j}$, and $c_{i,j}$ ($I = 0, \ldots, n–1; j = 0, \ldots, m–1$) share dependence and are mapped to three different matrices in the same entry position, (i,j), for execution within a single thread, (i,j). Membranes, m, having n objects of dependent variables, $a_{i,j}$, $b_{i,j}$, and $c_{i,j}$, and map those objects by assigning them to a single row of a given matrix. Consequently, for m membranes with n dependent objects, $a_{i,j}$, $b_{i,j}$, and $c_{i,j}$, each dependent object is mapped to three $m \times n$ matrices in the same entry position, (i,j), for execution on a single thread, (i,j). To avoid initiating multiple matrix operations simultaneously, it is possible to consider these matrices as a single matrix virtually divided into three sub-matrices.

To maximize occupancy, active threads, and the warps per SM should be equal to the maximum warps per SM, $MaxWarp_{sm}$, and the maximum number of threads per SM, $MaxThr_{sm}$ [34]. The maximum number of resident thread blocks per SM, $MaxBlk_{sm}$, is also limited depending upon GPU computational capability [34]. Therefore, the minimum number of threads per thread block necessary to achieve high or full occupancy, $MinThrBlk_{HighOccup}$, is calculated as:

$$MinThrBlk_{HighOccup} = MaxThr_{sm}/MaxBlk_{sm} \tag{1}$$

With $MinThrBlk_{HighOccup}$, the threads in each thread block achieve high occupancy; however, when the number of threads per thread block increases, more dependent threads can be allocated to a thread block, decreasing inter-block communication. In this case, the active threads per thread block can be maximized to the number of resident threads per thread block allowed, $MaxThrBlk$, based on GPU computational capability [34]. Therefore, to achieve high occupancy, the number of active threads per thread block, Thr_{Blk}, equal to $N_{Tx} \times N_{Ty}$ (Figure 4) can be determined as follows:

$$N_{Tx} \times N_{Ty} = Thr_{Blk} = \begin{cases} MinThrBlk_{HighOccup}\,; \text{If threads number increasing, not effects on interblock communications} \\ \text{Up to } MaxThr_{Blk}; \quad \text{If threads number increasing, decreases interblock communications} \end{cases} \tag{2}$$

The proposed approach for balancing occupancy and synchronization between threads and thread blocks (Balancing_Occup_Sync_Approach) is described below.

Step 1: Assign objects and membranes to a matrix and assign each sub-matrix, $N_{Tx} \times N_{Ty}$, in Equation (2) (Figure 4) to each thread block to achieve high occupancy.

Step 2: After assigning the membrane system to a matrix, if the communication rate between threads or thread blocks is high:

According to the object- and membrane-weighted network (described later), dependent objects are assigned to the same entry positions within different sub-matrices, and more than one sub-matrix is assigned to each thread block (Figure 4), allowing objects having shared dependencies to be assigned to the same thread. The quantity of sub-matrices per thread block, $NumSubMat_{Blk}$, is equal to the quantity of objects per thread, Obj_{Thr}, and can continue to increase without lowering occupancy. This number is represented as:

$$NumSubMat_{Blk} = Obj_{Thr} = Min\ (MaxDep_{Obj}, ObjThr_{HighOccup}) \tag{3}$$

The $MaxDep_{Obj}$ represents the maximum number of objects that share dependence, and $ObjThr_{HighOccup}$ represents the maximum quantity of objects that can be allocated to every thread while maintaining high occupancy. The number of membranes assigned to the thread block should be close to the estimated number of membranes capable of being assigned to the same thread block while maintaining 100% occupancy.

$$Memb_{Blk} \approx MembBlk_{HighOccup} \tag{4}$$

The $ObjThr_{HighOccup}$ and the $MembBlk_{HighOccup}$ will be determined in Equations (5) and (9). To improve the assignment of dependent objects and membranes, the weighted network described here allows determination of the degree of dependency between objects and membranes to identify the best assignment.

Step 3: If the communication rate between thread blocks (in this step, communication between threads within the same thread block is not considered) remains high after the previous two steps:

Assign additional objects sharing dependency exceeding $ObjThr_{HighOccup}$ from Equation (3) to the same threads to enable the assignment of more dependent membranes to the same thread blocks and potentially decrease communication rates. Assigning objects exceeding $ObjThr_{HighOccup}$ to each thread decreases occupancy. It decreases the rate of communication and synchronization time between *thread blocks.* However, this sacrificing occupancy is useful to improve performance because the *thread block* synchronization is a too time-consuming process. However, decreasing the communications between threads inside the same thread blocks by sacrificing occupancy is not recommended.

Step 4: If Step 3 does not decrease communication rate and synchronization between thread blocks, then kernel re-invocation is required. In this case, it is more advantageous to keep occupancy high, using Step 2 to decrease the communication rate as much as possible without sacrificing occupancy. However, to maintain adequate synchronization and communication between thread blocks, kernel re-invocation is necessary.

The $ObjThr_{HighOccup}$ in Equation (3) represents the maximum number of objects per thread when occupancy remains high and is determined by:

$$ObjThr_{HighOccup} = \min\ (ObjThr_{ConsSh}, ObjThr_{ConsReg}, ObjThr_{ConsLoc}) \tag{5}$$

The number of objects capable of being assigned per thread to achieve high occupancy (Equation (5)) is constrained by the available shared memory, $ObjThr_{ConsSh}$, and the limited amount of shared memory per SM, $MaxSh_{sm}$. The maximum number of threads per SM necessary to achieve high occupancy, $MaxThr_{sm}$ [34], and the average amount of shared memory utilized by each object, $AveSh_{Obj}$, is represented as:

$$ObjThr_{ConsSh} = MaxSh_{sm}/(MaxThr_{sm} \times AveSh_{Obj}) \tag{6}$$

If the objects exceed the number of SM registers (spilled registers) stored in local memory, performance will decrease relative to the slow local-memory access. Therefore, it is necessary to consider the limitations associated with the number of registers per SM. The number of objects capable of being assigned per thread to achieve high occupancy (Equation (5)) is constrained by the available registers, $ObjThr_{ConsReg}$, and the limited number of registers per SM, $MaxReg_{sm}$. The number of threads per SM necessary to achieve the high occupancy, $MaxThr_{sm}$, depends upon GPU computational capability [34]. The average number of registers utilized by each object, $AveReg_{Obj}$, is represented as:

$$ObjThr_{ConsReg} = MaxReg_{sm}/(MaxThr_{sm} \times AveReg_{Obj}) \tag{7}$$

The number of objects capable of being assigned per thread to achieve high occupancy (Equation (5)) is constrained by the available local memory, $ObjThr_{ConsLoc}$, and the limited amount of local memory per thread, $MaxLoc_{sm}$ [34]. The maximum number of threads per SM necessary to achieve high occupancy, $MaxThr_{sm}$ [40], and the average amount of local memory utilized by each object, $AveLoc_{Obj}$, is represented as:

$$ObjThr_{ConsLoc} = MaxLoc_{sm}/(MaxThr_{sm} \times AveLoc_{Obj}) \tag{8}$$

The $AveSh_{Obj}$ from Equation (6), the $AveReg_{Obj}$ from Equation (7), and the $AveLoc_{Obj}$ from Equation (8) are estimated by assigning each membrane object to a single thread, every membrane to a single thread block, and compiling the code using CUDA-C following management of compiler parameters. These parameters include the number of registers per thread, $AveReg_{Obj}$, the spilled registers stored in local memory per thread, $AveLoc_{Obj}$, and the amount of shared memory per SM, which should be divided by the number of objects assigned to each SM to obtain $AveSh_{Obj}$. For $MembBlk_{HighOccup}$, the same calculations can be undertaken from Equation (4) through Equation (8) for $ObjThr_{HighOccup}$ from Equation (3) as:

$$MembBlk_{HighOccup} = min\,\{\, MaxSh_{sm}/(Blk_{SM} \times AveSh_{Memb}),\, MaxReg_{sm}/(\, BlkSMHighOccup \times AveReg_{Memb})\} \tag{9}$$

