Improved Implementation of Simulation for Membrane Computing on the Graphic Processing Unit

Ali Maroosi*, Ravie Chandren Muniyandi, Elankovan A. Sundararajan, Abdullah Mohd. Zin

Research Center for Software Technology and Management, Faculty of Technology and Information Science
Universiti Kebangsaan Malaysia, 43600 UKM Bangi, Selangor, Malaysia

Abstract

Membrane computing is a theoretical model of computation that inspired from the structure and functioning of cells. Membrane computing models naturally have parallel structure. Most of the simulations of membrane computing have been done in a serial way on a machine with a central processing unit (CPU). This has neglected the advantage of parallelism in membrane computing. This paper uses Graphic Processing Unit (GPU) as a parallel tool to implement membrane computing. The method minimizes data transferring which is time consuming procedure between the device and the host by processing all computing on GPU and transfer only the final results to CPU. Simulations show that speed increases up to 15 times compared to sequential simulation and using of shared memory increases speed, up to 38 times.

© 2013 The Authors. Published by Elsevier Ltd. Open access under CC BY-NC-ND license.
Selection and peer-review under responsibility of the Faculty of Information Science & Technology, Universiti Kebangsaan Malaysia.

Keywords: Membrane computing, Graphic Processing Unit, Parallel Processing;

* Corresponding author.
E-mail address: ali.maroosi@gmail.com
1. Introduction

Membrane computing is one of recent branch of computer science that introduced by Paun [1]. The basic model consists of a hierarchical structure composed of several membranes, embedded into a main membrane called the skin. Membranes divide the Euclidean space into regions that contain some objects that represented by symbols of an alphabet and evolution rules. Using these rules, the objects may evolve and/or move from a region to an adjacent region [2].

Software applications for membrane computing normally implement sequential algorithms simulation adapted to common CPU architectures [3-6]. These kinds of algorithms do not get performance when the problem size increases. To use the advantage of parallelism of membrane computing other efforts has been done to implement membrane computing on parallel tools. For example membrane computing has been implemented on computer clusters [7], reconfigurable hardware as in the field programming gateway (FPGA) [8] and GPUs [9-11].

However, previous approaches have some limitations and need more investigation. Using computer cluster is not cost effective. Programming and changing the code on FPGA is very time consuming procedure. Therefore, GPU introduces very economical and easy way for parallel processing compared to computer clusters and FPGA. In this paper variant of membrane computing i.e. active membrane computing is implemented on GPU. This implementation has better performance with respect to sequential implementation. Latency of shared memory is lower than global memory in GPU. For further improvement of implementation, this paper uses shared memory instead of global memory to store data for processing on GPU.

2. P systems with Active membranes

P systems with active membranes are formed by a membrane structure, where a label and a polarization is associated to each membrane. The model of P system with active membranes usually has different elements as shown in Fig. 1 (Further information about active membrane is described by Paun [12]).

![Fig. 1 Membrane computing structure](image-url)
(6) \( R \) is a finite set of developmental rules defined as follows:

(a) Object evolution rules: \([a \rightarrow u]^h_o \) for \( h \in H, \alpha \in \{+,-,0\} \) (electrical charges), \( a \in O \) and \( u \) is a string over \( O \) that describes a multi set of objects associated with membranes that depends on the label and the charge in the membranes;

(b) “In” communication rules: \( \{\alpha\}_{h} \rightarrow \{\beta\}_{a} \) for \( h \in H, \alpha, \beta \in \{+,-,0\}, a, b \in O \). An object is introduced in the membrane, possibly modified, and the initial charge \( \alpha \) is changed to \( \beta \);

(c) “Out” communication rules: \( \{\alpha\}_{a} \rightarrow \{\beta\}_{b} \) for \( h \in H, \alpha, \beta \in \{+,-,0\}, a, b \in O \). An object is released from the membrane, possibly modified, and the initial charge \( \alpha \) is changed to \( \beta \);

(d) Dissolving rules: \([a]_{\alpha} \rightarrow b \) for \( h \in H, \alpha \in \{+,-,0\}, a, b \in O \). A membrane with a specific charge is dissolved in reaction to an object (possibly modified);

(e) Division rules: \([a]_{\alpha} \rightarrow [b]_{\beta} \) for \( h \in H, \alpha, \beta, \gamma \in \{+,-,0\}, a, b, c \in O \). A membrane is divided into two membranes. The objects inside the membrane are replicated, except for \( a \), which may be modified in each membrane.

The rules are applied according to the following principles:

(1) All the elements which are not involved in any of the operations to be applied remain unchanged.

(2) Rules associated with label \( h \) are used for all membranes with this label, no matter whether the membrane is an initial one or it was generated by division during the computation.

(3) Rules from (a) to (e) are used as usual in the framework of membrane computing, i.e. in a maximal parallel way. In one step, each object in a membrane can only be used by at most one rule (non-deterministically chosen), but any object which can evolve by a rule must do it (with the restrictions indicated below).

(4) Rules (b) to (e) cannot be applied simultaneously in a membrane in one computation step.

