Evolution of digital circuits using CUDA to determine the fitness function in genetic algorithm

Wilian Soares Lacerda
Department of Computer Science
Federal University of Lavras
P.O.Box 3037, Lavras, MG
Brazil, CEP 37200-000
Email: lacerda@dcc.ufla.br

Luiz Henrique Rowan Peixoto
Systems Eng. and Inf. Technology Institute
Federal University of Itajubá
P.O.Box 50, Itajub, MG
Brazil, CEP 37500-903
Email: henriquerowan@yahoo.com.br

Thomaz Chaves de A. Oliveira
Department of Computer Science
Federal University of Lavras
P.O.Box 3037, Lavras, MG
Brazil, CEP 37.200-000
Email: thomazchaves@dcc.ufla.br

***Abstract*—This paper proposes the use of Evolutionary Computing applied to the synthesis of digital circuits in the disjunctive normal form. Due to the computational cost of the employed technique, the platform of digital synthesis was implemented using parallel processing in order to reduce processing time. Performance comparisons were accomplished, proving that the digital platform using CUDA had considerable overall gain performance. A case study of digital synthesis was accomplished with simple digital comparators circuits. The presented result in this paper demonstrates the feasibility of the synthesis of combinational digital circuits by Evolutionary Computation, which represents a new design methodology for electronic circuits.**

I. Introduction

In recent years a new possibility has surged by the application of biological concepts to technological development: the conception of evolution capable devices, defined as Darwinian machines, more commonly known as Evolutionary Hardware (EHW) [1]. The implementation of such devices involves concepts and techniques of Evolutionary Computing, Genetic Algorithms and reconfigurable electronic circuits, leading towards the construction of autonomous, self-adaptive and fault tolerant systems [2] [3].

Genetic Algorithms (GAs) are applied to complex problems with high computational costs, such as the one of Evolutionary Hardware. Due to this, the concept of parallel genetic algorithms is widespread, that typically reduces the processing time that yields to a satisfactory result in comparison to an equivalent sequential process. These are accomplished by multicore processing, computer clusters and more recently also by general-purpose graphics cards also known as GPGPU (General-Purpose Computing on Graphics Processing Units).

The main objective of this work is to investigate methods for synthesizing digital electronic circuits using hardware evolution techniques. This was implemented in the form of a simulation on a parallel and distributed high performance processing platform using multicore processing implemented in a GPGPU. Throughout the use of a truth table as input, the evolutionary algorithm must encounter a digital circuit that all its entries match the input truth table and also has the smallest possible size (least number of circuit elements).

II. Evolutionary Electronic

In 1991 Louis [4] introduced the concept of evolutionary algorithms as a tool for designing digital circuits, where the concept was used to interconnect digital ports for solving a specific problem, such as the parity function. DeGaris [1], in 1993, introduced the concept of Evolutionary Hardware (EHW). In 1996 Higuchi et al. [5], evolved digital circuits for pattern recognition. In the same year, Thompson [6] intrinsically evolved a circuit in an FPGA system. These are some of the milestones of this new research area, which has since then gained interest of various researchers.

Degaris [1] defined some classifying categories within this research area regarding the following design properties: project type, design and nature of evolutionary platform. Regarding the nature of the project, the applications can either be classified as analogue or digital; the evolutionary platform can be either classified as extrinsic and intrinsic. Extrinsic applications are those in which the circuits are evaluated throughout circuit simulating environments such as SPICE [7], [8], on the other hand, in intrinsic applications, the characteristics of a circuit can be modified in real time in order to improve its performance in some or several requirements [9]. This technique is normally accomplished by by the use of programmable integrated circuits [10]. Finally, project type categories are divided into optimization and synthesis of circuits.

The design approach to EHW can be either classified as direct or indirect according to the chromosome representation level. The direct approach to EHW encodes bit circuit architecture as chromosomes, which specifies the connectivity and circuit element functions (usually the port-level) circuits. In contrast, the indirect approach does not directly involve bits circuit architecture. It uses a high-level representation, such as trees or grammar as chromosomes representatives. These trees or grammars are then used to generate circuits [11].

A. Evolution of Digital Circuits

The first research using evolutionary algorithms for the synthesis of digital circuits was proposed by Louis in 1991 [4]. There are some important reasons that demonstrate the feasibility of such applications: digital circuits have a higher fidelity simulations than those of analogue circuits and they are more widely used in complex systems [12]. The evolution of digital circuits as reconfigurable FPGAs allowed the first intrinsic

experiment in 1996 by Thompson [6], after that, several researchers performed experiments of intrinsic syntheses as Koza [13][14], Zebulum [12], Stoica , Higuchi [5], Yao [11], Zhang et al. [10], among others. A survey on evolutionary hardware design for electrical and mechanical systems can be found on [15].

Digital systems can be classified into four levels of abstraction: 1 - level of circuit elements, such as transistors and resistors; 2 - level of logic gates; 3 - Boolean equations using, for example, conjunctive normal form or sum of products; 4 - architectural level using ALU's, multiplexers, and other memories.

