Accelerating Cellular Automata Evolution on Graphics Processing Units

Luděk Žaloudek, Lukáš Sekanina, Václav Šimek
Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic
izaloude@fit.vutbr.cz, sekanina@fit.vutbr.cz, simekv@fit.vutbr.cz

***Abstract*—As design of cellular automata rules using conventional methods is a difficult task, evolutionary algorithms are often utilized in this area. However, in that case, high computational demands need to be met. This problem may be partially solved by parallelization. Since parallel supercomputers and server clusters are expensive and often overburdened, this paper proposes the evolution of cellular automata rules on small and inexpensive graphic processing units. The main objective of this paper is to demonstrate that evolution of cellular automata rules can be accelerated significantly using graphics processing units. Several methods of speeding-up the evolution of cellular automata rules are proposed, evaluated and compared, some with very good results. Also a comparison is made between mid-end and high-end graphics accelerator card based on the results of evolution speedup. The proposed methods are evaluated using two benchmark problems.**

Keywords–cellular automata; parallel computing; GPU; CUDA; genetic alghorithm

I. Introduction

The recent development of the SIMD-oriented general computation on Graphics Processing Units (GPUs) has motivated the research on new approaches to the acceleration of various computational models. Among others, accelerators of cellular automata (CA) and evolutionary algorithms (EA) have been proposed because of inherently parallel nature of these bio-inspired computing systems [1, 2, 6, 8].

Since their conception in 1950s [21], cellular automata have found many applications. These include physical systems modeling, road traffic simulation, random number generation, artificial life simulation, cracking of encryption standards [5, 7, 18, 20], etc. CA utilize several key features which make them unique computational models. Among these features are massive parallelism, locality of cell interactions, simplicity of basic building blocks and complex emergent behavior on a global level.

Because of inherent complexity, design of cellular automata is a difficult task for a human engineer. For example, Langton's self-replicating CA loops are based on identical cells; each of them has more than 280 transition rules [13]. In order to increase the efficiency when designing CA rules, evolutionary algorithms have been introduced to the field [14, 19]. By means of EA, the space of possible solutions to the problems of CA design may be explored efficiently. For example, Sipper has developed so-called "Cellular programming approach" [19] which allowed the CA rules to be evolved using a parallel cellular EA.

CA may be evolved either directly in hardware (such as FPGA) or in software, using simulators. This paper deals with the evolution of CA rules in a software CA simulator. However, design by EA is very computationally demanding. Not only is it necessary to simulate the CA which may consist of thousands of cells, but whole populations of CA have to be simulated and each CA may have many possible initial configurations, which need to be evaluated in order to determine the quality of a candidate solution.

One of possible ways to accelerate the CA simulation and, therefore, the execution of an EA is parallelization. Not surprisingly, there are some problems: Desktop CPUs with more than 6 cores are still not available (June 2010) and hi-end servers, supercomputers or computing clusters are highly expensive or overburdened if accessible. Since our interest lies with very large CA, we need a processing power capable of effectively accommodating hundreds of threads in order to justify the parallelization effort and the increased cost. However, with modern GPUs one can obtain computing power of supercomputers for a price of a hi-end PC.

The goal of the paper is to propose a GPU accelerator for evolutionary CA design. We will, in fact, propose and compare several architectures with the aim to identify the most efficient one. This paper extends our previous work [1] in the following aspects: (i) Models of CA and EA computation are presented in a greater detail. (ii) More experiments have been performed to evaluate the proposed architectures. (iii) We included another platform (9600 GT) for comparison. (iv) We have not investigated the speedup factor only; we have also measured the efficiency of the evolutionary algorithm using a simple benchmark problem.

The rest of the paper is organized as follows. Section II introduces one-dimensional cellular automata. Relevant evolutionary algorithms are to be briefly surveyed in Section III. Section IV describes the basic concept of general computing on GPUs. Section V proposes several methods how to utilize the parallel computing power of modern GPUs in CA rule evolution, whereas the benchmark problems – CA counter and majority – is defined in Section VI. Section VII describes in detail the experiments for the evaluation of the methods described in sections V and VI. Results of the conducted experiments are summarized in Section VIIII. While Section IX discusses obtained results, Section X concludes this paper and proposes several possibilities of further development.

II. 1D Cellular Automaton

A cellular automaton is an *n*-dimensional grid of identical cells, each working as a finite state automaton [3]. In its synchronous version, the state of the cells is periodically updated using a local transition function. If all the cells use the same local transition function, the automaton is known as uniform; otherwise, it is non-uniform. The next state of each cell is a function of its current state and the states of its neighboring cells. In case of 1D CA ($n = 1$), the neighborhood is defined using radius of r. In theory, the cellular automaton model supposes that the number of cells is infinite. However, in the case of practical applications the number of cells is finite. Then, it is necessary to define the boundary conditions, i.e. the setting of the boundary cells. Boundary conditions for the cells on the edges of the CA are usually either cyclic or constant (i.e., the states are taken from the opposite edge of CA or from the last cell). The state of the CA in the beginning of the run is called the initial configuration.

Experiments described in this paper deal only with 1D CA for simplicity and clarity purposes. The binary one-dimensional non-uniform CA of finite size may be described formally [17] as a 7-tuple $\mathbf{A} = (Q, N, R, z, b_1, b_2, c_0)$, where:

$Q = \{0, 1\}$ is a binary set of states,

N denotes a neighborhood ($N \subseteq \mathrm{Z}$),

z denotes the number of cells,

b_1 and b_2 are boundary values,

c_0 is an initial configuration, and

a mapping $R : S \rightarrow (Q^N \rightarrow Q)$ assigns to each cell of the grid $S = \{1, 2, \ldots, z\}$ a local transition function $\delta_1, \ldots, \delta_z$, where $\delta_i : Q^N \rightarrow Q$, $i \in S$.