(5) If a membrane is dissolved, its content (multi set and interior membranes) becomes part of the immediately external one. The skin is never dissolved neither divided.

### 3. Proposed Parallel implementation

#### 3.1. Graphics Processing Unit

GPUs constitute nowadays a solid alternative for high performance computing. The way GPUs exploit parallelism differ from multi-core CPUs, which raises new challenges to take advantage of its tremendous computing power. GPU is especially well-suited to address problems that can be expressed as data-parallel computations. GPUs can support several thousand of concurrent threads providing a massively parallel environment. This parallel computation model leads us to look for a highly parallel computational technology where a parallel simulator can run efficiently. The smallest parts of GPU are cores. A group of cores is named as streaming multiprocessors (SMP). Cores inside each SMP are synchronized to execute same instructions. Each SMP works asynchronously from other SMPs. Each core has a very small amount of memory that it is called local memory [13,14]. The smallest amount of shared memory is dedicated for each SMP, and all SMPs can access to a large amount of memory as Global RAM. From application point of view, instead of cores, SMP and group of SMP, the threads, block and kernel are used respectively. A program contains one or several kernels, and each kernel may contain one or more blocks. Each block is run on a single SMP and all threads within a block are able to use the same shared memory, and use barrier synchronization. Synchronization and sharing of shared memory are impossible across blocks. The programmer creates a program that is called a kernel. As illustrated in Fig.2 a kernel grid is subdivided into blocks and each block is subdivided into various threads.
3.2. Implementation of active membrane computing on GPU

Two kinds of implementation on GPU have been done in this paper. Implementation on GPU by using global memory and implementation on GPU that uses shared memory. It has been developed by using CUDA in Microsoft Visual Studio 2008 (C++) environment. The program is divided into two parts: the host (CPU side) and the device (GPU side). The host/CPU part of the code is generally responsible for controlling the program execution flow, allocating memory in the host or device/GPU, and obtaining the results from the device.

A simulation algorithm for GPU is given in Algorithm 1. Inputs are loaded first from the host; memory is allocated in the device (to receive the loaded inputs).

Algorithm 1: Pseudo code for active membrane systems on GPU

Step0: (Host and Device) Input data and initialize structure and content of membranes
Step1: (Device) Select the rules that can be applied
Step2: (Device) Execute applicable rules
Step3: (Device) Repeat step 1 to 3 till termination criteria is met
Step4: (Host and Device) Transfer output from Device to Host

For this algorithm we need input parameters as initial multi-sets \((w_1, \ldots, w_m)\), number of regions or membranes \((m)\), structure of membranes \((\mu)\) and termination criteria. Inputs are then moved from the host to the device and the device executes parallel computations on the input data. This procedure includes two main steps of selecting rules and executing rules. In our simulator each membrane assign to the block in GPU.

Each thread checks applicability of rules that are related to its object. Then if it is selected as the executable rule, it performs them in execution step. Finally, after finishing the execution result, it should be transferred to the Host. Transferring output data to the host of CPU memory across the peripheral component interconnect express (PCI-
Express) bus has negatively affected to the simulation time. So, this paper minimizes data transfer between the device and the host by transferring the processing of the output from host to GPU.

4. Simulations and results

This paper presents three types of simulations, i) sequential on CPU ii) parallel on GPU with using global memory and iii) parallel on GPU with using shared memory. We show how to improve the efficiency of execution membrane on the GPU by making full use of the GPU's computational resources. To evaluate the effect of using existing resources in GPU this paper defines the structure of membrane computing for benchmark as follows:

\[
evolverule: [a_i \rightarrow a_j] ; 0 \leq i < n, 0 \leq j < m
\]

where \( n \) is the number of rules inside each membrane and \( m \) is the number of membranes. In this structure, the number of membranes and number of objects in each membrane can be changed. Therefore the percentage of using computational resources can also be changed.

To use all computation resources of GPU, the number of objects and number of membrane should be large. Results (elapsed time in milliseconds) for simulation of defined benchmark for 1000 iterations on a computer with CPU Intel Core-i7-3820, 3.60GHz with RAM 8GB and GPU NVIDIA GTX 480 have been illustrated in Table.1,Table.2, Fig.4 and Fig.5. As shown in Fig. 4 when the number of objects \( (n) \) inside each membrane (in this simulation we consider the number of membranes \( m=8192 \)) is small for example \( n=2 \) CPU has better performance (smaller execution time) than GPU. When number of objects is high for example \( n=512 \) then GPU has better performance than CPU. This paper uses shared memory strategy. By this way we could improve performance of implementation on GPU because the latency of access to global memory is about 400-800 cycles while the latency of access to shared memory is about 8-22 cycles [13].