Hardware representation by Boolean expressions is widely used in this type of project since this brings a high level of abstraction, which is of interest in the context of Evolutionary Computation. Furthermore, this representation fits within the context of combinational circuits, i.e. circuits that the output depend only on binary input voltages.

The Figure 1 shows a hypothetical function translated into a ternary vector, where each gene is represented by a mini-term expression. It is necessary to know the number of literals (inputs) of the circuit. In this representation 0 is a neglected literal, 1 is a literal with logic level true value and 2 is an absent literal.

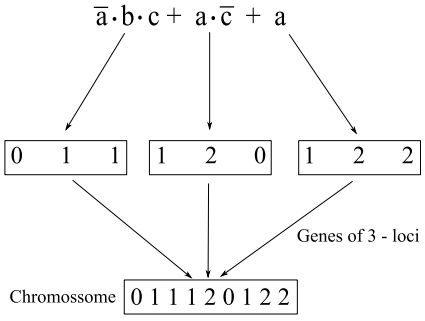

Fig. 1. Level representation of Boolean functions (Adapted from [12])

Over the last few years many researchers have created evolving digital circuits applications as Zebulum [12] brought new methodologies for the design and optimization of layout in VLSI chips; Nedja and Mourelle [16] synthesized a digital circuit for a RSA-Based Cryptosystem; Higuchi [17] used Evolutionary Electronics for an artificial hand to be attached to a human arm.

III. Parallel Programming Platforms

The parallel and distributed computing seeks to achieve high performance in time by optimizing the processing capacity of available machines, that is, exploiting in efficient manner the parallelism in the developed algorithms. There are different middleware that allow both parallel and distributed programming applications. These techniques can be combined to better exploit the levels of parallelism in an application according to the available architecture. For the development of parallel and distributed applications there are middlewares that support sequential programming languages. Among them: MPI [18], OpenMP [19], RMI [20] , CORBA [20] , CUDA [21], and others. In this section we present some characteristics of the MPI and CUDA platforms which were used to obtain high performance in time through parallelism in this work.

A. Message Parsing interface (MPI)

MPI is a standard widely used by the scientific community for data communication in parallel computing. It is popular because it provides a platform for writing message passing programs with a practical, portable and efficient methodology. It is ideal for applications where it is necessary to obtain high performance for both multicore processing and clusters of processors. According to Pacheco [18] and Quinn [19], MPI is widely used to accomplish the exchange of messages among threads of a parallel applications developed for a distributed environment. This standard has been implemented for C, C++, Fortran and Java (underdeveloped version). With the use of MPI is possible to make a natural and elegant disjunction of a problem, with its main characteristics being portability and efficiency. By its features, the MPI standard has also a wide acceptance in the industry [22].

B. Compute Unified Device Architecture (CUDA)

CUDA is a parallel computing architecture of NVIDIA Corporation using programming languages such as CUDA C, OpenCL, DirectCompute and CUDA Fortran aiming to exploit parallel processing of NVIDIA GPU's. Its main advantage is that is a multithreaded general purpose architecture that has gained interest of the scientific community [23]. For users familiar with the C programming language, the CUDA parallel programming framework was developed in order to offer fast learning for new users [24].

The framework's three main abstractions are: group hierarchy threads, shared memory and synchronization. Based on these abstractions, the programmer decomposes the problem into independent processing blocks that are either divided into processing segments or threads. Furthermore, this problem's data must be stored in a hierarchically organized memory.

CUDA's memory hierarchy is divided into three different levels as described below:

- **global memory:** manipulated by all instantiated blocks and threads. It has the largest amount of memory available among all levels and its access is very slow compared to the shared memory.
- **local memory:** manipulated only by a given thread, with serious size restrictions. The term "local" does not indicate that it has a fast access, due to the fact that both the local and the global memory are located beyond the processing core.
- **shared memory:** it is shared and manipulated by all threads instantiated by in a given block. It is the fastest manipulation level after the registers, because it is located within the processing chip.

All levels of memory have their disadvantages. It is necessary to emphasize that both the global and the local memory can be allocated at runtime by the CPU and they are slower in access time. Shared memory can not be allocated at runtime, as it is defined with a static size. It has superior access performance compared to the to global and local memories.

In most cases the performance is increased when data is transferred from the global or local memory to the shared memory [25].

The CUDA programming framework suppresses the complexity of NVIDIA GPU's. As programmers do not write code directly to the hardware, they use the predefined functions in the CUDA API. Another advantage of this is that even if the NVIDIA makes changes to the hardware architecture, older codes will continue to function, as they were written using the API.

```
// Kernell definition xxxx

__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] +B[i];
}

int main()
{
     // Kernel invocation with N threads
     VecAdd <<< 1, N >>> (A, B, C);
}
```

Within the CUDA programming framework is interesting to highlight a set of function markers that defines which codes run at the CPU or at the GPU:

- $\_\_global\_\_$: Defines the function as a kernel, i.e., it will be invoked by the CPU and will be physically executed by the GPU. In the code highlighted above, the CPU code makes a call to a kernel with one block and N threads.
- $\_\_device\_\_$: This marker defines that the function will be compiled for the GPU and will be invoked by the GPU.
- $\_\_host\_\_$: This marker defines that the function will be compiled for the CPU and will be invoked by the CPU. According to GPU developer NVIDIA [24] this marker can be used with the $\_\_device\_\_$ and indicates that the same code can be executed by both the GPU and CPU.