The maximum shared memory per SM, $MaxSh_{sm}$, and the maximum number of registers per SM, $MaxReg_{sm}$, depend upon GPU computational capability [40]. Blk_{SM} can be obtained by dividing the maximum possible resident threads per SM, $MaxThr_{sm}$, by the number of active threads per thread block, Thr_{Blk}, giving $\mathrm{N}_{\mathrm{Tx}} \times \mathrm{N}_{\mathrm{Ty}}$ from Equation (2). The average amount of shared memory consumed per membrane, $AveSh_{Memb}$, and the average quantity of registers utilized per membrane, $AveReg_{Memb}$, can be estimated by assigning each membrane object to one thread, every membrane to a single thread block, and compiling the code using CUDA-C following management of compiler parameters.

Object- and membrane-weighted network:

A weighted network is employed to appropriately assign objects having shared dependencies and membranes from Steps 2 and 3 of the Balancing_Occup_Sync_Approach. This network, $a \xrightarrow{p} b$, represents a reaction on the object a that occurs at the rate, p, and affects the amount of object b. For example, if object a evolves according to the evolution rule, $a^{100} \rightarrow b^{3}$ assumes that in each step, the number of objects a increases by one, and after 100 steps (with a rate of 0.01 per step), there would be 100 copies of the object a and the rule would evolve and generates three copies of object b. Therefore, in the weighted network, the transition is written as $a \xrightarrow{0.01} b$. In Step 2, each $NumSubMat_{Blk}$ or Obj_{Thr} of objects from Equation (3) is assigned to each thread to improve performance. The Obj_{Thr} objects with higher levels of shared dependency within the weighted network are assigned to the same entry of sub-matrices and assigned to the same thread (Figure 5).

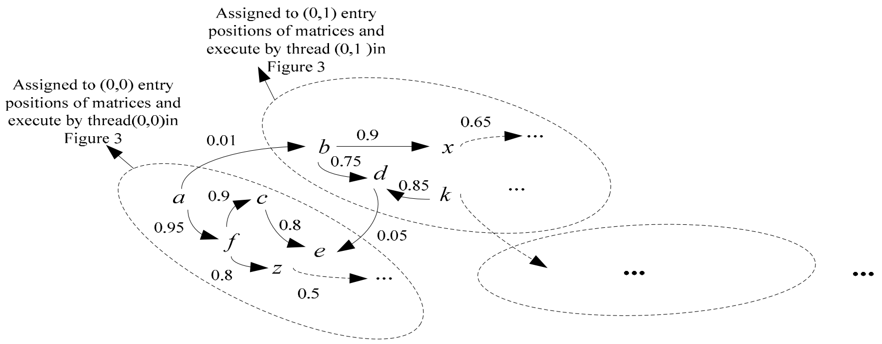

Assigned to (0,1) entry positions of matrices and execute by thread (0,1)in Figure 3
Assigned to (0,0) entry positions of matrices and execute by thread(0,0)in Figure 3
0.01
b
0.9
x
0.65
...
0.75
d
0.85
k
...
a
0.9
c
0.95
0.8
f
e
0.05
0.8
z
...
0.5
...
...

Figure 5. The proposed object-weighted network and classification. Objects that have higher communication rates are organized within the same groups (dashed ellipses). The number of objects per group should satisfy Equation (3) and objects are assigned to the same entry positions within the matrices, enabling execution by the same thread.

To organize related objects within the same groups, begin with those objects having the highest communication rates in the weighted network (Figure 5) and assign them to the same group until the number of objects per thread does not exceed that from Equation (3). For example, according to the rates associated with the objects in Figure 5, *a* and *f* (rate 0.95) and *f* and *c* (0.9) are assigned to the first group, then *b* and *x* (0.9) are assigned to the second group, given that there is no link between *b* and *f*. This is followed by *d* and *k* (0.85) being assigned to the third group. Subsequently, *f* and *z* (0.8) and *c* and *e* (0.8) will be assigned to the first group. Up to this point, *b* and *x* were assigned to the second group, and *d* and *k* were assigned to the third group. Now, object *b* from the second group and object *d* from the third group (0.75) are classified in the same group, and the third group is dissolved. This continues until all objects are classified according to their communication rates in the weighted network and the limitation to the number of objects per thread from Equation (3) has been reached.

We have also defined a membrane-weighted network (Figure 6). Here, $[\,]_i \xrightarrow{p} [\,]_j$ denotes a membrane, *i*, communicating with a membrane, *j*, at the rate, *p*. Similar to the object-weighted network, membranes that have higher communication rates are organized within the same group until Equation (4) is satisfied. Each group is then assigned to sub-matrices, $N_{Tx} \times N_{Ty}$, from Equation (2) to be executed in the same thread block (Figure 4).