A configuration of **A** is a mapping $c \in Q^S$ which assigns a state to each cell **A**. If only a single neighborhood $N = \{-1, 0, 1\}$ (i.e., $r = 1$) is considered, then the global transition function $G : Q^S \rightarrow Q^S$ is defined as:

$$G(c(i)) = \begin{cases} \delta_i(c(i-1), c(i), c(i+1)) & i = 2 \ldots z-1, \\ \delta_1(b_1, c(1), c(2)) & i = 1, \\ \delta_z(c(z-1), c(z), b_2) & i = z, \end{cases}$$

where c_i denotes the CA configuration in a step i. G is used to define a sequence of configurations $c_0, c_1, c_2, \ldots$ such that $c_j = G(c_{j-i})$, for $j \geq 1$. This sequence represents the *computation* of **A**.

Consider a uniform version of **A**, with the nearest neighbors neighborhood (i.e., $|N| = 3$) and cyclic boundary conditions. Each such cellular automaton is defined by a mapping $\{0, 1\}^N \rightarrow \{0, 1\}$ uniquely. Hence there are 2^8 such cellular automata, each of which is uniquely specified by the following (transition) rule

$000 \rightarrow a_0$
$001 \rightarrow a_1$
$010 \rightarrow a_2$
$011 \rightarrow a_3$
$100 \rightarrow a_4$
$101 \rightarrow a_5$
$110 \rightarrow a_6$
$111 \rightarrow a_7$.

We can speak of the cellular automaton with rule i, where i is an integer ($0 \leq i < 256$) with the binary representation $a_7a_6a_5a_4a_3a_2a_1a_0$. For example, Figure 1 shows the behavior of the cellular automaton with rule 150 that starts its computation from the initial configuration $c_0 = \ldots00100\ldots$ (the black square represents logic 1).

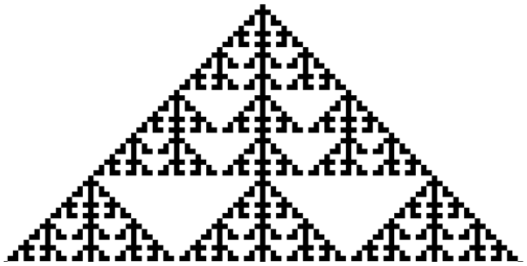

Figure 1. Development of 1D CA with rule 150.

The properties of cellular automata have been investigated by means of analytic as well as experimental methods. In general, the objectives are either to (i) find a method for the design of cellular automaton rules for a given application or (2) predict the global behavior of a given cellular automaton if the rules and the initial configuration are known. Because of the inherent complexity of cellular automaton, evolutionary design of CA rules has been adopted [19].

III. Evolutionary Algorithms

Evolutionary algorithms are stochastic search methods. They are inspired by Darwin’s theory of biological evolution. Instead of working with one solution at a time (as random search, hill climbing and other search techniques do), these algorithms operate with the population of candidate solutions (candidate CA rules in our case). Every new population is formed by genetically inspired operators such as crossover (a part of CA rules is taken from one parent, the rest from another one) and mutation (inversion of some bits of the CA rule) and through a selection pressure, which guides the evolution towards better areas of the search space. The EAs receive this guidance by evaluating every candidate solution to define its fitness value. The fitness value calculated by the fitness function indicates how well the solution fulfills the problem objective.

The most common form of EA is a genetic algorithm (GA) which has the following form:

1. create randomly initialized population of individual solutions
2. evaluate the population – assign the fitness value to each individual
3. select the best individuals based on their fitness value
4. apply genetic operators (crossover and mutation) on the selected individuals and create a new population
5. if termination criteria are met (fitness, number of generations), finish, otherwise continue with 2

In our case, a candidate solution will be encoded as a finite size binary string composed of substrings that define the transition function for every cell.

It has been shown that EA may generate innovative results in many fields. However, the scalability of representation and scalability of fitness calculation were identified as major problems of the evolutionary approach [22]. In this work, the scalability problem is approached using parallelization of the CA rules evolution.

IV. nVIDIA GPUs and CUDA

Although there are other universal computation-capable graphic accelerators with GPU programming interfaces such as ATI Stream from AMD, the most notable is nVIDIA with their Computer Unified Device Architecture (CUDA).

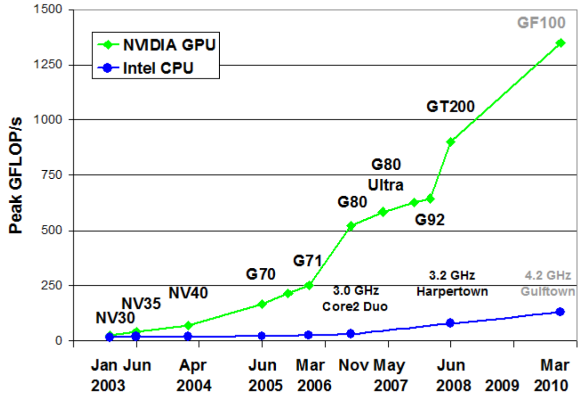

Figure 2. Performance of nVIDIA GPUs compared with Intel CPUs [15]. Supposed performance [16] is provided for the last chips as there are no official data yet (June 2010).

nVIDIA graphic accelerators contain GPUs with manycore streaming multiprocessors (MP) capable of outperforming general-purpose CPUs in some tasks (Figure 2). Each of these accelerators has from 4 to 30 multiprocessors with 8 scalar processor cores, two special units for transcendentals, a multithreaded instruction unit and on-chip shared memory (Figure 3). The multiprocessor creates, manages and executes concurrent threads in hardware with zero scheduling overhead [15].

Up until recently, direct programming for this hardware was not possible and indirect practices using OpenGL or DirectX had to be employed [11]. That changes with CUDA, which is a direct GPU programming interface.

CUDA works as a C language extension providing abstractions of thread groups, shared memories and barrier synchronization. This renders fine-grained data and thread level parallelism.

The code is separated into two classes: Host code which is executed on the CPU and device code which is executed on the GPU. Memory is differentiated in similar way, although it is possible to access device memory from host and vice versa trough the CUDA runtime library.

There are several types of device memory: Constant, shared, global, local and texture. Constant, texture and shared memory space is cached (4 clock cycles latency) but limited opposed to local and global memory (400 to 600 clock cycles latency) [15]. Effective usage of fast cached memory is key to high performance of the parallel application.