The CUDA programming framework assumes that both the GPU and the CPU have separate memories [24]. In this context, the GPU is referred as "device" and the CPU is referred to as "host". Basically every program involving GPU processing has the following steps:

1) The host allocates memory in the device;
2) The host transfers data to the device into allocated memory;
3) The host invokes the execution of kernels by passing pointers as parameter to the memory's device;
4) Results are transferred from the device back to the host;
5) The memory is deallocated in the device.

In this processing methodology, the occupation of processing cores is surpassed. Processing blocks are handled by the CUDA scheduler, which handles load distribution among cores in order to get the best possible performance (Figure 2).

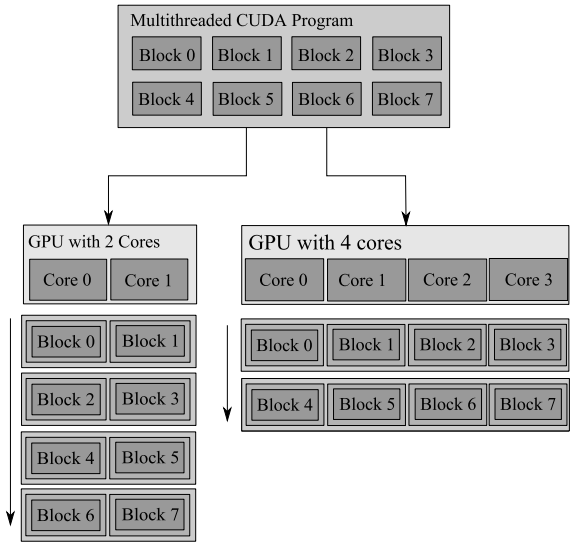

Fig. 2. Escalation of blocks in the cores (Adapted from [24]).

With the increased acquisition of this technology by researchers, the number of applications is ever increasing, with the following examples: videos [26] and images processing [27] [28], computational biology and chemistry, fluid dynamics simulation, image reconstruction in computed tomography, seismic analysis, ray tracing, and others [24].

IV. Genetic Algorithms in Parallel Programming

The Genetic Algorithm (GA) parallelism is presented in the literature considering both single and multiple populations. The main implementations of parallel genetic algorithms that are presented by Cantupaz [29] are:

- global single-population master-slave GAs;
- fine-grained single population GAs [30];
- multi-population coarse-grained GA;
- hybrid GAs.

The combination of different models of parallel genetic algorithms, associated with the presented computational architecture has the potential to produce good quality solution results and with high computational performance.

In the GAs literature there are studies that explore different techniques for parallel and distributed programming in order to achieve good overall performance. Zhang et al. [31] proposed a parallel genetic algorithm implementation coded in C and a MPI master-slave model for the optimization of parameters in a flexible multi-body model.

Berger and Barkaoui [32] also proposed a parallel implementation for a hybrid genetic algorithm applied to the vehicle routing problem with time windows, where two populations evolve by competing where each uses different strategies.

A review of the literature involving parallel computation and evolutionary algorithms is presented by Alba and Tomassini [33].

V. SYNTHESIS OF DIGITAL CIRCUITS

In this section, the evolutionary synthesis methodologies employed to the synthesis of digital circuits at a functional level are explained in detail. Furthermore, CUDA parallel processing techniques are also approached in order to reduce time in the synthesis process.

A. Methodology Representation in Digital Electronics

One of the factors that most influences the performance of an evolutionary algorithm is the adopted representation for both genes and individuals due to the fact that these structures encode probable solutions. This work investigates the application of such methodologies in logic level functions at the disjunctive normal form, also known as sum of products.

1) Level Representation of Functions: Among the logical representations, the Boolean function corresponds to the highest level of abstraction among chromosome and the circuit itself, since the circuit can be represented by an algebraic Boolean expression where the output can be either 1 or 0. In the case of a combinational function of N inputs, each gene must have N internal variables (or literals) and each of these variables can assume three possible values:

- 0 - indicates that the entry in this variable must be denied;
- 1 - indicates that the entry in this variable should not be denied;
- 2 - indicates that the entry in this variable is absent, therefore disregarded from the expression.

In the disjunctive normal form presented in this work, disjunctions from conjunctions of literals occur. The disjunction is equivalent to an OR gate and the conjunction is equivalent to an AND Gate. An example of a Boolean function in disjunctive normal form with its corresponding equivalent digital circuit is presented in Figure 3.