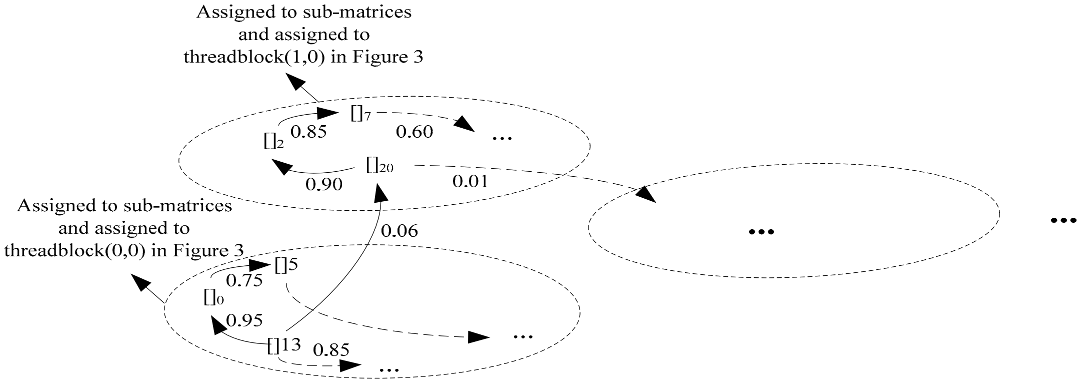
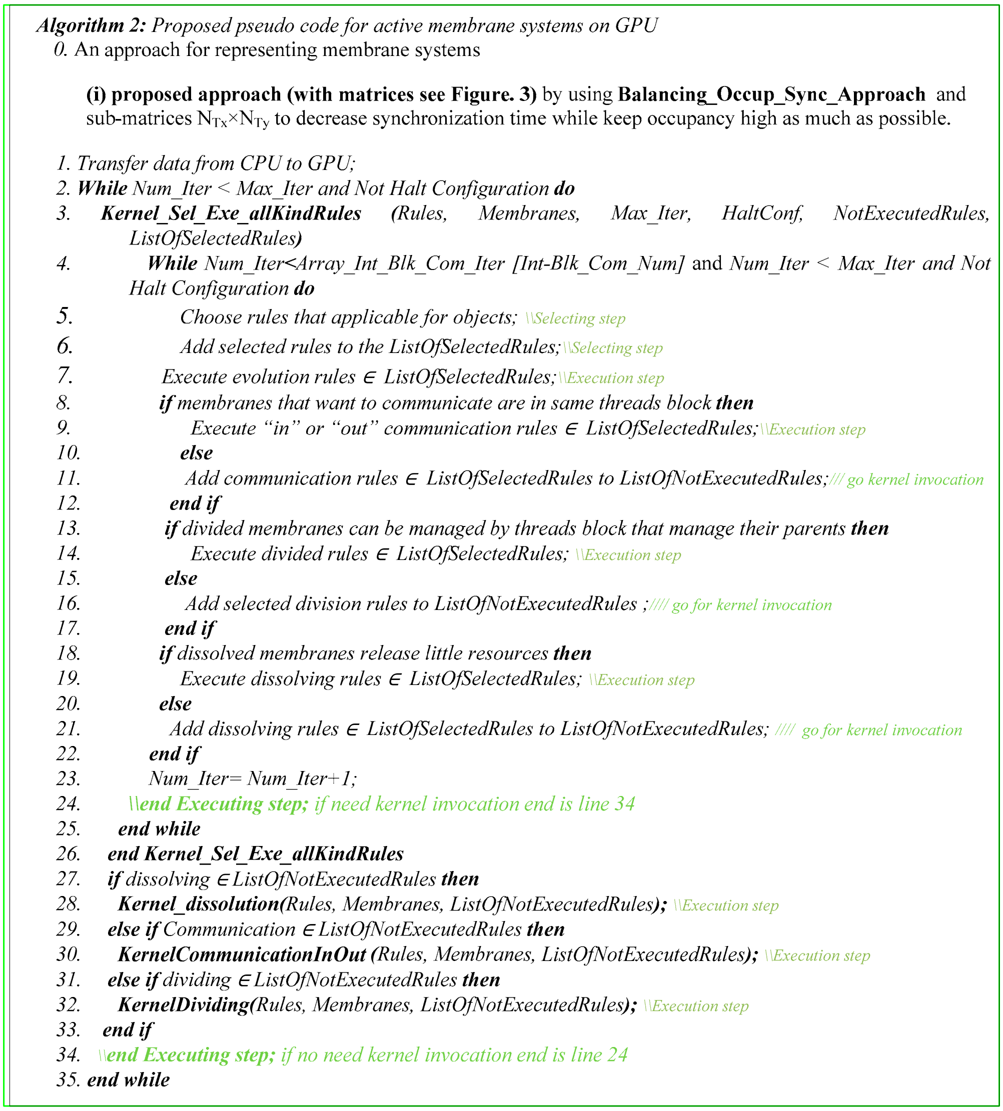


Figure 6. The proposed membrane-weighted network and classification. Membranes having higher communication rates are organized within the same groups (dashed ellipses). The number of membranes per group should satisfy Equation (4), and the membranes assigned to the sub-matrices, NTx × NTy, will be executed in the same thread block.

Previous approaches mapped one membrane to one thread block without considering the communication between membrane objects [30–33] (Figure 3). Given that each membrane was assigned to one block, communication among membranes necessitated re-invocation of separate kernels and decreased performance. According to Balancing_Occup_Sync_Approach, a trade-off is made between maintaining high occupancy and reducing communication rates. Object- and membrane-weighted networks are also proposed to automatically classify objects and membranes having shared dependencies within the same threads and thread blocks. These methods enhance algorithm performance by balancing occupancy, decreasing communication rates between thread blocks, and executing dissolution and dividing rules within the same kernel, thereby minimizing the need to re-invoke separate kernels.

3.2. Proposed Algorithm

The proposed algorithm for GPU simulation of active membrane systems is presented below (Figure 7). This algorithm represents membrane systems as matrices (Figure 7, line 0). Membrane objects are assigned to one matrix, with sub-matrices, NTx × NTy, from Equation (2), allowing maintenance of high occupancy levels due to their being grouped within individual thread blocks (Figure 4). The proposed approach uses Balancing_Occup_Sync_Approach to appropriately classify objects and membranes according to their communication rates by applying methods associated with an object- and membrane-weighted networks to the sub-matrices. This allows assignment of membranes having shared dependencies to the same thread block, enabling division and communication rules to be performed within the same kernel without needing to invoke others (Figure 7, lines 9, 14, and 19 are performed in the same kernel for selection and evolution rules, i.e., *Kernel_Sel_Exe_allKindRules*). When limited resources preclude assignment of all membranes having shared dependence on the same thread block, portions of the membranes can be assigned in ways that enhance thread-block communication rates. In these cases, inter-block communication occurs using separate kernels (Figure 7, lines 28, 30, and 32), but at decreased rates relative to previous approaches (Figure 3). When q inter-block communications are occurring for *MaxIter* steps, the time consumed for these processes is kept in an array (Array_Int_Blk_Com_Iter[q] = {Iteration of first inter-block communication, Iteration of the second inter-block communication, . . . , Iteration of the q^{th} inter-block communication}) [41]. Notice that for inter-block communication, the accurate time would be established by tracking the communication, aside from the dissolution and division rules that ensue among thread blocks. Entire thread blocks would be dismissed and synchronized by kernel invocation before initiation of thread-block communication. When it is not possible to determine the number of iterations necessary for thread-block communication, kernels invocation in any iteration are recommended.



Algorithm 2: *Proposed pseudo code for active membrane systems on GPU*

0. An approach for representing membrane systems

(i) proposed approach (with matrices see Figure. 3) by using **Balancing_Occup_Sync_Approach** and sub-matrices $N_{Tx} \times N_{Ty}$ to decrease synchronization time while keep occupancy high as much as possible.

```
1. Transfer data from CPU to GPU;
2. While Num_Iter < Max_Iter and Not Halt Configuration do
3.    Kernel_Sel_Exe_allKindRules (Rules, Membranes, Max_Iter, HaltConf, NotExecutedRules,
         ListOfSelectedRules)
4.       While Num_Iter<Array_Int_Blk_Com_Iter [Int-Blk_Com_Num] and Num_Iter < Max_Iter and Not
         Halt Configuration do
5.             Choose rules that applicable for objects; \\Selecting step
6.             Add selected rules to the ListOfSelectedRules;\\Selecting step
7.           Execute evolution rules ∈ ListOfSelectedRules;\\Execution step
8.           if membranes that want to communicate are in same threads block then
9.               Execute "in" or "out" communication rules ∈ ListOfSelectedRules;\\Execution step
10.            else
11.              Add communication rules ∈ ListOfSelectedRules to ListOfNotExecutedRules;/// go kernel invocation
12.          end if
13.          if divided membranes can be managed by threads block that manage their parents then
14.              Execute divided rules ∈ ListOfSelectedRules; \\Execution step
15.          else
16.              Add selected division rules to ListOfNotExecutedRules ;//// go for kernel invocation
17.          end if
18.          if dissolved membranes release little resources then
19.              Execute dissolving rules ∈ ListOfSelectedRules; \\Execution step
20.          else
21.            Add dissolving rules ∈ ListOfSelectedRules to ListOfNotExecutedRules; //// go for kernel invocation
22.        end if
23.        Num_Iter= Num_Iter+1;
24.     \\end Executing step; if need kernel invocation end is line 34
25.    end while
26.   end Kernel_Sel_Exe_allKindRules
27.   if dissolving ∈ListOfNotExecutedRules then
28.     Kernel_dissolution(Rules, Membranes, ListOfNotExecutedRules); \\Execution step
29.   else if Communication ∈ListOfNotExecutedRules then
30.     KernelCommunicationInOut (Rules, Membranes, ListOfNotExecutedRules); \\Execution step
31.   else if dividing ∈ListOfNotExecutedRules then
32.     KernelDividing(Rules, Membranes, ListOfNotExecutedRules); \\Execution step
33.   end if
34.  \\end Executing step; if no need kernel invocation end is line 24
35. end while
```

Figure 7. Pseudocode of the proposed approach for active membrane systems on GPU.