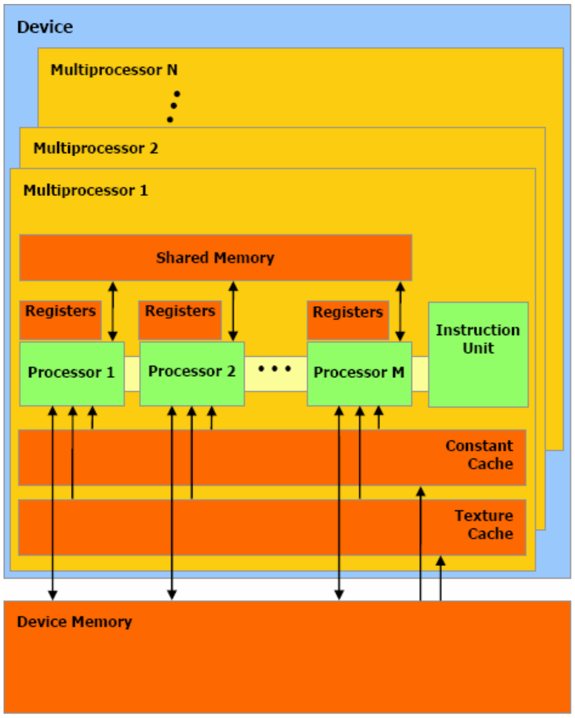

Figure 3. nVIDIA GPU structure [15]

Notable is the thread hierarchy used in CUDA programs: Threads may be arranged into blocks, where each block runs on one multiprocessor. It is possible to have more blocks than multiprocessors and more threads per block than cores. Shared memory may be accessible only within the block and thread synchronization is possible also only within the block. This is a possible drawback in some applications.

In recent years, CA have been implemented in GPUs, for example in [8]. As mentioned before, previous implementations of CA used Open GL or similar “shading” languages which brought several disadvantages: General purpose programming with Open GL or DirectX is overly complicated due to their specialization to computer graphics and also it does not enable direct control over the GPU’s parameters, possibly rendering the computations ineffective.

V. Proposed Methods of Parallelization

In order to parallelize a GA, a computational complexity of its components must be considered. In the case of evolutionary design of CA rules, the evaluation of candidate rules (fitness) is surely the most demanding part. CA consisting of possibly thousands of cells must be simulated for a pre-specified number of simulation steps. Furthermore, many possible initial CA configurations have to be evaluated.

When determining the quality of CA, e.g. in the majority task [19] (a benchmark task for a 2-state CA which determines whether the initial configuration contains more 1s then 0s by filling all the cells with the prevalent state after a number of steps), a 1D CA with 64 cells has 2^{64} possible initial configurations. This problem is dealt with by evaluating only several thousand randomly generated training vectors and measuring the success rate.

In GA, each generation has a population of possibly hundreds of individuals which further multiplies the number of calls to the fitness function.

Three possible approaches to parallelization will be proposed in following subsections.

A. *The Level of Cells*

First approach executes parallel lookup of transition rules in cells. There are as many threads as there are cells within the CA. The cell states are kept in the shared memory so the threads have to be synchronized after each step in order to guarantee the proper sequence of simulation steps.

The algorithm follows:

```
/*host part*/
Generate the initial CA configuration
Load the configuration into device
shared memory;
Load the CA transition rules into
device shared memory;

/*device part*/
For each thread do
  Repeat for S steps
    Compute transition function for
    one cell and update the cell;
    Synchronize the shared memory;

/*host part*/
Load the final CA configuration into
host memory;
```

This limits the algorithm only to one MP due to lack of synchronization between blocks. Graphical representation of the algorithm is shown in Figure 4.

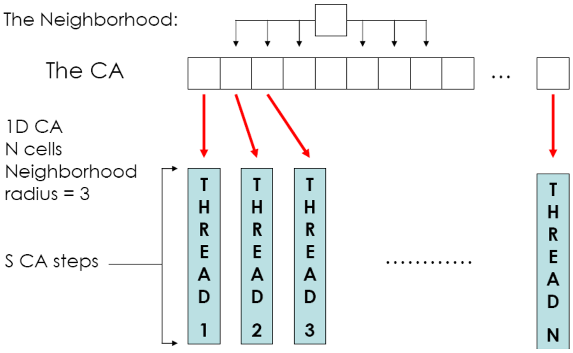

Figure 4. Simulation of CA with parallelization on the level of cells. Here, *S* denotes the number of CA simulation steps

B. *The Level of Training Vectors*

Second approach utilizes parallelization on the level of training vectors. There are as many threads as there are training vectors. Because there is no dependency between two same automata running two different simulations, it is possible to use more parallel blocks (i.e. multiprocessors) than one.

The algorithm ensues:

```
/*host part*/
Generate V initial configurations;
Load the CA configurations into device
global memory;
Load the CA transition rules into
device shared memory;

/*device part*/
For each thread do
  Load one CA configuration into
  registers/local memory;
  Simulate the CA for S steps;
  Calculate the fitness function;
  Update the fitness result into
  device global memory;

/*host part*/
Load the fitness results from device
global memory into host memory;
Calculate fitness for the individual;
```

Graphical representation of the main part of the algorithm is depicted in Figure 5.

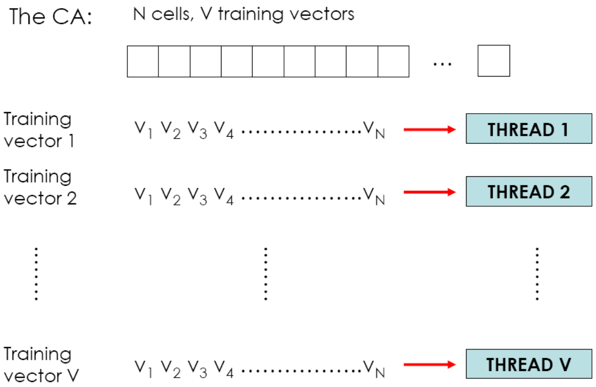
The CA:
N cells, V training vectors
...
Training vector 1
$V_1\ V_2\ V_3\ V_4$ ....V_N
THREAD 1
Training vector 2
$V_1\ V_2\ V_3\ V_4$ ....V_N
THREAD 2
Training vector V
$V_1\ V_2\ V_3\ V_4$ ....V_N
THREAD V

Figure 5. Parallelization of CA simulation on the level of training vectors. Here *V* denotes the number of training vectors per individual CA and *S* denotes the number of CA simulation steps

C. *The Level of Individual Solutions*

The last approach is to evaluate one individual per thread. There are as many threads as there are individuals within the GA population.