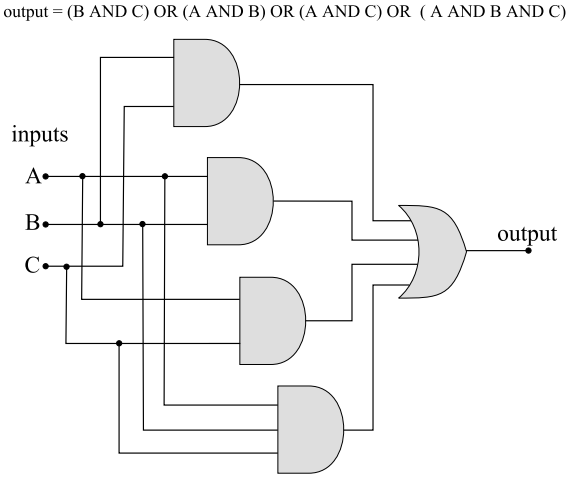

Fig. 3. Boolean expression and corresponding digital circuit

2) Genes: The data structure representation of digital circuits should facilitate genotype-phenotype mapping. Throughout the use of the ternary representation previously described and modelled in the disjunctive normal form, each gene must map a literal conjunction, i.e. each literal represented in a gene receives an input value (e.g. a truth table). Depending on its state (0, 1 or 2), its value will be altered and an additional AND operation will occur with the other literals of this gene (Figure 1).

The above described gene mapping structure is known in the literature as minterm and has this designation only in the sum of products representation.

3) Individuals: An individual in the functional representation must be capable of decoding its chromosome to a complete Boolean expression that can be a possible solution of the input truth table. This chromosome, or genes vector, does the mapping of disjunctions among minterms, result of which is a binary number that may be equal to truth table's output. The quality of an individual in this problem depends directly to the quantity of hits that it presents for a determined input truth table. This quality is also dependent to the number of minterms on its chromosome. In this work, when a solution's fitness is the closets to zero, it is considered the best solution within a population, i.e. the evolution occurs only when there is a minimization of an individual's fitness.

The fitness function proposed in this paper is represented by Equation 1 for integer values.

$$Fitness = s + (p \cdot e) \tag{1}$$

where, e is the error for an individual, s is the chromosome size and p is the penalties applied to errors of individuals.

In order to make the fitness calculation easier, the size of a chromosome always has weight 1. Penalties are also applied to errors that individuals present in the rows of truth table. It can be noticed that in the minimization process of a Boolean circuit given a determined truth table, errors can not be tolerated in the evolutionary process.

4) Hierarchically Structured Population : The hierarchically structured population of this work utilized a tree array data structure, where each node is an individual. The root node, also known as a leader node, of a sub-tree is always a better solution than those of the branched nodes. Individuals in the branches are known as followers. Figure 4 represents a population in a ternary tree data structure.

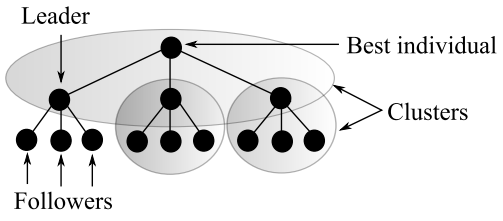

Fig. 4. Population in ternary tree (adapted from Toledo [34]).

B. Evolutionary Methodologies in Digital Electronic

This subsection presents the multi-population genetic algorithm with the populations structured in hierarchical trees, as proposed by Toledo et al. [35]. The crossover and mutation processes for this type of application are also described.

1) Multi-populational Genetic Algorithm: The operation of the multi-population genetic algorithm used in this study is described by the following pseudo-code by Toledo et al. [35].

```
Method MultiPopulationGeneticAlg
begin
   repeat
      for i = 1 to numberOfPopulatoins do
        initializePopulation(pop(i));
        evaluatePopulationFitness(pop(i));
        structurePopulation(pop(i));
        repeat
           for j = 1 to numberOfCrossover do
              selectcParents(indivualA, individualB);
              newInd = mutation(newInd);
              if(executeMutation newInd)
              then
                 newInd = mutation (newInd);
              evaluateFitnessIndividual(newInd);
              insertPopulation(newInd, pop(i));
           end
        structurePopulation(pop(i));
       until(populationCovergence pop(i));
     end
     for i =1 to numberOfPopulations do
         executeMigration pop(i);
     end
    until (stop criterion)
end
```

The first loop described in the pseudo-code is responsible for initializing the population. The function `initializePopulation()` generates populations with random individuals acquired from external data. This external data's objective is to establish constrains within the search domain, and is configured by two files:

- truth table of the circuit to be synthesized;
- configuration file that contains data for: population's structures, types of crossover and mutation, runtime platform, etc.

The function `evaluatePopulation()` is responsible for evaluating the individuals of each generation. Function `structurePopulation()` organizes individuals in a hierarchical tree, as explained in subsection V-A4.

Function `selectParents (indA, INDB)`, which is the next step of the algorithm, tends to choose the fittest individuals in order to get the best combinations among them, but it is not guaranteed that the crossing of the best individuals generate good descendants. It is also worth noting that nothing prevents good descendants to be generated by the crossing of individuals with low fitness. The first step is to drawn a cluster or sub-tree within the population tree, afterwards, the root node and one of its followers that is randomly chosen form the crossing pair.

