Greedy Dynamic Crossover Management in Hardware Accelerated Genetic Algorithm Implementations using FPGA

Shubhalaxmi Kher  
Arkansas State University  
skher@astate.edu

T S Ganesh  
Ambarella corporation  
ganesh@ambarella.com

Prem Ramesh  
Iowa State University  
pramesh@iastate.edu

Arun K Somani  
Iowa State University  
arun@iastate.edu

Abstract

Genetic algorithms are robust parallel calculation methods based on natural selection. Various crossover and mutation methods to accomplish Genetic Algorithm (GA), namely, single point, multipoint, uniform, greedy, migration, and on-demand etc.; exist. However, these mechanisms are static in nature. This paper presents a dynamic crossover (DC) mechanism. We investigate its performance by implementing in hardware (FPGA) with convergence rate and higher fitness as the performance metric.

The purpose of the DC concept is two fold; to achieve faster convergence and to consume lesser memory by keeping the population size static. The results indicate that for a linear and a nonlinear objective function, DC outperforms all static crossover mechanisms.

Keywords:
Genetic Algorithm, Single-Point Crossover, Fixed point crossover, Dynamic Crossover, Fitness Function, FPGA.

1. Introduction

Over the past years Genetic Algorithm has been successfully applied to many NP-hard optimization problems such as robot trajectory planning, protein molecules shape identification, non-linear dynamic systems - prediction, data analysis etc. Genetic algorithms finds applications where there is practically no way to find a deterministic solution. These applications include network routing job scheduling of computer applications, database mining, molecular conformation in chemistry, equilibrium resolution in game theory, and bio-informatics applications such as protein folding, RNA structure prediction, sequence alignment etc.

GA are powerful optimization algorithms inspired by natural evolution. The optimization is performed by creating a population of solutions. Off springs are produced by standard genetic operators; reproduction, crossover, and mutation. In each generation a selection scheme is used to select the survivors to the next generation according to the fitness values defined by the user. With this artificial evolution, the solutions are improved generation by generation. The GA process starts with a random population and iterates until the termination condition is met [1, 2].

The power of GA has been well explored in the past through efficient crossover mechanism and hierarchical population [3, 4]. However, the GA process is time consuming and there is a need for optimizing it for better run time. For many real world applications, GA can run for days even when executed on a high performance workstation. One way to make GA efficient is through statistical approach that reduces the converging steps of GA to some extent [5].

In this context, a number of crossover techniques such as, single point crossover, uniform crossover, and multipoint crossover have been proposed. Another practical attempt to accelerate extensive computation is by designing hardware based genetic algorithms [6, 7]. Recently, several hardware implementations have been proposed and have shown to be effective in accelerating the run time of GA [8-10]. Also, it has been shown that hardware based implementation of GA outperforms software implementation by making use of the inherent parallelism within GA [9, 10]. Most of the existing works present with hardware acceleration for simple GA. However, complex and expensive hardware is employed to attain significant speedups [11]. However, each of the mechanisms focuses on the GA algorithm. We focus instead on the crossover management within the GA and reduce convergence time for the GA. To test our concept we employ it over a variety of complex functions. In this paper, we propose dynamic crossover algorithm and implement architecture for hardware accelerated GA implementation using FPGA for a nonlinear objective function.

The paper is organized as follows. Section 2 explains the concept of the GA. Section 3 describes crossover and mutation operators of GA in hardware architecture set up using FPGA. This is followed by making out a case for DC management and presenting its details. Section 4 gives hardware architecture design and its implementation details. This includes the design of DC module, mutation module the FPGA implementation details. Section 5 discusses results of the implementation and analysis. Finally, Section 6 presents conclusion.

2. GA Concept

Genetic algorithm is explained in Algorithm 1. Initial population of chromosomes is normally created by signing random values to each of the elements of each chromosome, with subsequent populations resulting from the application of the various genetic operators. Parents are selected for processing by a genetic operator so as to give more chances of reproduction to members of the current population that have the largest fitness values, i.e., members that represent a better solution to the problem under investigation. This can be achieved by means of roulette-wheel selection, in which \( P_{s} = \frac{Fitness(chromosome)}{Fitness(Chromosome)} \) (which may be scaled or normalized in
some way). The same procedure is used in the GA above to choose which genetic operator should be applied at each stage [12]. Chromosomes in a population can be represented in a number of ways. The simplest approach uses binary encoding, where each chromosome is represented by a bit-string of ones and zeros, but more complex encodings are also possible. We considered a 32-bit string for each chromosome. The position and number of crossover points in the chromosome is generated randomly for the crossover operator. Mutation operator takes a single chromosome and involves its alteration to allow the exploration of an alternative part of the potential solution space. Bit-string mutation is normally effected by replacing each one by a zero (or vice versa) if a randomly-generated number is less than selected threshold probability value.