GPUs execute threads in warps in which each warp is a group of 32 threads. Thread discrepancy appears while threads in a warp follow dissimilar implementation paths as compared to others. Thus, when allocating objects to groups in which every group allocated to one thread, every group of

32 having a similar path would be allocated to 32 threads in a single warp whenever achievable to prevent branch deviation. In cases where branch deviation is inevitable, several approaches could be implemented to decrease its consequence, including branch distribution [42,43] and runtime data re-mapping across multiple warps [44]. In the algorithm presented here, dependent objects are assigned to the same thread whenever possible (Figure 7, line 0). This decrease both communication rates and synchronization time between threads compared to previous approaches that failed to consider shared dependence between objects before the thread assignment. Initial parameters are also assessed before execution to ensure that sufficient memory is allocated (Figure 7).

The initial information required concerning the active membrane system, $\pi = (O, H, \mu, w_0,\ldots, w_m, R)$, includes initial multisets, $(w_0,\ldots,w_m)$, the number of regions or membranes, m, membrane structure, μ, a group of rules, R, and *MaxIter*, a maximum number of iterations,. Halted conditions are encountered when *NumIter*, the number of iterations achieves *MaxIter,* or the configuration achieves a termination position (a condition in which no additional rules can be implemented).

Inputs are transmitted from host (CPU) to device (GPU), where the device begins execution of computations in parallel. This process comprises two main steps: rules selection and execution. When rules are selected (Figure 7, lines 5 and 6), the relevance of every rule is examined, including the accessibility of every object participating in the left-hand side of the rule (the reaction object). Additionally, for those rules requiring single or additional mutual objects, a single rule is selected non-deterministically, given that each membrane object can be used by at most one rule. Finally, all rules that are applicable are added to a list, *ListOfSelectedRules*, for the application in the rule-execution step. In the rule-execution step, all relevant rules are activated, and the quantity of objects, membranes, and other variables are revised consistent with the outcomes associated with rule implementation (while there is no inter-block communication and no demand for kernel invocation, this step implemented from lines 7 to 24 (Figure 7); else, this step is implemented from lines 7 to 34 (Figure 7).

Using this method, it is feasible to implement membranes and objects having shared dependencies (membranes that interact amongst each other or membranes produced through division rules) from within a similar thread block. Consequently, communication between thread blocks and kernel invocation is unnecessary, allowing the execution of rules incorporating evolution, communication ("in", "out"), dissolving, and division to be implemented using the similar kernel (Figure 7, *Kernel_Sel_Exe_allKindRules*) and intensifying execution. Both selection and execution steps for all rules can be accomplished in *Kernel_Sel_Exe_allKindRules* (Figure 7, lines 4 to 24) until a termination criterion is reached. Previous methods required the invocation of separate kernels before executing communication, dissolution, and division rules (Figure 3).

For dissolving rules (Figure 7, lines 18 to 20), the algorithm estimates the number of resources that will be discharged following membrane dissolution. While a single membrane is dissolved, the assigned threads will be idle. If the quantity of unemployed threads following dissolution is sizeable, it is advisable to re-invoke the kernel with no launching of unemployed threads or blocks. Else, the program resumes with no kernel re-invocation. The benefit of the proposed method is that the assignment of multiple membranes to one thread block enables other membranes to continue to exist in a given thread block following the dissolution of others. Previous methods wherein one membrane was assigned per thread block resulted in thread blocks being unused following the dissolution of the associated membrane.

4. Results and Discussion

The simulations designated in this section were implemented on a computer with an Intel Core i7-3820 and an NVIDIA GeForce GTX680 GPU. The code was designed using NVIDIA CUDA 4.1 and Visual C++ 2008. The program comprises of two parts: the host (CPU) and the device (GPU). The host/CPU portion of the code is responsible for controlling program execution, allocating memory in the host or device, and obtaining results from the device. GPU processing is scaled in seconds, and *MaxIter* is set high enough to assure transient factors do not affect GPU speed. Simulation timescales for different numbers of objects with the same iterations vary from milliseconds to hours. Therefore, various variable iterations are used to achieve the same CPU (sequential) time for all object combinations that have been considered (Table 1). However, a similar speedup is observed with *MaxIter* set to 10,000 for object combinations on the GPU, indicating that speed does not depend on *MaxIter*.

Simulations were undertaken to show the effect of mapping objects to different matrices (Figure 4) to decrease communication rates between threads. To perform this simulation, previous approaches were followed for defining a membrane system having objects with shared dependencies [31]. The membrane system with active membranes is defined as $\pi = (O, H, \mu, \omega_0, \ldots, \omega_m, R)$, with m elementary membranes located within the skin along with the set of rules

$$\mu = [[\,]_0, [\,]_1, \ldots, [\,]_{m-1}]_m \tag{10a}$$

and:

$$R : \left[a_i b_i^2 c_i^3 \rightarrow a_i^2 b_i^4 c_i^6\right]_h,\ i = 0, \ldots, n-1; h = 0, \ldots, m-1. \tag{10b}$$

The preliminary 3 × n objects (specifically, $a_i, b_i, c_i; i = 0, \ldots, n-1$) inside of the membrane ω_i, $0 \leq i < m$, are produced arbitrarily, the quantity of objects within the skin, ω_m, is zero, and dependencies exist between objects $a_i, b_i, c_i; i = 0, \ldots, n-1$. This membrane system allows control of the number of rules, n, and the number of different types of objects, $3n$, in each membrane to measure algorithm performance against these variables. Furthermore, it is also possible to change the number of membranes, m, for simulations. The initial multiplicities for objects, $a_i, b_i, c_i; i = 0, \ldots, n-1$, are randomly generated in each membrane, e.g., a_1^2 denotes that two objects from a_1 existing, meaning that the multiplicity of a_1 is 2. The number of each object is a random integer between 0 and U_{nOb}, where U_{nOb} is a user-defined integer (for these simulations, $U_{nOb} = 10$). For example, given six objects, $3n$ ($n = 2$) objects, $a_i, b_i, c_i; i = 0, \ldots, n-1$ (i.e., $a_0, b_0, c_0, a_1, b_1, c_1$), can be available in each membrane (Table 1). The initial multiplicity of each object is a random number between zero and U_{nOb}, where a multiplicity equal to zero signifies that the object is not available in the membrane in the initial step. The number of membranes, m, is 16,382 but is capable of being set differently. This system contains dependencies between objects, a_i, b_i, c_i, since producing objects $a_i^2 b_i^4 c_i^6$ according to the rules, $R : \left[a_i b_i^2 c_i^3 \rightarrow a_i^2 b_i^4 c_i^6\right]_h$, depends upon consuming $a_i b_i^2 c_i^3$. These rules can be applied in each step, given that the objects required for their application are available and the communication rates between $a_0, b_0, c_0, a_1, b_1, c_1$ and a_i, b_i, c_i are one. Simulating membrane systems over small, sequential timespans enables discovery of the average application rate associated with each rule per step, which can subsequently be used to obtain the communication rate between objects and measure the classification for parallel implementation.