Table 1. Comparison of the performance of CPU and GPU for different number of objects inside each membrane for number of membrane equal to 8192

<table>
<thead>
<tr>
<th>Number of Objects</th>
<th>Sequential on CPU (ms)</th>
<th>GPU Shared Memory (ms)</th>
<th>GPU Global memory (ms)</th>
<th>Speed UP GPU Shared Memory</th>
<th>Speed UP GPU Global Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>62</td>
<td>129</td>
<td>145</td>
<td>0.48</td>
<td>0.42</td>
</tr>
<tr>
<td>4</td>
<td>140</td>
<td>130</td>
<td>157</td>
<td>1.07</td>
<td>0.89</td>
</tr>
<tr>
<td>8</td>
<td>296</td>
<td>131</td>
<td>186</td>
<td>2.25</td>
<td>1.59</td>
</tr>
<tr>
<td>16</td>
<td>608</td>
<td>133</td>
<td>236</td>
<td>4.57</td>
<td>2.57</td>
</tr>
<tr>
<td>32</td>
<td>1217</td>
<td>135</td>
<td>240</td>
<td>9.01</td>
<td>5.07</td>
</tr>
<tr>
<td>64</td>
<td>2449</td>
<td>137</td>
<td>248</td>
<td>17.87</td>
<td>9.87</td>
</tr>
<tr>
<td>128</td>
<td>4898</td>
<td>140.6</td>
<td>319</td>
<td>34.83</td>
<td>15.35</td>
</tr>
<tr>
<td>256</td>
<td>10077</td>
<td>264</td>
<td>652</td>
<td>38.17</td>
<td>15.45</td>
</tr>
<tr>
<td>512</td>
<td>20155</td>
<td>518</td>
<td>1244</td>
<td>38.90</td>
<td>16.20</td>
</tr>
<tr>
<td>1024</td>
<td>40300</td>
<td>1067</td>
<td>2520</td>
<td>37.76</td>
<td>15.99</td>
</tr>
</tbody>
</table>
In Fig. 4, the comparison execution time of membrane computing model on CPU and GPU with and without shared memory for different number of objects is shown.

In Fig. 5, the effect of number of membranes on performance of GPU has been considered. When number of membranes increases, then usage of computation resource of GPU increases and GPU has better performance than CPU.

![Fig. 4. Comparison execution time of membrane computing model on CPU and GPU with and without shared memory for different number of objects](image)

![Fig. 5. Comparison of the execution time of membrane computing model on CPU and GPU with and without shared memory for different number of membranes](image)

Table 2. Comparison of the performance of CPU and GPU for different number of membranes with number of objects equal to 512

<table>
<thead>
<tr>
<th>Number of membranes</th>
<th>Sequential on CPU (ms)</th>
<th>GPU Using Shared Memory (ms)</th>
<th>GPU Using Global Memory (ms)</th>
<th>Speed UP GPU Shared Memory</th>
<th>Speed UP GPU Global Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4.7</td>
<td>2.01</td>
<td>2.02</td>
<td>2.3</td>
<td>2.3</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>2.05</td>
<td>2.3</td>
<td>7.8</td>
<td>6.9</td>
</tr>
<tr>
<td>8</td>
<td>31</td>
<td>2.15</td>
<td>2.7</td>
<td>14.4</td>
<td>11.4</td>
</tr>
<tr>
<td>16</td>
<td>47</td>
<td>2.25</td>
<td>3.4</td>
<td>20.8</td>
<td>13.8</td>
</tr>
<tr>
<td>32</td>
<td>78</td>
<td>2.37</td>
<td>5.6</td>
<td>32.9</td>
<td>13.9</td>
</tr>
<tr>
<td>64</td>
<td>156</td>
<td>4.7</td>
<td>11.1</td>
<td>33.1</td>
<td>14.0</td>
</tr>
<tr>
<td>128</td>
<td>296</td>
<td>8.5</td>
<td>20.9</td>
<td>34.8</td>
<td>14.1</td>
</tr>
<tr>
<td>256</td>
<td>624</td>
<td>17.4</td>
<td>44</td>
<td>35.8</td>
<td>14.1</td>
</tr>
<tr>
<td>512</td>
<td>1201</td>
<td>35.07</td>
<td>84.4</td>
<td>34.2</td>
<td>14.2</td>
</tr>
<tr>
<td>1024</td>
<td>2449</td>
<td>69.26</td>
<td>167</td>
<td>35.3</td>
<td>14.6</td>
</tr>
<tr>
<td>2048</td>
<td>4914</td>
<td>137.7</td>
<td>331</td>
<td>35.6</td>
<td>14.8</td>
</tr>
</tbody>
</table>
5. Conclusion

This paper introduced the use of shared memory efficiently instead of global memory, and this has improved performance of execution. For number of objects that equal to 256, GPU with global memory and GPU with shared memory have 15.45 and 38.17 times speedup respectively compared to CPU implementation. In the future work, we want to use a matrix representation of membrane computing and by this way we decide to solve the limitation at the number of levels that should not have more than two levels in the structure of membrane computing to implement on GPU.

Acknowledgements

This work supported by the Exploratory Research Grant Scheme (ERGS) of the Ministry of Higher Education (Malaysia; Grant code: ERGS/1/2011/STG/UKM/03/)

References