3. Different Crossover Mechanisms

Various techniques of crossover operators include single point crossover, multipoint crossover, uniform crossover, greedy crossover, migration, etc. [13]. The prevalent method of crossover is to choose a random location within the chromosome/gene, split the parent at that position and interchange gene segments to generate new off-springs, parents, and continue to generate new off-springs locally again. They tend to generate the best solution finally in this method.

3.1 Dynamic Crossover Mechanism (DC)

The crossover operator is selected depending upon the complexity of the application and there is no generic solution for all applications. Essentially the crossover techniques are static. The number of crossover points and locations are fixed a priori. Further, these techniques also work for a fixed number of epochs (iterations).

DC technique is inspired by the greedy algorithm [14] as well as migration algorithm [15, 16]. This is a technique much more amenable to hardware implementations in comparison with software implementations. The techniques involved can be implemented to execute in parallel with the rest of the computations. In contrast, software implementations are inherently sequential and DC management imposes unnecessary overhead on algorithms which are already faulted as being too slow for practical purposes. We propose a crossover mechanism that dynamically updates the number of crossover points as well as locations based on the history of fitness values. Since the updates are carried out along with the population generation, the convergence is faster. To analyze the results, we consider a very simple linear function and a complex nonlinear function. We also vary the threshold to see if our approach is better in all the cases and claim that for the selected objective functions DC converges faster than single point, 2 point, or 3 point crossover mechanisms.

DC mechanism is given in Algorithm 2 below.

<table>
<thead>
<tr>
<th>ALGORITHM 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Assign weight to each genetic operator (reflecting its relative importance)</td>
</tr>
</tbody>
</table>
| 2 • Create an initial population  
  • Evaluate fitness of each chromosome |
| 3 Select an operator based on operator weights |
| 4 Select parents using selection criteria based on their fitness |
| 5 • Apply operators (crossover and mutation)  
  • Generate child chromosomes  
  • Evaluate their fitness |
| 6 Replace the least fit members of the population by the children if not already present in the population |
| 7 • check for solution  
  • If an acceptable solution is not found  
  • Repeat steps 3-7 |

<table>
<thead>
<tr>
<th>ALGORITHM 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definitions:</td>
</tr>
</tbody>
</table>
| PDF: Probability Distribution Function  
CP : Crossover Point, initially set to 1 |
| • Partition the population into equal blocks  
  \( B = \sum_{i=1}^{n} B_i \), where \( n \) is total number of processors, and \( |B| = |B_j|  
  \) |
| • For \( i = K \) (number of iterations/epochs) do  
  For each block \( B_i, i = 1,...,n \), execute steps 3-6 of ALGORITHM 1 w.r.t. to selected objective function \( f(x) \)  
  Evaluate fitness value \( f(x_i), \) \( x_i = \) offspring, local best solution of \( B_i \)  
  For \( i = 1,...,n \), Associate \( f(x_i) \) with \( f(x_j) \) \( (i \neq j) \) w.r.t. PDF and do migration by swapping \( x_{x_i} \) \( x_{x_j} \)  
  If \( f(x_i) \) does not improve, \( CP = CP + 1 \)  
  Desired solution given by \( max/min(f(x_i)), i = 1,...n \) |
4. Hardware Architecture Design

Our hardware architecture design and implementation [17] brings out the advantages of DC management in genetic algorithms, and does not aim at displaying the prowess of GA. The design and implementation is broadly based upon Tommiska and Vuori's AHDL description of hardware architecture for a generic GA implementation [18]. We extend this architecture to provide for DC management along with possible future extensions for dynamic mutation management. Further, our implementation is a single cycle one, though it can be made multi-cycle if the objective function demands so. The architecture is generic enough for any application with similar gene representations. It is enhanced by the fact that the fitness function can be designed as a plug-in module.