4.1. Comparison Between Previous and Proposed Methods

Comparisons between previous methods (Figures 2 and 3) and the proposed method are undertaken by mapping several matrices (Figures 4 and 7) and using the classifications of dependent objects within the active membrane systems as illustrated (Table 1, Figure 8).

The size of the block is a vital aspect of achieving improved performance in algorithms-by-blocks. Sub-optimal performance occurs when the unconventional techniques are utilized to minimize data transfers, to enhance data affinity and an incorrect selection of parameters. In this study, the maximum value for the block size is a trade-off between several aspects such as the capability of parallelism in which smaller block size is transformed into a better concurrency, and a larger block component results in less concurrency.

Table 1. Comparing the proposed approaches with decreased communication for 16,382 membranes, different objects inside each membrane for the system as in Equation (10).

Iterations, Objects, CPU Time			Previous Approaches on the GPU (Figure 3) [31–33]					Proposed Approach with Matrices on the GPU (Figure 7)				Proposed Approach with One Membrane/ Thread Block	
Max Iteration	**Objects/Membranes (n × 3)**	**CPU Time (s)**	**Membrane/Block**	**Active Threads/Block (Objects in the Membrane)**	**GPU Occupancy (%)**	**Time (s)**	**Speed up**	**Membrane/Block**	**Optimum (N_{Tx}, N_{Ty})**	**Time (s)**	**Speed Up**	**Time (s)**	**Speed Up**
780,000	6	1000	1	6	25	7187	0.14	64	(2, 64)	11	89	807	8.9
390,000	12	1000	1	12	25	6284	0.16	32	(4, 32)	10	91	683	9.2
180,000	24	1000	1	24	25	5157	0.2	16	(8, 16)	10	92	526	9.8
92,000	48	1000	1	48	50	618	1.6	8	(16, 8)	10	93	59	10.4
43,000	96	1000	1	96	75	346	2.8	4	(32, 4)	10	99	30	11.5
21,000	192	1000	1	192	100	188	5.3	2	(64, 2)	9.4	106	16	11.7
9500	384	1000	1	384	100	121	8.2	1	(128, 1)	8.3	120	10	12.1
4700	768	1000	1	768	100	119	8.3	1/2	(128, 1)	8.2	121	9.8	12.1
2300	1536	1000	1	1024	100	118	8.4	1/4	(128, 1)	7.6	130	9.7	12.1

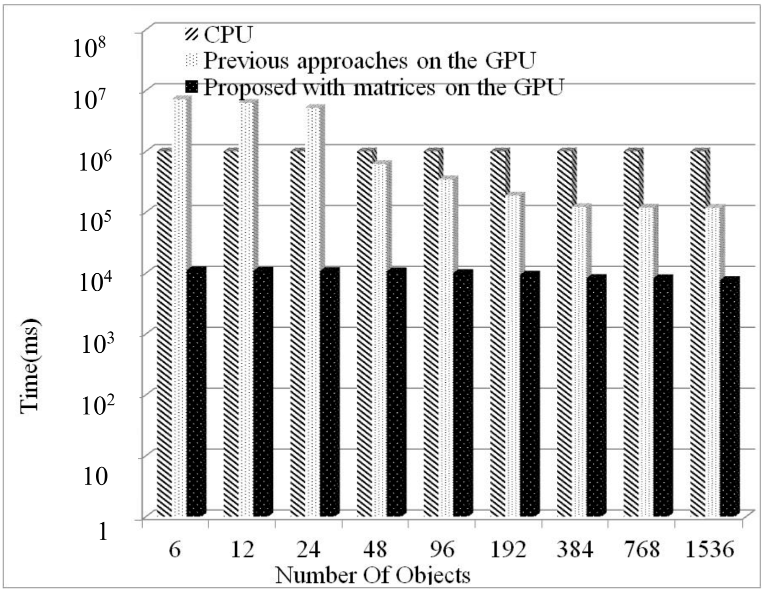


Figure 8. Comparison of previous and new methods for GPU optimization of active membrane systems. Execution times are associated with previous methods (Figure 3) and the algorithm presented in this paper (Figure 7).

Previous approaches assigned membrane objects to one thread block. Therefore, the number of assigned membranes per thread block for all objects was 1, and the maximum number of active membranes per thread was equal to the number of objects per membrane (Table 1). For a GPU with a computational capability of 3, the maximum number of threads per SM is 2048 ($MaxThr_{sm}$ = 2048), the maximum warp per SM is 64 ($MaxWarp_{sm}$ = 64), and the maximum number of thread blocks per SM is 16 ($MaxBlk_{sm}$ = 16). Therefore, the $MinThrBlk_{HighOccup}$ from Equation (1) is 128. Each warp contains 32 threads. The occupancy is the ratio of active warps per SM to the highest achievable warps that can be resident to the SM. To achieve maximum occupancy, at least four warps should be active per thread block (MaxWarpsm/$MaxBlk_{sm}$ = 4).

According to Table 1, for 6, 12, and 24 objects (threads), one warp was required, for 48 objects, two warps were necessary, for 96 objects, three warps were needed, and for more than 96 objects, four warps were required, resulting in occupancies of 25%, 50%, 75%, and 100%, respectively. With one membrane allocated to each thread block, when the quantity of objects per membrane increases, both the active threads per thread block and the occupancy increase due to the increased speed. Assumed that the collection of implementation units accessible within the system, the size of the block significantly affects the degree of parallel implementation on the sub-problems and decreases the unoccupied times. In data transfer effectiveness, a small block size is transformed into a larger quantity of data transmissions of a condensed dimension. On the contrary, a greater block dimension transforms into better efficient bandwidth and consequently benefits the ultimate performance of the application. Therefore, considering merely the possible parallelism aspect, the optimum block size somewhat relies on the entire matrix dimension.

Since there is no inter-block communication present in the membrane system described in Equation (10), the number of active threads per thread block, Thr_{Blk}, and the dimensionality of the sub-matrices according to Table 1, Equations (1) and (2) is NTx × NTy = $Thr_{Blk} = MinThrBlk_{HighOccup}$ = 128. The number of active threads per thread block does not depend on the quantity of objects per membrane. When the number of objects is low, more than one membrane is assigned to each thread block to achieve $MinThrBlk_{HighOccup}$ = 128 active threads per thread block. For this reason, occupancy, regardless of the quantity of objects is 100% (when adequate membranes are available), as opposed to results seen from previous approaches.