However, this approach is the most memory consuming, because we need to hold not only multiple CA configurations but also multiple transition rules.

The algorithm follows:

```
/*host part*/
Load the CA transition rules into
device global memory;

/*device part*/
For each thread do
  Load the CA transition rules into
  local memory;
  Generate training vectors;
  For each training vector do
    Simulate the CA for S steps;
    Update the fitness into device
    global shared memory;

/*host part*/
Load the fitness results from device
global memory into host memory;
```

Graphical representation of the algorithm is shown in Figure 6.

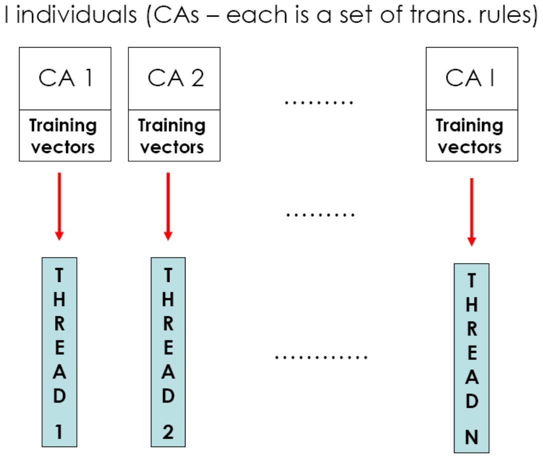

Figure 6. Parallelization of CA simulation on the level of individuals. I denotes the number of individuals and threads.

D. *Problems Without Training Vectors*

Previous subsections proposed parallelization approaches for problems which require evaluation with sets of training vectors. However, there are also problems which do not require such measures because the initial configuration of the CA is known. Such problems include applications as counters or random number generators, which were not mentioned in [1].

VI. Benchmark Problems

A. *4-bit Counter*

A counter is a device (in this case implemented by means of CA), which is able to generate a certain sequence of numbers or in this case a certain sequence of CA configurations. For example a 4-bit counter seeded by the value of 5 has to generate a sequence $seq^{(5)}$ = 5-6-7-8-9-10-11-12-13-14-15-0-1-2-3-4. The sequence is encoded in four cells, each with 2 possible states (0, 1).

The goal will be to evolve a simple 1D non-uniform CA which will generate the desired sequence. The CA will have a simple neighborhood with radius r = 1. The non-uniform CA was selected because non-uniform cellular automata enable us to perform more complex tasks than with uniform CA of the same number of cells [19]. Generating a certain sequence is not a simple task in this context.

Since training vectors aren't used, the proposed approach will be different from the scenarios assumed in Section V. The evaluation of the candidate solution works in the following way: The CA is simulated for 16 steps (because we have 16 numbers in the sequence) and in each step, the configuration is compared with a desired number. So for example, the second step configuration is compared with the value 6, the third step configuration with the value 7 etc. In the end, all 16 configurations should correspond with the 16 desired numbers. Each match is awarded with a point to the fitness value, so the maximum fitness is 16 and minimum is 1 (the initial configuration is counted automatically).

It is needed to keep as many rules as there are cells, because a non-uniform CA is used. A non-uniform CA does not need to have necessarily the same number of rules as the number of cells (some rules may apply for more than one cell) but in this case, the encoding is simple and keeping references to rules and interpreting them may prove unnecessary and too complex. Also there are no training vectors, so it is not possible to parallelize a single individual with the same set of transition rules which are placed in the shared memory within the same block. So either the cell parallelization approach has to be used (Section V.A) or an approach similar to the parallelization on the level of individuals mentioned in Section V.C. The latter approach seems more promising, so we will use it. Fitting more transition rules into the GPU memory several times could prove to be challenging but the CA has a simple neighborhood and 2 possible states, so in this case, it is not a serious problem.

B. *Majority*

The second benchmark problem is the majority task defined in Section V. A 2-state CA computing the majority task has to determine whether the initial configuration contains more 1s then 0s by filling all the cells with the prevalent state after a number of steps. Usually, the number of steps equals double the number of cells in the automaton and the quality of the solution is determined by randomly generating certain number of training vectors and measuring the proportion of successful CA runs [19].

VII. The Experiments

A. *The Evolution of CA Rules for the Majority Problem*

Several experiments were conducted in order to evaluate the speedup of proposed parallelization methods. Two different computers were used: (i) A laptop with Intel Core 2 Duo processor at 1.83 GHz with 667 MHz FSB and low-end nVIDIA GeForce 8600M GS graphic accelerator with 4 multiprocessors; (ii) A workstation with Intel Core 2 Duo processor at 3.33 GHz with 1333 MHz FSB and hi-end nVIDIA GeForce FX 285 (iia) graphic accelerator with 30 multiprocessors or a mid-end nVIDIA GeForce 9600 GT (iib) graphic accelerator with 8 multiprocessors, however (iib) was used only in later experiments described in Subsection B.

The programs were compiled with MS Visual C++ 9.0 compiler (serial versions) and with CUDA 2.3 SDK (parallel versions).

Each parallelization approach was tested on both computers and the execution times were compared with those of serial versions of the programs. This means the serial versions were run on a single CPU core without the use of GPU. The whole program execution time was measured.

The 1D 2-state uniform CA with 64 cells, r=3 (7 cell neighborhood) and cyclic edge conditions was used for the experiments. Each simulation lasted 128 synchronization steps. The CA rule is represented as string of 128 integers.