Figure 1 Genetic Algorithm Hardware Architecture

Figure 1 shows the details of the hardware architecture design and configuration of the components of the GA. We consider the population genome to be made up of bit strings, which incidentally is the most common genome representation. The design accepts either a static or DC operator. Towards this, there is a configuration input 'DynamicCO' which can be set to “High” to perform dynamic crossover. Static crossover can be obtained by not asserting this input. 'NumPoints' is a two-bit input which sets the number of crossover points in a static crossover scenario. For DC depending on the particular application, the end-user might like to vary the threshold for the difference in the fitness values of successive generations. The architecture allows for this by providing the facility to load a threshold value during initialization. The outputs of the system include the best offspring of the four instances evaluated in the present cycle, and the number of generations elapsed since initialization.

Each generation is processed in one cycle, though this could be modified in case the objective function is complex enough to warrant spending multiple cycles on. The initialization phase involves filling up of the 32x32 'Population RAM' with a random population. After this, two parent genes are selected at random and subjected to the crossover operation. Crossover is performed with 100% probability on these two parents to generate two off-springs. Depending on the mode of operation, the number of crossover points can be single or multiple. Depending on the crossover mode (static or dynamic), the crossover pattern 'COPattern' is generated. Details of the DC generation hardware are provided in the following subsection. The results of the crossover operation are subjected to mutation with a pre-determined mutation rate. In our present implementation, mutation on the off-springs is performed with a probability of 0.0645. Post mutation, we are left with four sample genes, two from the crossover-mutation operations, and the two original parents. A fitness evaluator (which can be designed as a stand-alone module) chooses the best two out of the four genes inputs according to their fitness values. At the end of the cycle, the two best genes and the RAM addresses from which the two parents were chosen are written into registers so that they may be updated on the 'Population RAM' in the next cycle.

The single cycle difference between the selection and writing back of the genes enables the DC management scheme to keep track of the history of fitness values. The current fitness value/gene is available at the output of the fitness module, while the best fitness value/gene from the previous generation is available from the registers mentioned previously. These are input to the dynamic control unit (DCU) (refer Figure 1) which compares the fitness values. Further, information about the current number of crossover points is available within the DCU. Depending on the threshold value and the input fitness values, the number of crossover points is either increased or decreased. A static crossover control unit generates control signals for the usual GA tasks, allowing it to be configured for both single point and multipoint static crossovers. The DCU is aided by the crossover pattern generator to implement DC. Just like all GA implementations, a multitude of random number generators based on Linear Feedback Shift Registers (LFSRs) is utilized for coordinating various aspects of the data path operations. The architecture for crossover and mutation modules is used from our earlier design [17].

4.1 FPGA Implementation Details

The architecture described in the above section was designed and simulated using VHDL and synthesized after verifying functional correctness on a Xilinx Virtex 2P (XC2VP4FG256-7) FPGA. The following section gives the results obtained after hardware synthesis using Xilinx Virtex II Pro, XC2VP30-6-FG676 FPGA. The code is developed in Verilog and VHDL independently. Details of hardware usage for each case are given below.

**GA Unit: Quadratic Function**

Number of Slices: 6468 out of 13696 47%
Number of 4 input LUTs: 12608 out of 27392 46%
Number of bonded IOBs: 86 out of 416 20%
Clock period: 113.613ns (Max frequency: 8.802MHz)

**Quadratic Function Module**
Number of Slices: 145 out of 13696 1%
Number of 4 input LUTs: 279 out of 27392 1%
Number of bonded IOBs: 96 out of 416 23%

**Control Unit – 2- point/ 3point / DC Module**
Number of Slices: 41 out of 13696 0%
Number of 4 input LUTs: 81 out of 27392 0%
Number of bonded IOBs: 102 out of 416 24%
Minimum period: 4.692ns (Max Freq: 213.129MHz)

The design is dependent on the objective function under consideration. For all practical purposes, the evaluation of the objective function would be split over multiple cycles in order to maximize throughput. The architecture is robust enough to handle pipelining if the objective function demands so. To evaluate the architecture for design efficiency, the objective function modules were synthesized separately on Xilinx Virtex 2P (XC2VP4FG256-7) FPGA. It took 227 slices. On Xilinx Virtex II Pro, XC2VP30-6-FG676 FPGA, we synthesized the quadratic fitness function taking 145 slices. Any practical implementation of the DC enhanced genetic algorithm would thus take up to 1,879 and 3,077 slices on XC2VP4FG256-7 and XC2VP30-6-FG676, respectively, in addition to the requirements of the objective function. The FPGA resources are mapped to the application in such a way that the slices are not utilized fully. Hence, pipelining will not cause much higher penalty. This is because pipelining may take advantage of the flip-flop resources available in each slice, which are currently not used. The current design only uses purely combinational circuit for the objective function. In fact, more complex objective functions are likely to lead to efficient usage of the FPGA resources.