The number of objects per thread, Obj_{Thr}, equal to $NumSubMat_{Blk}$ can be obtained from Equation (3). In Equation (3), $ObjThr_{HighOccup}$ is calculated from Equation (5). In the simulation, when a membrane with 128 objects is assigned to 128 threads in every thread block, the amount of consumed shared memory per thread block is 2560 bytes (Sh_{Blk} = 2560), the number of used registers per thread is 10 (Reg_{Thr} = 10), and the amount of used local memory per thread is zero (Loc_{Thr} = 0) (values were obtained by compiling the code in CUDA using the PTXAS option). The used shared memory per thread, Sh_{Thr}, is obtained by dividing Sh_{Blk} by the number of threads per thread block, giving Sh_{Thr} = 2560/128 = 20 bytes. For the estimated memory usage and since one thread is assigned per object, $AveLoc_{Obj} = Loc_{Thr} = 0$, $AveSh_{Obj} = Sh_{Thr}$ = 20 bytes, and $AveReg_{Obj} = Reg_{Thr} = 10$. The values $MaxThr_{sm} = 2048$, $MaxLoc_{sm}$ = 512 KB, $MaxReg_{sm}$ = 64 KB, and $MaxSh_{sm}$ = 48 KB were obtained from [40], corresponding to a GPU computational capability of 3. By applying these values to Equations (6)–(8), $ObjThr_{ConsSh}$ = 2.4 KB, $ObjThr_{ConsLoc} = \infty$ (unconstrained by this factor), and $ObjThr_{ConsReg} = 32$. According to Equation (5), $ObjThr_{HighOccup} = 32$. The objects sharing dependencies in the membrane system from Equation (10) are a_i, b_i, and c_i. Therefore, $MaxDep_{Obj} = 3$. With $ObjThr_{HighOccup} = 32$, and $MaxDep_{Obj} = 3$, according to Equation (3), $NumSubMat_{Blk} = Obj_{Thr} = 3$.

Given these results, n objects from each membrane are assigned to $n/3$ threads. According to Step 2 of Balancing_Occup_Sync_Approach (Figure 7, line 0), all three dependent objects are assigned to one thread. For a case involving six objects within a membrane, all six would be assigned to $N_{Tx} = 2$ threads, meaning that $N_{Ty} = 64$ membranes are needed for assignment to the same thread block, resulting in $NTx \times NTy \approx 128$ threads according to Equation (2). Communication between threads is a very time-consuming process. Since objects having shared dependencies can be executed on the same thread using the method described here, the communication rate between threads decreases, thereby increasing performance. For example, given 1536 objects, the increase in speed achieved by previous approaches and relative to sequential-computation methods is 8.4-fold. In contrast, the method described in this paper utilizing multiple matrices results in a 130-fold increase.

When the most favorable block size is restored for a particular matrix size, the advantages of the several implementations of the runtime are predominantly determined by the quantity of objects and membranes necessary to perform a given operation. For "Balancing_Occup_Sync_Approach" (Figure 7), the inputs (the number of objects, membranes, and rules) are determined when the membrane system is designed. If there are errors in these inputs, especially on the rules, the system could behave abnormally, and the expected output won't be generated. As a precaution to avoid errors on the input, we validated the algorithm before it was implemented on the GPU. In this study, the processing time depends on communication between objects and membranes that are triggered by the rules. As Table 1 and Figure 8 demonstrated, when there is maximum occupancy with minimal communication between thread blocks, time efficiency and performance improved.

In cases involving large numbers of objects per membrane, occupancies resulting from the application of either method eventually reach 100% (Table 1). In these situations, the impact associated with decreasing communication rates between threads through the use of matrices, wherein objects having shared dependencies are assigned to the same threads can be observed. Given 384 objects, the increase in speed relative to sequential processing methods and using previous algorithms is 8.2-fold compared to a 106-fold increase using the method described here. This translates to a 12.9-fold increase in processing speed achieved by this algorithm relative to the algorithm used previously, mainly based upon improving communication between threads without adversely impacting occupancy. GPUs are low-cost, low-power consumption, and high performance concerning conventional multiprocessors. Many current desktop computers have equipped with the GPU enable graphics cards, which can improve the performance of processing without additional costs [28,45]. Thus achieving speed up even around 5× by GPU can be valuable work.

The results are quantified in the last two columns of Table 1 and confirm our prediction for improvement because of decreasing communication between threads inside each membrane discussed in this paragraph.

The proposed object-weighted network, membrane-weighted network, and classification algorithms are not constrained and limited to the specific type of membranes. Membranes can have different objects with different sizes and various types of rules. Object classification can be applied to each membrane individually. The membrane classification can be applied between membranes with different sizes. Thus, proposed approaches are not only for some particular class and regular membrane systems.

The dimension of sub-matrices depends on the number of objects in the membrane. When the number of objects increases, the dimension of sub-matrices will also increase. Table 1 and Figure 8 shows that when the number of objects increases, the performance of the proposed approach is also improving. Typically, a better parallel speedup is obtained for large problem sizes. The efficiency of the parallel program increases with the size of the matrix.

4.2. *Effects of the Number of Objects and Membranes*

In this section, the significance of the quantity of objects in every membrane and the quantity of membranes in the system on the GPU effectiveness with previously implemented approaches (Figure 3) are discussed. It is disclosed that while the quantity of objects per membrane and the quantity of membranes in an active membrane system are sizable, the performance in terms of time efficiency attained by the GPU related to a CPU is high. The reason is that once the quantities of objects and membranes increase, the utilization of the computational resource of the GPU also surges. Additionally, by seizing the benefit of the shared memory, the simulation speediness is amplified, because by gaining access to the shared memory, it is significantly quicker compared to the global memory. The quantity of membranes and the quantity of objects in every membrane can be altered.

Consequently, the proportion of the utilization of computational resources could also be altered. To use all the computational resources of the GPU, the quantity of objects and the quantity of membranes should be big. Simulation results (milliseconds) for a defined benchmark for 1000 iterations (*MaxIter*) on a computer with an Intel Core-i7-3820, 3.60-GHz CPU with 8 GB RAM and an NVIDIA GTX 680 GPU are illustrated in Tables 2 and 3.

Table 2. Comparison of CPU and GPU performance for different numbers of objects inside each membrane for a model with 8192 membranes in algorithm 1 (Figure 3) [41].

Number of Objects	Sequential on CPU (ms)	GPU Shared Memory (ms)	GPU Global Memory (ms)	Speedup GPU Shared Memory	Speedup GPU Global Memory
2	62	129	145	0.48	0.42
4	140	130	157	1.07	0.89
8	296	131	186	2.25	1.59
16	608	133	236	4.57	2.57
32	1217	135	240	9.01	5.07
64	2449	137	248	17.87	9.87
128	4898	140.6	319	34.83	15.35
256	10,077	264	652	38.17	15.45
512	20,155	518	1244	38.90	16.20
1024	40,300	1067	2520	37.76	15.99

Table 3. Comparison of CPU and GPU performance for different numbers of membranes for a model with 512 objects [41].