First approach (Section V.A) was tested both only on a number of CA simulations without the GA (and without fitness evaluation!) and with the GA, which was a standard algorithm with 10 generations, crossover rate of 70% and mutation rate of 1%. The crossover was one-point and the mutation probability is meant for a single gene (i.e. bit). Two-step tournament selection was used. Population size was 100 and 240 training vectors were used. Note that 10 generations are not sufficient to find a good solution, however we are interested in the speedup analysis only.

The rest of the experiments (based on Section V.B and V.C) were executed with the same GA. The size of the population and the number of training vectors varied for the last experiments (240 vectors and 100 individuals for the training vector approach and vice versa) but the number of fitness evaluations was proportionally the same. The number 240 was conveniently selected because the number of processing cores within the GTX 285 accelerator.

Additionally, several more experiments were conducted focusing on the scalability of the proposed algorithms. The same parameters were used except for the number of training vectors or individuals within the population. Sizes of the problems were increased up to hundredfold relative to the original problem sizes.

The fitness function of the GA consists of V simulation runs of the individual CA each for S simulation steps, where V is the number of training vectors and S is equivalent to double the width of the CA. At the end of each simulation run, the success of the run is evaluated based on the majority task (selected for demonstration purposes): All cells of the CA should be in the state prevalent at the initial configuration.

B. *Further Experiments With Majority*

Based on the results of previous experiments (see Section V), the best approach was selected and more detailed results were obtained. The focus of these experiments was to determine best block-size setting policy and to make a more detailed comparison between two accelerators on the same machine (iia and iib).

Different problem sizes (namely 240, 480, 1200, 2400, 4800. 9600 and 24000 training vectors) and block sizes were evaluated (8, 10, 15, 20, 40, 80, 120, 160, 240 and 320 threads per block). Not all the results could be obtained for some tasks because the number of threads per block must be an integer.

C. *The Evolution of Binary Counters*

The other proposed approach mentioned in Section VI was also evaluated. The 4-bit counter design was selected as a benchmark since is not complicated in terms of search space, so the best solution is well known due to experiments with brute force search [17]. The problem has best solution with fitness 10 which means that the CA can approximate only 10 numbers from the sequence.

The initial configuration of 5 was chosen deliberately, because the CA has the best results with this. For example, the seq$^{(0)}$ has best fitness of 8 and seq$^{(2)}$ has best fitness of 9. Best result for seq$^{(5)}$ generates the sequence 5-**2**-7-**4**-9-**14**-11-12-13-**6**-15-0-1-2-**7**-4. The incorrect numbers are in bold [16].

The experiment was designed as follows: The CA was 1D non-uniform 4-cell with r=1 (3-cell neighborhood) and static boundary conditions. Each CA simulation lasted 16 steps.

The evolution of the desired CA was performed with the standard GA. The population was set to 32 individuals, crossover rate of 70% and mutation rate of 18%. The crossover was one-point and the mutation probability is meant for a single gene (i.e. bit). Two-step tournament selection was used. The GA parameters match the experiment conducted in [17] to maintain the best possible quality of solutions and comparability to previous result.

The termination was set to the achievement of the maximum fitness and several hundred runs were conducted with the serial version and several dozen with the parallel version. The machine for the parallel version was the same workstation as in previous experiments (ii), fitted either with GeForce 9600 GT or with GeForce GTX 285. The serial experiments were conducted on several dozen of blade servers comparable with the workstation (Intel Xeon 2.8-3.2 GHz processors with 1 GHz FSB). The goal was to measure the average generation needed to find the best solution (fitness 10). Also, the time was measured for the serial and for the parallel versions.

VIII. Results

The objective of this paper was to evaluate the performance of several parallel algorithms incorporated into an EA. The majority problem was selected only to conveniently pose as the EA’s goal in the first series of experiments and so we are not interested in the quality of evolved solutions. The only relevant information is the achieved speedup compared to serial implementations.

The results for the laptop are shown in Table I while the results for the workstation are shown in Table II. The values in the tables are averages of 10 runs.

Fitness evaluation means the simulation of one CA with one training vector (e.g. 100 individuals with 240 training vectors each and over 10 generations means 24000 fitness evaluations).

TABLE I. RESULTS FOR LAPTOP WITH 8600M GS

Approach	**Fitness evaluations**	**Serial time**	**Parallel time**	**Speedup**
Simulation of CA only	50000	347.39	0.56	621.68
Parallelization of GA				
CA cells	240000	1787.23	2597.60	0.69
Training vectors	240000	1787.74	251.20	**7.12**
Individuals	240000	1784.26	821.04	2.17

TABLE II. RESULTS FOR WORKSTATION WITH GTX 285

Approach	**Fitness evaluations**	**Serial time**	**Parallel time**	**Speedup**
Simulation of CA only	50000	187.33	0.38	489.75
Parallelization of GA				
CA cells	240000	981.34	1597.60	0.61
Training vectors	240000	980.38	72.18	**13.58**
Individuals	240000	980.95	105.21	9.32

The scaling properties of the three proposed algorithms are summarized in Table III. Only the best results from both testing computers are shown. The result for 24000k fitness evaluations with the cell level parallelization is not shown because the computation did not terminate before 10 hours reserved for parallel computation.

TABLE III. SCALING PROPERTIES OF PROPOSED PARALLEL ALGORITHMS

Fitness evaluations	240k	2400k	24000k
Approach	**Speedup**		
CA cells	0.69	0.69	N/A
Training vectors	14.36	127.16	**417.36**
Individuals	9.32	28.41	192.88

After this, several extremely long runs (more than 240k fitness evaluations) were computed with the training vector approach and the best results approached a speedup of 420.

The other series of experiments shows results of comparison between serial version, workstation with 9600 GT (iib) and workstation with GTX 285 (iia). Note that the original contribution [1] included result with GTX 280. In this article the accelerator was replaced with its more modern version GTX 285. The difference between GTX 280 and 285 is in higher core and memory operating frequencies, other characteristics remain the same. More details on the accelerators may be found in [4].