With the recombination performed by function `crossover(Inda, INDB)`, the new generated individual may go through the process of mutation in function `mutation(newInd)`, according with a criterion explained below. Finally the new individual's fitness is evaluated by `evaluateFitnessIndividual(newInd)` and it is inserted in the population by function `insertPopulation(newInd, pop(i))`. If its fitness is better than one of its parents, it will replace the parent with the worst fitness that participated in the crossing process.

The operations of selection, recombination and mutation described above are repeatedly performed in a loop defined by a parameter n_c coded as `numberOfCrossover`, which is calculated by Equation 2.

$$n_c = p_s \cdot c_r \tag{2}$$

where n_c is the number of crossovers, indicating the number of generated individuals, p_s is the size of the population concerned and c_r is the crossover rate, which is a recombination rate calculated for this population.

In this work, each population is structured in a ternary tree with 13 individuals. The adopted recombination rate value was 10, consequently 130 individuals are generated. After this step, the population is internally ordered by function `structurePopulation()` so that population's structure obeys the hierarchy within the tree.

The processes described above repeats until a specified population has converged. This occurs when no new individuals are inserted in the population, as a consequence the next population will be processed.

The last phase of the algorithm consists of a migration procedure. In this process the best individuals are migrated to ring-shaped adjacent populations, as exemplified by the arrows of Fig. 5, where the migration among populations organized in three ternary trees can be observed.

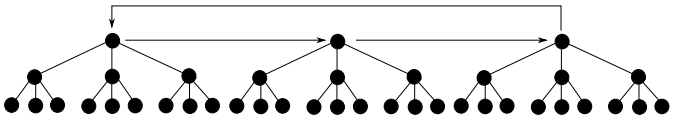

Fig. 5. Example of migration among three populations (adapted from [34]).

The stopping criterion for the the multi-population genetic algorithm is defined by a time execution limit.

2) Crossover and Mutation: The problem concerning crossover of chromosomes with different sizes is to establish a criteria in relation to a new individual's chromosome size. This work, adopted a procedure known as uniform crossover, its criterion for the child's chromosome size is a random number between the smaller and the larger size for both parent's chromosome size. This procedure is described by the following pseudo-code for the Uniform Crossover Algorithm.

```
Method uniformCrossover(p1, p2, child)
begin
   major = max(p1.size, p2.size);
   j=0;
   while(j < major) begin
      if ((rand()%2 == 0) and (p1.size > 1)
      then
         child.chromo.push_back(p1.chromo[j]);
      else if (p2.size > j);
      then
         child.chromo.push_back(p2.chromo[j]);
      j++;
   end
end
```

In this code, the variable `major` stores largest parent chromosome size involved in the crossing. After this, the process of copying genes from parents to descendant begins, where a draw occurs (`rand()%2`) with equal probability among parents. This will indicate which parent will contribute with its gene. The parent can only pass a gene to its descendant if it is drawn and if the gene is at position `j`.

In this algorithm neither the crossover generated individuals rate or mutation generated individuals rate were used. Thus, every individual is generated by crossover and it may pass throughout

the mutation operator. What defines this is is a parameter called probability of mutation.

If the individual is selected for mutation, a routine initiates for each one of the individual's gene. Each gene will have a probability equal to 50% of passing throughout only one of the three types of mutation:

- Removes the current gene;
- Adds a random gene to a chromosome end;
- Alters the value of only one literal chosen randomly within the minterm (gene), the current value of this literal will be replaced by a random value.

C. Implementation Using CUDA

The highest computational cost task involved in the evolution of the digital circuits is the task of calculating fitness. In this step it is necessary to check all outputs of the truth table for a particular individual, where the number lines with errors for and individual (e) is encountered and applied to Equation 1.

To reduce communication overhead of host and device, all individuals generated by crossover and mutation are inserted in a vector described by Equation 2. This vector is passed to the method of evaluation of individuals in the GPU by the Pseudo-code evaluation of individuals presented bellow.

```
Method evaluateFitness(individuals)
begin
    allocateMemoryGpu(chromos, individuals.bytes);
    allocateMemoryGpu(errors, individuals.size);
    transferMemory(chromos, individuals.chromos);
    for i = 0 to individuals.size do
       KernelErrors <<< linesOfTable, entriesTable
       >>>(chromos[i], individuals[i].tam_chromo,
       errors[i],table, outputs);

    transferMemory (individuals.errors, errors);
    deallocateMemoryGpu (chromos);
    deallocateMemoryGpu (errors);
    penaltiesApply (individuals, individuals.errors);
end
```

At first, this method allocates memory for chromosomes and the error vector in the GPU. After this, it transfers the chromosomes to the device and makes reference to `kernel`. The next step is to transfer the error vector for the host and finally the method deallocates the memory used in the device and calculates the individual's fitness using Equation 1.

To implement the error calculation in GPU code, it was necessary to diagnose the highest possible level of parallelism that could be applied for this problem. The encountered solution was to calculate the error of one row, as this calculus is independent among rows of the truth table. By the adoption of such solution each processing core may be responsible for evaluating one truth table row of every individual. Throughout the acknowledgement the CUDA programming paradigm that the blocks are scaled to be processed in the cores, illustrated by Fig. 2, each block will be responsible for the evaluation of only one row exclusively; i.e. the quantity of blocks that perform the `kernel` is always the number of rows of the truth table.