Number of Membranes	Sequential on CPU (ms)	GPU Using Shared Memory (ms)	GPU Using Global Memory (ms)	Speedup GPU Shared Memory	Speedup GPU Global Memory
2	4.7	2.01	2.02	2.3	2.3
4	16	2.05	2.3	7.8	6.9
8	31	2.15	2.7	14.4	11.4
16	47	2.25	3.4	20.8	13.8
32	78	2.37	5.6	32.9	13.9
64	156	4.7	11.1	33.1	14.0
128	296	8.5	20.9	34.8	14.1
256	624	17.4	44	35.8	14.1
512	1201	35.07	84.4	34.2	14.2
1024	2449	69.26	167	35.3	14.6
2048	4914	137.7	331	35.6	14.8
4096	10,031	272	637	36.8	15.7
8192	20,161	520	1252	38.7	16.1
16,384	40,286	1033	2488	38.9	16.1
32,768	80,589	2047	4977	39.3	16.1

As disclosed in Table 2, while the quantity of objects n in every membrane m, is small, for instance, $n = 2$ and $m = 8192$, the CPU demonstrates better performance with shorter implementation time compared to the GPU. When the quantity of objects is increased to, for instance, $n = 512$, subsequently, the GPU has improved performance compared to CPU. This investigation utilizes the benefit of shared memory that faster compared to when operating merely with global memory. With shared memory, improved performance can be attained. Objects, intermediary outcomes for regulating the relevance of rules, and consequences of rule implementations that have an effect on objects are kept in shared memory. Notice that while accessible shared memory is not sufficient for keeping objects and intermediary outcomes of employing rules in a thread block, it is prone to distribute data in global memory into smaller sub-blocks so that in each phase, one sub-block loaded into shared memory can utilize the fast-memory-access benefit of shared memory.

Table 3 illustrates the effect of the number of membranes on GPU performance. When the number of membranes increases, then the usage of GPU computational resources increases also, and the GPU has better performance than the CPU.

5. Conclusions

Previous studies mapped each membrane and its corresponding objects to one thread block and the associated threads, resulting in decreased efficiency when the number of membrane objects was smaller than the number of threads per thread blocks required to achieve maximum occupancy. Kernel invocation, which is a time-consuming process, was needed when communication and division rules happen between membranes. In previous approaches, each membrane was assigned to one thread block, and synchronization required re-invocation of separate kernels. Here, active membrane systems were modeled as matrices divided into sub-blocks according to the described Balancing_Occup_Sync_Approach and using object- and membrane-weighted networks to match the number of threads in each thread block, ensuring full thread utilization and decreased communication rates between threads and thread blocks. This approach alleviates problems encountered in previous studies by assigning several membranes to one thread block when the number of membrane objects is small. Our method increases processing speeds 93-fold as compared to sequential methods simulating active membrane systems with 48 objects (Table 1). Given the many similarities between membrane systems, the algorithm presented in this paper for active membrane systems involving GPUs should be transferable for application in other membrane-system variations. Moreover, the concepts described and associated with weighted-network classification can also be used in different fields to improve GPU simulations. We acknowledge the essentials of analyzing the gaps between theoretical speedup limitations and experimental results of GPU implementation of membrane computing. Still, unfortunately, it wasn't under the scope of this study, and it will be considered in our future works.

Author Contributions: Conceptualization, A.M. and R.C.M.; Methodology, A.M. and R.C.M; Visualization, A.M.; Writing—original draft preparation, R.C.M., and A.M.; Writing—review and editing, A.M. and R.C.M.; Funding acquisition, R.C.M. All authors have read and agreed to the published version of the manuscript.

Funding: This paper is supported by University Kebangsaan Malaysia (UKM), UKM Grant Code: GGP-2019-023.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Zhang, G.; Pérez-Jiménez, M.J.; Gheorghe, M. *Real-life Applications with Membrane Computing*; Springer Science and Business Media LLC: Berlin, Germany, 2017.
2. García-Quismondo, M.; Hintz, W.D.; Schuler, M.S.; Relyea, R.A. Modeling diel vertical migration with membrane computing. *J. Membr. Comput.* **2020**, *2*, 1–16. [CrossRef]
3. Song, T.; Pang, S.; Hao, S.; Rodríguez-Patón, A.; Zheng, P. A Parallel Image Skeletonizing Method Using Spiking Neural P Systems with Weights. *Neural Process. Lett.* **2018**, *50*, 1485–1502. [CrossRef]
4. Wang, B.; Chen, L.; Wang, M. Novel image segmentation method based on PCNN. *Optics* **2019**, *187*, 193–197. [CrossRef]
5. Orozco-Rosas, U.; Montiel, O.; Sepúlveda, R. Mobile robot path planning using membrane evolutionary artificial potential field. *Appl. Soft Comput.* **2019**, *77*, 236–251. [CrossRef]
6. Andreu-Guzmán, J.A.; Valencia-Cabrera, L. A novel solution for GCP based on an OLMS membrane algorithm with dynamic operators. *J. Membr. Comput.* **2019**, *2*, 1–13. [CrossRef]
7. Liu, C.; Du, Y. A membrane algorithm based on chemical reaction optimization for many-objective optimization problems. *Knowledge-Based Syst.* **2019**, *165*, 306–320. [CrossRef]
8. Liu, C.; Du, Y.; Li, A.; Lei, J. Evolutionary Multi-Objective Membrane Algorithm. *IEEE Access* **2020**, *8*, 6020–6031. [CrossRef]
9. Maroosi, A.; Muniyandi, R. Accelerated execution of P systems with active membranes to solve the N-queens problem. *Theor. Comput. Sci.* **2014**, *551*, 39–54. [CrossRef]
10. Maroosi, A.; Muniyandi, R.; Sundararajan, E.; Zin, A.M. A parallel membrane inspired harmony search for optimization problems: A case study based on a flexible job shop scheduling problem. *Appl. Soft Comput.* **2016**, *49*, 120–136. [CrossRef]
11. Paun, G. Tracing some open problems in membrane computing. *Rom. J. Inf. Sci. Tech.* **2007**, *10*, 303–314.
12. Zhang, G.; Perez-Jimenez, M.J.; Gheorghe, M. Membrane Computing—Key Concepts and Definitions. *Recent Adv. Theory Appl. Fit. Landsc.* **2017**, *25*, 1–9. [CrossRef]
13. García-Quismondo, M.; Gutiérrez-Escudero, R.; Pérez-Hurtado, I.; Pérez-Jiménez, M.J.; Riscos-Núñez, A. An overview of P-Lingua 2.0. In Proceedings of the WMC'09: Proceedings of the 10th international conference on Membrane Computing, Curtea de Arges, Romania, 24–27 August 2009; pp. 264–288.
14. Maroosi, A.; Muniyandi, R.C. Accelerated Simulation of Membrane Computing to Solve the N-queens Problem on Multi-core. In *Proceedings of the Haptics: Science, Technology, Applications*; Springer Science and Business Media LLC: Berlin, Germany, 2013; Volume 8298, pp. 257–267.
15. Maroosi, A.; Muniyandi, R.C. Membrane computing inspired genetic algorithm on multi-core processors. *J. Comput. Sci.* **2013**, *9*, 264–270. [CrossRef]
16. Orozco-Rosas, U.; Picos, K.; Montiel, O. Hybrid Path Planning Algorithm Based on Membrane Pseudo-Bacterial Potential Field for Autonomous Mobile Robots. *IEEE Access* **2019**, *7*, 156787–156803. [CrossRef]
17. Perez-Hurtado, I.; Perez-Jumenez, M.; Zhang, G.; Orellana-Martín, D. Simulation of Rapidly-Exploring Random Trees in Membrane Computing with P-Lingua and Automatic Programming. *Int. J. Comput. Commun. Control.* **2018**, *13*, 1007–1031. [CrossRef]
18. Ciobanu, G.; Wenyuan, G. P Systems Running on a Cluster of Computers. In *Computer Vision*; Springer Science and Business Media LLC: Berlin, Germany, 2004; Volume 2933, pp. 123–139.