As mentioned in Section V.B, several variations of the task were evaluated. Table IV. shows the best results for different problem sizes (number of training vectors) and different thread-per-block setup. The results are average speedup over several runs with respect to serial run described in Section VII. The best results for each accelerator are highlighted.

TABLE IV. SPEEDUP FOR 9600 GT AND GTX 285

Block size	**Problem size [training vectors]**						
	240	600	1200	2400	4800	9600	24000
	9600 GT						
8	13.95	26.14	22.66	26.72	26.77	28.09	28.39
10	14.14	27.08	33.02	33.28	33.29	35.29	34.99
15	**14.36**	**27.86**	34.30	44.65	52.33	52.51	52.64
20	13.99	27.42	59.38	61.26	61.18	61.42	64.13
40	13.97	27.27	62.54	92.90	97.37	98.59	97.96
80	13.84	27.50	**64.47**	106.06	108.02	110.00	109.64
120	13.65	27.09	60.92	97.18	98.68	118.61	121.03
160	N/A	26.71	N/A	**108.56**	101.22	**120.32**	**126.34**
240	12.69	25.15	61.00	83.97	98.44	123.19	121.02
320	N/A	N/A	N/A	N/A	**109.68**	110.49	111.10
	GTX 285						
8	**13.58**	27.00	64.21	64.00	83.28	97.20	94.61
10	13.57	27.02	65.27	117.72	118.85	122.58	120.79
15	13.58	**27.04**	**66.08**	126.68	124.73	161.32	170.53
20	13.22	26.68	65.27	125.01	215.24	218.03	218.67
40	13.25	26.49	65.45	125.99	222.77	286.31	272.57
80	13.14	26.32	64.82	**127.16**	232.85	350.96	329.73
120	12.96	25.92	64.21	125.87	222.62	351.56	387.20
160	N/A	25.55	N/A	124.10	**236.04**	**370.48**	**417.36**
240	12.08	24.10	59.75	117.43	224.61	302.47	387.37
320	N/A	N/A	N/A	N/A	208.42	371.04	377.95

Figure 7 shows best results for different accelerators and problem sizes only for the optimal block size settings. The results are average execution time obtained from several runs. Figure 7 indicates the difference in performance growing with the problem size.

The last series of experiments measured the speed of evolution designing the solution for the 4-bit counter problem.

The average generation needed to achieve the maximum fitness was 14509 and the average time to achieve it was 5.25 seconds.

With the 9600 GT accelerator (iib), the average generation was 14591 and the time needed 8.5 seconds.

For the GTX 285 accelerator (iia), the average results were 14374 generations and 8.77 seconds.

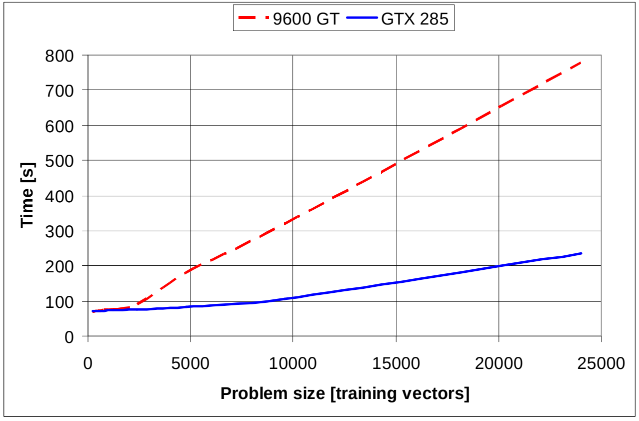

Figure 7. Execution time for different problem sizes and best block size settings for GeForce 9600 GT and GTX 280

It can be seen that no actual improvement of the GA was achieved. However, this was to be anticipated. Moreover, not even a speedup of computation was achieved but on the contrary, the result was slower than serial computation. The reason for this is too small population of the used GA. The overhead required to start the parallel computation on the GPU is larger than the speedup. The result is summarized in Table V.

TABLE V. SPEEDUP OF EVOLUTION WITH POPULATION OF 32

GPU	Avg. generation	Time [s]	Speedup
Serial	14509	5.25	N/A
9600 GT	14591	8.50	0.62
GTX 285	14374	8.77	0.60

In order to prove that accelerating the GA has any sense, another experiment was conducted. This time the population was increased tenfold to 320 individuals. An unsolvable fitness condition was set and the computation was terminated at generation 15000. This had to be done in order to actually measure something, because otherwise the GA would end too soon. This time the experiment proved the conclusion of the previous one: The new problem was large enough, that the parallelization could pay off with a speedup of 38.66 for (iib) and 38.11 for (iia). Results may be found in Table VI.

TABLE VI. SPEEDUP OF EVOLUTION WITH POPULATION OF 320

GPU	Avg. generation	Time [s]	Speedup
Serial	15000	54.12	N/A
9600 GT	15000	1.40	38.66
GTX 285	15000	1.42	38.11

IX. DISCUSSION

A. *Speedup vs. Cost*

As can be seen in Tables I and II, the parallelization of the CA simulation on the level of CA cells shows massive speedup. More surprisingly, those results were obtained with only one multiprocessor in the GPU due to the inter-block synchronization problem mentioned in Section IV. The possible explanation is the effective use of the device shared memory which is much faster than ordinary memory. The results from the laptop GPU are better, because they were compared with much slower processor in the laptop opposed to the hi-end processor in the workstation.

However, when the simulator was inserted into a GA, the speedup declined due to more memory accesses and fitness evaluation. The one-block approach utilizing only one multiprocessor shows its weaknesses and the overall result is even worse than the serial approach.

The results for the parallel GA (the approach from Section V.B) with threads executing individual training vectors are the best of the experiments. The maximum speedup for the workstation is 417.36 and the speedup for the laptop is only 31.34 for the largest problem. As opposed to the cell parallelization approach from Section V.A, this algorithm has to upload large quantities of data to and from the device memory but it has more processor cores and the data don’t conflict with each other.

As seen in Table III, there was no speedup drop with larger problems (2400 and 24000 training vectors) and the performance was even better than for the smaller problem (240 training vectors).