1) Parallel Fitness Calculation Algorithm: The error calculation method was implemented to be performed in the GPU and its pseudo-code is presented bellow.

```
Method operation(a,b)
begin
   return (^a and b) or (a and ^b);
end

Method kernelErrors(chromo, tam_chromo,
errors, table, outputs)
begin
   _shared_aND;
   _shared_oR;
   _local_bid = blockIdx.x;
   _local_tid = threadIdx.x;
   _local_var_thread = table[bid][tid];

   if (bid==0) and (tid == 0) then
      errors = 0;

   oR = 0;
   for i = 0 to tam_chromo do
       aND = 1;
       synchronizeThreads();
       if (operation(var_thread, chromo[i][tid])== 1)
       then
           aND = 0;
       synchronizeThreads();
       if (aND == 1)
       then
          oR = 1;
          break;
       end
   end

   if (tid == 0)
   then
      if (oR <> outputs[bid])
      then
          atomicAdd (errors, 1);
end
```

To understand the working principle of this algorithm is necessary to acknowledge that:

- The number of blocks used is always the number of rows in the table and these are actually instantiated in one dimension;
- The amount of threads involved in the calculation of errors for each block is always equal to the number of entries in truth table. These are also instantiated in one dimension;
- Variables identified as `_shared_` are shared among all threads of a block, as all blocks have access to this variable;
- Variables identified as `_local_` are manipulated only by a single thread, as all threads in all blocks have this variable;
- Parameter inserted variables are GPU global memory blocks, where all threads of all blocks have access;
- The `sysncronizeThreads()` function synchronizes all threads of a block. It is not possible to synchronize threads from different blocks using this function;
- The function `atomicAdd()` is a CUDA library function that allows that only one thread can handle one variable at a time, excluding the possibility that several threads have access to the same critical region at the same time.

In the Parallel Fitness Calculation Algorithm, at first, each thread identifies which block it acts through the call of `blockIdx.x` and each thread identifies its position within the block by the calling of `threadIdx.x`. With every thread identified, only the one that contains identifiers `(0.0)` initializes the global variable `errors` with `0` value.

The variable `aND` stores an individual's minterm result in the truth table's row where the block is processing on, where the default output value for a minterm is `1`. The variable `oR` stores an individual's final result in the same row, where an individual's default output value for a particular truth table row is `0`.

After initialization of variables, the threads run the function `operation(var_thread, chrome [i][tid])` synchronized and in parallel, where the objective of this function is to detect if the minterm's output is 0. This function only returns value 1 if:

- The truth table's input is 0 and the minterm's variable has value 1;
- The truth table's input is 1 and the minterm's variable has value 0.

After the thread's synchronization, each thread evaluates if the minterm's output is 1 (`if (aND == 1)`) and they assign 1 to the individual's output (`oR=1;`). The individual's evaluation within a block is terminated by the `break` command. In the case of a minterm's output does not result in 1, the evaluation of next minterm in the loop begins.

When the evaluations in a block are finished, only `thread 0` of each block can verify if an individual committed an error for a row of the truth table, if so, it is accounted by the calling of function `atomicAdd(errors,1)`.

VI. Results and Discussion

This section presents the performance results of sequential, multi-threading, and sequential with GPU implementations of digital circuit synthesis.

For the tests a computer with the following characteristics was used:

- Intel Core™ 2 Duo E8400 @ 3GHz, with 4GB of RAM;
- NVIDIA GeForce GTX 285 with 240 cores of 1.47GHz.

Three algorithm versions were implemented, described in section V-B1 called **Seq_CPU**, **Mult_CPU** and **Seq_c_GPU**, where:

- **Seq_CPU** is a sequential implementation of the algorithm running by the CPU only.
- **Mult_CPU** is a parallelized implementation with multi-threading, where the evolution of each population is performed on a different thread, and only the migration step described in Figure 5 is accomplished in a sequential manner. This implementation is entirely executed by the CPU.
- **Seq_c_GPU** is a sequential implementation of the algorithm for CPU, where the error calculation is accomplished by parallel computing by the GPU, as described in Section V-C.

For the tests five truth table instances were used from comparators in digital form $A > B$, where A is represented in the first half of the inputs and B in the second half. When A is greater than B the logic output should be 1. These instances have 4, 6, 8, 10 and 12 entries respectively being called by the names of Comp_4, Comp_6, Comp_8, Comp_10 and Comp_12.

The performance results are described in the Mean column of Table I, and it indicates the number of individuals that were generated and evaluated during the execution period described by the Time column. Ten executions were accomplished for each instance in every implementation. From these ten runs the mean value was extracted. The execution time is extented from one instance to another according to the complexity of each instance; e. g., from one instance to another the execution time is multiplied by four.

TABLE I. Performance of the implementations **Seq_CPU**, **Mult_CPU** and **Seq_c_GPU**.