5. Results and analysis

All values are in Hexadecimal.

**Filtering:** Filtering reduces the computation without affecting the performance. Threshold values for the genes help in filtering genes that are too low.

**Threshold Selection:** Low threshold increases the number of cycles to converge. Choosing between high value for threshold and very high threshold is not trivial. Very high threshold e.g., 0x7FFFFFFF (for linear objective function) and 0x70000000 (for nonlinear objective function) leads to several unassigned values in the population RAM, as high thresholds lead to fewer distinct values that can be written into RAM. Thus threshold selection is an important criterion. (refer Table 1)

- When new offsprings are generated, the population RAM is checked for any entries that match with these. If a match is found, the genes are not written back thus duplication is avoided.
- To start the next iteration, two genes are selected from the RAM, based on roulette wheel selection. Two best genes from these two and the current offspring are selected to be the next parents.
- Previous step leads to faster convergence to a near optimal solution but more likely to fall into local maxima/minima. To overcome this we pick up two random genes and start the simulation process again. In this way, each time the algorithm gets saturated in a local value, we push it further near the optimal by re-starting it. This gradually improves the result.
- Choosing and implementing a fitness function in hardware is a real hard task. Most mathematical functions based on real numbers are not synthesizable in FPGA. For example, we tried implementing the exponential function. Although we could run the functional simulation of this, we could not synthesize on FPGA. So we designed an approximate method described next.
- For testing the GA using various crossover mechanisms we consider a nonlinear functions and implement using unsigned numbers. We spent much time in overcoming the round off errors. We consider following nonlinear function to find their maximum value and implement

\[ F(x) = a' - ((x-a)/a); \text{where } a = 2^{(m-1)}, \text{ } n = \text{number of bits in the gene}; \text{(in our case } n = 32) \]

This is a quadratic function, viz., a parabola whose theoretical maximum value is at 0x7FFFFFFF (2147483647 decimal), half of the range. We implement GA using dynamic crossover and compare its performance for GA using single point, 2-point, and 3 point crossover independently. Table 1 shows the final converged results of our implementation compared to theoretical maximum.

<table>
<thead>
<tr>
<th></th>
<th>Single Point Crossover</th>
<th>Dynamic Crossover</th>
</tr>
</thead>
<tbody>
<tr>
<td>Theoretical Max</td>
<td>0x7FFFFFFF</td>
<td>0x7FFFFFFF</td>
</tr>
<tr>
<td>% of Theoretical Max</td>
<td>99.2164</td>
<td>99.9990</td>
</tr>
</tbody>
</table>

| No Threshold            | 0x7FFFFFFF             | 0x7FFFFFFF        |
|                        | 0x7FFFFFFF             | 0x7FFFFFFF        |
| % of Theoretical Max     | 99.2164                | 99.9990           |
| Threshhold              | 0x7FFFFFFF             | 0x7FFFFFFF        |
|                          | 0x7FFFFFFF             | 0x7FFFFFFF        |
| % of Theoretical Max     | 99.2164                | 99.9990           |
| Thresshold              | 0x7FFFFFFF             | 0x7FFFFFFF        |
|                          | 0x7FFFFFFF             | 0x7FFFFFFF        |
| % of Theoretical Max     | 99.2164                | 99.9990           |

**Table 1. Simulation Results for Different Crossovers**

The performance of the proposed DC management is evaluated by implementing GA using nonlinear objective function. [17] gives design and performance of GA with DC for a linear objective function. It is known that implementing GA using hardware is faster as compared to its software counterpart. However, implementing using hardware is application specific and in each case the rate of convergence may be different.