The individual-per-thread approach (Section V.C) showed smaller speedup than training vector-per-thread approach. The lower performance is probably caused by more memory transfers (several sets of CA rules per block opposed to only one set of CA rules per block).

There are also some problems with graphic accelerator cards used as primary display adapters due to graphic driver timeouts caused by long thread execution times so this approach may not be suitable for this reason. This problem may be solved by using second graphics adapter as the primary display adapter at the expense of increased cost in hardware.

The scaling capabilities of the last approach are also good as seen in Table III. The conclusion for the experiments is that parallelization on the level of evaluation of training vectors is the most effective due to utilization of all multiprocessors in the GPU and quick and small parallel kernel (code for the GPU - as opposed to parallelization on the level of individuals where the kernel lasts longer).

The comparison of graphic accelerators showed partly interesting results. A GeForce 9600 GT with 8 multiprocessors showed a nice speedup not falling so far behind the GTX 285 with 30 multiprocessors in some cases. The mid-end card was even faster in some of the smallest tasks. This could be influenced by slightly larger scheduling overhead with the more complex GPU. This opposes the manufacturer’s claim about the zero scheduling overhead [15]. No other explanation seems to fit and since precise details about GPU hardware are obfuscated by the manufacturers, we may only guess the real reason.

Of course, when the problem size increased sufficiently, GTX 285 manifested its higher number of processors which could be used to full capacity and the speedup was significantly higher. The interesting part is, that 9600 GT costs about 80 Euro and the GTX 285 about 370 Euro (Jan 2010, retail price for the Czech Republic). The low-end card shows no promise for scientific computation for two main reasons: It is mounted in a laptop and the memory bus is too slow.

B. *Other Tasks for GPU*

It seems that the mid-end card is the most cost-effective for small and medium-sized tasks. However, the results may be different with other parallelization tasks. It is important to note the two laws for parallel computation. First is the Amdahl's law [12] which states:

$$S(P) = P / (1 + \alpha\ (P - 1)),$$

where S(P) is the speedup, P is the number of processors and α is the sequential part of the task. The equation clearly shows that a fixed task speedup is severely limited by the sequential part, no matter how many processors are used.

The second law is attributed to Gustafson [12]:

$$S(P) = P - \alpha\ (P - 1)$$

and it may be applied to a class of problems, where the non - sequential part of the task is limited by the number of processors and when a new processor is added, we may assign it the same amount of work as to the other processors.

This class of problems includes evolutionary design of CA rules which need to be evaluated by high number of training vectors. More training vectors means more accurate results and more fine-grained fitness function which may contribute to better evolution results.

There is also another class of evolutionary design problems which is limited by the first law. This class includes CA design, which needs to evaluate a small fixed number of steps or possibilities and is not suitable for parallelization. These problems may be also limited by the number of individuals within the GA population, as was the case with the 4-bit counter evolution, where the best population size is 32 [17]. Some of these problems may drift to the area where the task size could be expanded and so Gustafson's law may be applied. Thankfully, many CA design tasks by means of EA fall in this category like our example with the majority benchmark. The 4-bit counter is only a simple demonstration and evolution of larger non-uniform CA will probably need larger populations.

The last conclusion obtained from the results is the fact that block size settings greatly impacts the GPU's performance. Table IV. shows that the optimal settings changes with the size of the problem. The reason for the need to tailor the settings for a certain task is the way the GPU dispatches threads within the blocks.

The GPU programming model uses several levels of data- and thread-parallelism which enable the simultaneous execution of data-independent threads [15]. This concept works in concert with the hardware which offers scheduling consistent with multi-threading concept. This enables the threads within one block to be scheduled as the need arises.

The threads are executed on groups called "warps" which are 16 threads wide and which work in a SIMD concept. That means that each thread within the warp has to execute the same instruction, of course on different data. If the blocks are not large enough, significant parts of the warp may be wasted. Also, the warps are scheduled depending on their availability. If one warp waits for a memory access, several more warps may be executed, even from different blocks. Figure 8 illustrates simultaneous execution of warps on three different multiprocessors.

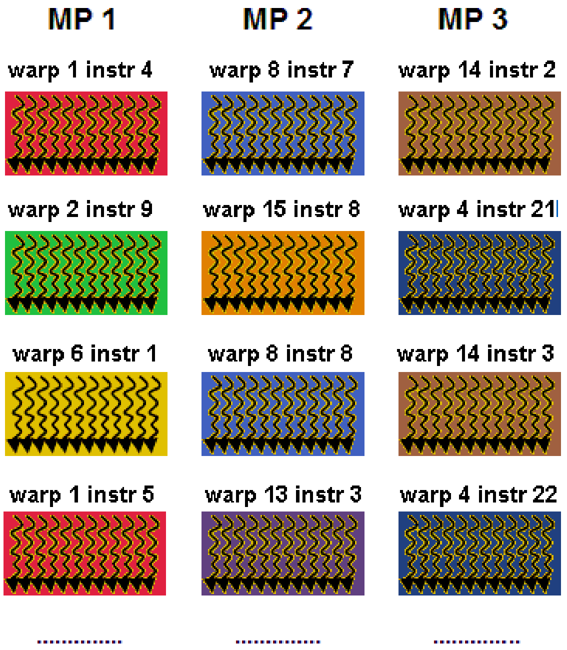

Figure 8. Warps executing on three different multiprocessors (MP): Each warp contains 16 threads and each is executing the same instruction. Warps are scheduled according to readiness for execution.

X. Conclusion

The experiments have shown that evolution on GPUs has several limitations. The most significant one is the fact, that the amount of device shared memory and registers is limited thus restricting the size of evolved CA, the number of training vectors or the size of the EA population.

Dealing with this problem may be the objective of further research and development. Possible solutions include partitioning the computation and serializing the parts in order to save memory. Another solution may be using device global memory instead of shared memory and local memory instead of registers. However, both of these approaches would result in slower performance. Future accelerators may possess larger shared memory [16].

Another possibility for development is testing combinations of the three proposed approaches. E.g. it would be possible to evaluate several individuals of the GA population each in one block while running parallel lookup of cell transition rules in that block.