Config.	Instance	Table rows	Time (s)	Mean[1]
Seq_CPU	Comp_4	16	4	1.53E+06
Mult_CPU	Comp_4	16	4	2.16E+06
Seq_c_GPU	Comp_4	16	4	3.71E+05
Seq_CPU	Comp_6	64	16	1.09E+06
Mult_CPU	Comp_6	64	16	2.04E+06
Seq_c_GPU	Comp_6	64	16	9.08E+05
Seq_CPU	Comp_8	256	64	1.18E+05
Mult_CPU	Comp_8	256	64	2.19E+05
Seq_c_GPU	Comp_8	256	64	2.29E+05
Seq_CPU	Comp_10	1024	256	2.98E+04
Mult_CPU	Comp_10	1024	256	5.81E+04
Seq_c_GPU	Comp_10	1024	256	8.55E+04
Seq_CPU	Comp_12	4096	1024	7.83E+03
Mult_CPU	Comp_12	4096	1024	1.43E+04
Seq_c_GPU	Comp_12	4096	1024	2.32E+04

The results in Table I indicate that the **Mult_CPU** implementation has performance gain that reaches 95% higher than of **Seq_CPU** in the Comp_10 instance, since the utilized processor has only two cores and this instance is the one that most benefits from parallelism.

Mult_CPU implementation had a 5.8 times better performance than the **Seq_c_GPU** implementation and **Seq_CPU** implementation 's performance was 4.1 times better than **Seq_c_GPU** implementation, both running the same instance Comp_4. This lower **Mult_CPU** result was caused by communication overhead, which in this case is large compared to the GPU processing time. Another important factor is that using a table with only 16 rows, the maximum number of processing cores used by the GPU is 16, as explained in subsection V-C1.

From the instance Comp_8 with 256 rows, using GPU became feasible as all the cores could be used in parallel, as the implementation **Seq_c_GPU** had a 5%, 47% e 62% superior performance than the **Mult_CPU** implementation in the Comp_8, Comp_10 e Comp_12 instances respectively. Beyond that, the Seq_c_GPU implementation had a superior performance over the **Seq_CPU** implementation of 94% , 187% and 197% in same respective instances.

These benchmark results highlight that the use of GPU is feasible when its possible to use all its processing cores and also when the ratio of communication overhead over processing on GPU is very low.

VII. Conclusion

This work presented a methodology for the synthesis of digital electronic circuits in disjunctive normal form. It was automated using a multi-population genetic algorithm. For this type of application, a truth table must be used as input as well as a set of parameters for the evolutive platform.

Increasing the number of entries in the truth table causes an exponential increase of computational cost for synthesis. To minimize the runtime platform this work investigated the use of parallel programming on a GPU with CUDA technology, where it was possible to increase performance by up to 197% compared to the sequential processing. However the use of GPU is feasible only when you can use all your processing power and also when the ratio of communication overhead over processing on GPU is very low.

The results presented in this paper demonstrate the feasibility of the synthesis of combinational digital circuits by Evolutionary

[1] Average individuals who passed through the implementation of the evolutionary process.

Computation, but with the exponential growth of the problem it may required other techniques to be aggregated to the methodology.

ACKNOWLEDGMENT

The authors would like to thank the suport of FAPEMIG and CNPq.

REFERENCES

[1] H. De Garis, "Evolvable Hardware: Genetic programming of a Darwin machine," in *International Conference on Artificial Neural Networks and Genetic Algorithms*, Innsbruck, Austria, 1993, pp. 441–449.

[2] J. Mizoguchi, H. Hemmi, and K. Shimohara, "Production genetic algorithms for automated hardware design through an evolutionary process," in *IEEE World Congress on Computational Intelligence*, Orlando-FL, USA, 1994, pp. 661–664.

[3] L. Sekaninaa, "Evolutionary hardware design," in *Proceedings of SPIE, VLSI Circuits and Systems*, Prague, Czech Republic, 2011, pp. 1–11.

[4] S. J. Louis and G. J. E. Rawlins, "Designer genetic algorithms: Genetic algorithms in structure design," in *Proceedings of the Fourth International Conference on Genetic Algorithms*, R. Belew and L. Booker, Eds., 1991, pp. 53–60.

[5] T. Higuchi, M. Iwata, I. Kajitani, H. Iba, T. Furuya, and B. Manderick, "Evolvable hardware and its applications to pattern recognition and fault-tolerant systems," in *Toward Evolvable Hardware: The Evolutionary Engineering Approach*, E. Sanchez and M. Tomassini, Eds., vol. 1062. Berlin, Germany: Springer-Verlag, 1996, pp. 118–135.

[6] A. Thompson, "Silicon evolution," in *Stanford University*. MIT Press, 1996, pp. 444–452.

[7] L. W. Nagel, "Spice2: A computer program to simulate semiconductor circuits," EECS Department, University of California, Berkeley, Ph. D. Thesis, 1975. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/1975/9602.html

[8] A. Vladimirescu, *The SPICE Book*, first edition ed. New York-NY, U.S.A.: John Wiley and Sons, Inc., 1994.

[9] F. A. Salazar and A. Mesquita, "Synthesis of analog circuits using evolutionary hardware," in *Proceedings of Sixth Brazilian Symposium on Neural Networks*, Rio de Janeiro - RJ, Brazil, 2000, pp. 101–106.