5.1 Quadratic Function (Threshold t = 0)

For the nonlinear function given above we investigate performance of GA using single point, 2 point, 3 point, and dynamic crossover mechanism for function with no
threshold (all the population considered for solution). We also consider a threshold value to see if GA performs better with restricted population. Applying a threshold necessarily reduces the population as all the values below the threshold will not be a candidate for crossover. In other words we test the GA with respect to two conditions of threshold; namely i) no threshold, ii) threshold = 0x70000000. The theoretical maximum for the non-linear objective function is 0x7FFFFFFF (2147483647 in decimal). We achieve x = 0x7EFF3F53 (2130657107 in decimal), 99.2164% of theoretical maximum. Figure 3(a), (b), (c), and (d) depict performance of various crossover mechanisms with t=0.

Figure 3(a) Nonlinear function; single point; t=0
Figure 3(b) Nonlinear function, 2 point, t= 0
Figure 3(c) Nonlinear function, 3 point, t= 0
Figure 3(d) nonlinear function, DC, t=0

5.2. Quadratic Function (Threshold = 0x70000000)

Figure 4(a) shows output of GA using single point crossover with a threshold value ‘t’. It attains a maximum value very quickly. However, it is unstable and fluctuates at a high value as shown in 4(a). GA output with 2 point crossover is unstable as given in Figure 4(b). Figure 4(c) shows GA output attaining a high value in nearly 75 iterations. However, output modulates between two values and fails to stabilize after 500 generations.

Figure 4(a), single point; nonlinear; t= 0X70000000
Figure 4 (b), 2 point; nonlinear; t= 0X70000000
Figure 4 (c) 3 point, nonlinear; t= 0X70000000
Figure 4 (d) DC: nonlinear; t= 0X70000000

For GA with 2-point crossover Figure 3(c) shows output of GA using 3-point crossover. It performs better than 2-point crossover, reaches the same maximum value. However, it toggles between a higher and a lower bound. Figure 3(d) shows stable and better output of GA using DC compared to other crossover mechanisms.
Figure 4(d), GA using DC attains a max value in 82 iterations using DC. It also becomes stable for the maximum value in the next 300 iterations. Interestingly, as we limit the crossover operation for 500 generations, it is observed that after achieving a maximum value stabilizing till 370 iterations, it starts to toggle between the achieved maximum value and a lower bound. We may choose to stop the generations once the maximum value is achieved and stabilized. We achieve $x = 7FFF7EB$ (2147477483 in decimal), 99.9999% of theoretical maximum using this threshold.

6. Conclusion

Genetic algorithms are significant methods to solve applications where there is practically no deterministic solution. However, these algorithms take more time to converge when implemented in software [16, 18]. We developed a genetic algorithm with dynamic crossover mechanism to exploit the inherent parallel nature. Dynamic crossover mechanism finds near maximum/minimum value of the selected objective function by dividing the entire population into blocks for parallel execution. It dynamically manages crossover points and migrates during runtime. The migration is performed by randomly pairing the blocks according to the selected probability distribution function (PDF). Our results clearly indicate that algorithm outperforms other static approaches. We have shown that dynamic crossover management is well suited for parallelized versions by efficiently supporting migration. DC mechanism is implemented for hardware accelerated GA using FPGA. The outcome is faster convergence with limited memory resource. Currently, implementation supports linear [17] and nonlinear objective functions, the evaluation is done using hardware complexity and convergence rate as the metric. We infer the following from our results:
1. It is observed that DC is better irrespective of function, threshold, and iterations. For the select nonlinear objective functions DC converges faster than single point, 2-point, or 3-point crossover mechanisms.
2. For linear and non-linear objective functions, we achieve maximum value using GA with DC that reaches up to 99.9999% of the actual maximum value given by the function [15].
3. As shown in the FPGA implementation details in section 3.3, the control unit for implementing a nonlinear (complex) function using 2-point, 3-point or DC employs exactly same number of slices. In other words, Hardware implementations for all the functions and threshold values as given in Section 4.
4. For less complex linear functions, it is observed that the behavior of the DC mechanism is almost identical to that of single point crossover. This is in accordance with the fact that there may be virtually no need to employ the DC as single point crossover is sufficient enough.
5. We applied a set of other linear functions to verify our results. From our observations, we conclude that dynamic crossover management is the way to future hardware accelerated genetic algorithms establishing substantial a performance improvement.

7. References