19. Kulakovskis, D.; Navakauskas, D. Automated Metabolic P System Placement in FPGA. *Electr. Control. Commun. Eng.* **2016**, *10*, 5–12. [CrossRef]
20. Quiros, J.; Verlan, S.; Viejo, J.; Millan, A.; Bellido, M.J. Fast hardware implementations of static P systems. *Comput. Inform.* **2016**, *35*, 687–718.
21. Maroosi, A.; Muniyandi, R. Enhancement of membrane computing model implementation on GPU by introducing matrix representation for balancing occupancy and reducing inter-block communications. *J. Comput. Sci.* **2014**, *5*, 861–871. [CrossRef]
22. Ravie, C.; Ali, M. Enhancing the Simulation of Membrane System on the GPU for the N-Queens Problem. *Chin. J. Electron.* **2015**, *24*, 740–743. [CrossRef]
23. Martínez-Del-Amor, M.A.; García-Quismondo, M.; Macías-Ramos, L.F.; Valencia-Cabrera, L.; Riscos-Núñez, A.; Perez-Jimenez, M.J. Simulating P Systems on GPU Devices: A Survey. *Fundam. Inform.* **2015**, *136*, 269–284. [CrossRef]
24. Valencia-Cabrera, L.; Martínez-Del-Amor, M.Á.; Pérez-Hurtado, I. *A Simulation Workflow for Membrane Computing: From MeCoSim to PMCGPU Through P-Lingua*; Springer Science and Business Media LLC: Berlin, Germany, 2018; pp. 291–303.
25. Fujita, K.; Okuno, S.; Kashimori, Y. Evaluation of the computational efficacy in GPU-accelerated simulations of spiking neurons. *Computing* **2018**, *100*, 907–926. [CrossRef]
26. Idowu, R.K.; Muniyandi, R. Enhanced Throughput and Accelerated Detection of Network Attacks Using a Membrane Computing Model Implemented on a GPU. In *Quality of Experience and Learning in Information Systems*; Springer Science and Business Media LLC: Berlin, Germany, 2018; pp. 253–268.
27. Martınez-del-Amor, M.A.; Riscos-Núnez, A.; Pérez-Jiménez, M.J. A survey of parallel simulation of P systems with GPUs. *Bull. Int. Membr. Comput. Soc. (IMCS)* **2017**, *3*, 55–67.
28. Raghavan, S.; Rai, S.S.; Rohit, M.; Chandrasekaran, K. GPUPeP: Parallel Enzymatic Numerical P System simulator with a Python-based interface. *Biosystems* **2020**, *196*, 104186. [CrossRef]
29. Martínez-Del-Amor, M.A.; Perez-Hurtado, I.; Orellana-Martín, D.; Perez-Jimenez, M.J. Adaptative parallel simulators for bioinspired computing models. *Futur. Gener. Comput. Syst.* **2020**, *107*, 469–484. [CrossRef]
30. Guerrero, G.D.; Cecilia, J.M.; García Carrasco, J.M.; Martínez del Amor, M.Á.; Pérez Hurtado de Mendoza, I.; Pérez Jiménez, M.J. Analysis of P systems simulation on CUDA. *Conf. Days Parallelism* **2009**, *20*, 289–294.
31. Cecilia, J.M.; Garcia, J.M.; Guerrero, G.D.; Martínez-Del-Amor, M.A.; Pérez-Hurtado, I.; Perez-Jimenez, M.J. Simulation of P systems with active membranes on CUDA. *Briefings Bioinform.* **2009**, *11*, 313–322. [CrossRef]
32. Cecilia, J.M.; García, J.M.; Guerrero, G.D.; Martínez-Del-Amor, M.A.; Perez-Hurtado, I.; Perez-Jimenez, M.J. Simulating a P system based efficient solution to SAT by using GPUs. *J. Log. Algebraic Program.* **2010**, *79*, 317–325. [CrossRef]
33. Cecilia, J.M.; Garcia, J.M.; Guerrero, G.D.; Martínez-Del-Amor, M.A.; Perez-Jimenez, M.J.; Ujaldon, M. The GPU on the simulation of cellular computing models. *Soft Comput.* **2011**, *16*, 231–246. [CrossRef]
34. Cheng, J.; Grossman, M.; McKercher, T. *Professional CUDA C Programming*; John Wiley & Sons, Inc.: Indianapolis, IN, USA, 2014.
35. Cui, X.; Chen, Y.; Mei, H. Improving performance of matrix multiplication and FFT on GPU. In Proceedings of the IEEE 2009 15th International Conference on Parallel and Distributed Systems, Shenzhen, China, 9–11 December 2009.
36. Huang, Z.; Ma, N.; Wang, S.; Peng, Y. GPU computing performance analysis on matrix multiplication. *J. Eng.* **2019**, *2019*, 9043–9048. [CrossRef]
37. Paun, G.; Rozenberg, G.; Salomaa, A. *The Oxford Handbook of Membrane Computing*; Oxford University Press: Oxford, UK, 2010.
38. Xiao, S.; Feng, W.-C. Inter-block GPU communication via fast barrier synchronization. In Proceedings of the 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), Atlanta, GA, USA, 19–23 April 2010; 2010; pp. 1–12. [CrossRef]
39. Sun, X.; Wu, C.C.; Chen, L.R.; Lin, J.-Y. Using Inter-Block Synchronization to Improve the Knapsack Problem on GPUs. *Int. J. Grid High Perform. Comput.* **2018**, *10*, 83–98. [CrossRef]
40. Nvidia Developer. CUDA C Programming Guide, Version 10.2. Available online: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (accessed on 4 August 2020).

41. Maroosi, A.; Muniyandi, R.; Sundararajan, E.; Zin, A.M. Parallel and distributed computing models on a graphics processing unit to accelerate simulation of membrane systems. *Simul. Model. Pr. Theory* **2014**, *47*, 60–78. [CrossRef]
42. Fukuhara, J.; Takimoto, M. Branch Divergence Reduction Based on Code Motion. *J. Inf. Process.* **2020**, *28*, 302–309. [CrossRef]
43. Kim, Y.; Kim, J.; Chae, D.; Kim, D.; Kim, J. μLayer: Low Latency On-Device Inference Using Cooperative Single-Layer Acceleration and Processor-Friendly Quantization. In Proceedings of the EuroSys '19: Proceedings of the Fourteenth EuroSys Conference 2019, Dresden Germany, 25–28 March 2019.
44. Lin, H.; Wang, C.-L.; Liu, H. On-GPU Thread-Data Remapping for Branch Divergence Reduction. *ACM Trans. Arch. Code Optim.* **2018**, *15*, 1–24. [CrossRef]
45. Gong, Q.; Greenberg, J.A.; Stoian, R.-I.; Coccarelli, D.; Vera, E.M.; Gehm, M.E. Rapid simulation of X-ray scatter measurements for threat detection via GPU-based ray-tracing. *Nucl. Instruments Methods Phys. Res. Sect. B Beam Interactions Mater. Atoms* **2019**, *449*, 86–93. [CrossRef]