[10] Y. Zhang, S. L. Smith, and A. M. Tyrrell, "Digital circuit design using intrinsic evolvable hardware," in *Conference on Evolvable Hardware*, NASA/DoD., 2004, pp. 55–62.

[11] X. Yao and T. Higuchi, "Promises and challenges of evolvable hardware," *IEEE Transactions on Systems, Man and Cybernetics, Part C, Applications and Reviews*, vol. 29, pp. 87–97, 1999.

[12] R. S. Zebulum, M. A. Pacheco, and M. Vellasco, *Evolutionary Electronics - Automatic Design of Electronic Circuits and Systems by Genetic Algorithms*. New York: CRC Press, 2002.

[13] J. R. Koza, F. H. B. III, D. Andre, M. A. Keane, and F. Dunlap, "Automated synthesis of analog electrical circuits by means of genetic programming," *IEEE Transactions on Evolutionary Computation*, pp. 109–128, 1998.

[14] J. R. Koza, J. Yu, M. A. Keanne, and W. Mydlowec, "Evolution of a controller with a free variable using genetic programming," in *Proceedings of the European Conference on Genetic Programming (EuroGP)*, Edinburgh, Scotland, UK,, 2000, pp. 91–105.

[15] J. D. Lohn and G. S. Hornby, "Evolvable hardware using evolutionary computation to design and optimize hardware systems," *IEEE Computational Intelligence Magazine*, vol. 1, pp. 19–27, 2006.

[16] N. Nedjah and L. Mourelle, "Multi-objective evolutionary hardware for rsa-based cryptosystems," in *Proceedings of the IEEE International Conference on Information Technology: Coding and Computing*, Las Vegas - NE, U. S. A., 2004, pp. 503 – 507.

[17] T. Higuchi, M. Iwata, D. Keymeulen, H. Sakanashi, M. Murakawa, I. Kajitani, and E. Takahashi, "Real-world applications of analog and digital evolvable hardware," in *IEEE Transactions on Evolutionary Computation*, vol. 3, 1999, pp. 220–235.

[18] P. Pacheco, *Parallel Programming with MPI*. Morgan Kaufmann Publishers, 1997.

[19] M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2004.

[20] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*. Addison-Wesley, 2011.

[21] NVIDIA, "Cuda dynamic paralelism guide," Available at: <http://docs.nvidia.com/cuda/cuda-dynamic-parallelism/index.html, 2012, accessed in March, 2013.

[22] G. Karniadakis and R. M. KirbII, *Parallel Scientific Computing in C++ and MPI*. Cambridge: Cambridge University Press, 2003.

[23] D. Luebke, "Cuda: Scalable parallel programming for high-performance scientific computing," in *5th IEEE international symposium on biomedical imaging: From nano to macro*, Paris-France, 2008, pp. 836 – 838.

[24] NVIDIA, "Programming guide 4.2," Available at: <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>, 2010, accessed in March 2013.

[25] ——, "Cuda C best practices guide 4.1," Available at: <http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html>, 2010, accessed in March 2013.

[26] R. D. Salvo and C. Pino, *Image and Video Processing on CUDA: State of the Art and Future Directions*, 2011.

[27] Z. Yang, Y. Zhu, and Y. Pu, "Parallel image processing based on cuda," in *International Conference on Computer Science and Software Engineering*, Wuhan, China, 2008, pp. 198–201.

[28] C. Zou, C. Xia, and G. Zhao, "Numerical parallel processing based on gpu with cuda architecture," in *International Conference on Wireless Networks and Information Systems*, Shanghai - China, 2009, pp. 93–96.

[29] E. Cantu-Paz, *A survey of parallel genetic algorithms*. Calculateurs Paralleles, 1998, vol. 10.

[30] S. E. Eklund, "A massively parallel architecture for distributed genetic algorithm," *Parallel Computing*, vol. 30, pp. 647–676, 2004.

[31] J. Zhang, K. Lou, G. Liu, Y. Sun, and R. Gao, "Application of a parallel genetic algorithm for the optimal design of the flexible multi-body model vehicle suspensions," *Second IEEE Conference on Industrial Electronics and Applications*, pp. 793–696, 2007.

[32] J. Berger and M. Barkaoui, "A parallel hybrid genetic algorithm for the vehicle routing problem with time windows," *Computers & Operations Research*, vol. 31, pp. 2037–2053, 2004.

[33] E. Alba and M. Tomassini, "Parallelism and evolutionary algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 6, pp. 443–462, 2002.

[34] C. F. M. Toledo, L. de Oliveira, R. R. R. de Oliveira, and M. R. Pereira, "Parallel genetic algorithm approaches applied to solve a synchronized and integrated lot sizing and scheduling problem," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, 2010.

[35] C. F. M. Toledo, P. M. França, R. Morabito, and A. Kimms, "Multi-population genetic algorithm to solve the synchronized and integrated two-level lot sizing and scheduling problem," *International Journal of Production Research*, vol. 47, pp. 3097–3119, 2009.