Further improvement of methods mentioned in this paper could lead into fast parallel version of Sipper's Cellular programming approach [19].

Cellular programming is a methodology developed to design non-uniform CA systems capable of computing complex tasks such as synchronization, majority or sorting. Speeding up the evolution of CA systems may prove to be appropriate step in perfecting such systems via simulation and implementing them in hardware.

The most recent language contribution to the field of GPGPU (General Purpose GPU computing) is OpenCL (Open Computing Language). OpenCL is a C language extension similar in use to CUDA but there is one significant improvement: It supports many different GPU or CPU architectures, being almost universal. OpenCL uses abstractions independent of manufacturers which can be automatically transformed into efficient code for any supported architecture by means of compilation. With Open CL, it is possible to run the same program on nVIDIA and ATI. The language and its tools started to become available only recently, so this article deals only with CUDA. Future plans include transferring current and prospective work to this new language.

Generally the future of GPGPU looks bright. Manufacturers like nVIDIA and ATI compete with each other in GPU performance pushing the development further. Right now more advanced GPU are being planned and developed. Example of this new generation may be nVIDIA Fermi which will include 512 cores (64 MPs) and such technologies as simultaneous kernel execution, shared cache memory for the entire GPU or ECC (error checking and correction) [16].

Moreover with the introduction of OpenCL the programming of competitor's hardware will be unified, further improving the programmer's experience. The architecture is also designed as scalable from the start enabling to connect several GPUs together or to migrate old programs to newer versions just by adjusting the problem size and block settings. More cores and higher operating frequency means more computing power for problems which can be enlarged in accordance with Gustafson's law.

Acknowledgment

This work was partially supported by the grant **Natural Computing on Unconventional Platforms** GP103/10/1517, the FIT grant FIT-10-S-1 and the research plan **Security-Oriented Research in Information Technology**, MSM0021630528.

References

[1] Žaloudek, L., Sekanina, L., Šimek, V.: "GPU Accelerators for Evolvable Cellular Automata", Computation World: Future Computing, Service Computation, Adaptive, Content, Cognitive, Patterns, Athens, GR, IEEE, 2009, pp. 533-537.

[2] Chitty, D.M.: "A data parallel approach to genetic programming using programmable graphics hardware", GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation, Volume 2., London, ACM Press, 2007, pp. 1566–1573.

[3] Codd, E., Cellular Automata, Academic Press, 1968.

[4] CUDA-Enabled GPU products - NVIDIA
URL: <http://www.nvidia.com/object/cuda_learn_products.html>
[cit. 29.1.2010]

[5] Durbeck, L. and Macias, N., "The Cell Matrix: An Architecture for Nanocomputing", Nanotechnology 12, IOP Publishing 2001, pp. 217–230.

[6] Fok, K.L., Wong, T.T., Wong, M.L.: "Evolutionary computing on consumer graphics hardware", IEEE Intelligent Systems, Vol. 22, No. 2, IEEE, 2007, pp. 69–78.

[7] Gardner, M., "Mathematical games: The fantastic combinations of John Conway's new solitaire game 'Life'", Scientific American 223, Oct 1970, pp. 120-123.

[8] Gobron, S., Devillard F., Heit B., "Retina simulation using cellular automaton and GPU programming", Machine Vision and Applications Journal 66, Springer, 2007, pp. 331–342.

[9] Harding, S.: "Evolution of Image Filters on Graphics Processor Units Using Cartesian Genetic Programming", 2008 IEEE World Congress on Computational Intelligence, Hong Kong: IEEE CIS, 2008, pp. 1921–1928.

[10] Harding, S. and Banzhaf, W.: "Fast genetic programming on GPUs", Proceedings of the 10th European Conference on Genetic Programming, LNCS 4445, Springer, 2007, pp. 90–101.

[11] Harris, M.: "Mapping computational concepts to GPUs", ACM SIGGRAPH 2005, ACM, New York, NY, 2005.

[12] Henessy, J. and Patterson, D.:Computer Architecture A Quantitative Approach, The Morgan Kaufmann Series in Computer Architecture and Design, Morgan Kaufmann Publishers, 2003.

[13] Langton, C.G., "Self-Reproduction in Cellular Automata", Physica D: Nonlinear Phenomena 10(1-2), Elsevier, 1984, pp. 135-144.

[14] Lohn, J.D., and Reggia, J.A., "Automatic discovery of self-replicating structures in cellular automata", IEEE Transactions on Evolutionary Computation, vol.1, no. 3, IEEE CS, 1997, pp. 165-18.

[15] nVIDA CUDA Programming Guide, Version 3.0
URL: <http://developer.nvidia.com/object/
cuda_3_0_downloads.html.>
[cit. 14.6.2010]

[16] Next Generation CUDA Architecture, Code Named Fermi
URL: <http://www.nvidia.com/object/fermi_architecture.html>
[cit. 31.1.2010]

[17] Sekanina, L.: Evolvable Components: From Theory to Hardware Implementations, Natural Computing Series, Springer-Verlag, Berlin Heidelberg, DE, 2004.

[18] Šimek, V., Dvořák, R., Zbořil, F., V., Kunovský, J. "Towards Accelerated Computation of Atmospheric Equations using CUDA", Proceedings of Eleventh International Conference on Computer Modelling and Simulation, Cambridge, GB, IEEE CS, 2009, pp. 449-454.

[19] Sipper, M., Evolution of Parallel Cellular Machines: The Cellular Programming Approach, Springer Verlag, Heidelberg, 1997.

[20] Tomassini, M., Sipper, M., Perrenoud, M., "On the generation of high-quality random numbers by two-dimensional cellular automata," Computers, IEEE Transactions on , vol.49, no.10, Oct 2000, pp.1146-1151.

[21] Wolfram, S.: A New Kind of Science, Wolfram Media Inc., Champaign, IL, 2002.

[22] Yao, X. and Higuchi, T., "Promises and Challenges of Evolvable Hardware", IEEE Transactions on Systems, Man, and Cybernetics 29(1), IEEE, 1999, pp. 87–97.