Verifying an Implementation of Genetic Algorithm on FPGA-SoC using SystemVerilog

Hayder Al-Hakeem Suvi Karhu Jarmo T. Alander

Department of Electrical and Energy Engineering, University of Vaasa, Finland, {firstname.lastname}@uva.fi

Abstract

In this paper we show how an efficient implementation of genetic algorithms can be done on Field Programmable Gate Array i.e. on programmable hardware using the latest hardware design language aiding verification. A fourway number partitioning problem of 128 unsigned 16-bit integers is used as a test case of the implementation. However, other similar problems could be solved using the proposed approach. The design was implemented using a combination of reusable verified intellectual property cores for arithmetic operations and VHDL to describe the genetic algorithm operators in register transfer level. The register transfer level components were verified in ModelSim using SystemVerilog assertions and covergroups. Test results show significant improvements in performance compared to C language implementation running on a core i-7 desktop computer.

Keywords: genetic algorithms, verification, FPGA, system on chip (SoC)

1 Introduction

The idea of using hardware to speed up processing of evolutionary algorithms is not new (Alander, 2008; Alander et al., 1995). Here we show how the implementation can be done in a way that uses the latest verification techniques that are available in modern hardware design languages such as SystemVerilog.

The proposed Genetic Algorithm (GA) implementation is tested on a number partitioning problem that belongs to the set of NP complete problems meaning that its solution might be intractable in practice. However, many real life optimization problems belong to the NP complete set and must be somehow solved approximately for practical purposes. Having enough computing power helps somewhat. Field Programmable Gate Arrays (FPGA) are energy efficient and fast due to their massive parallel processing. In problems that they are suitable, they can be significantly faster than a PC while using only a fraction of the energy of a corresponding processor solving the same task. An obvious drawback of FPGAs is that they need both programming and hardware design skills. Thus, creating high quality implementation solution on FPGA is both demanding and needs a lot of testing and verification. Prototype Verification System (Owre et al., 1992) has been used to verify crossover operator in GAs (Nawaz et al., 2013). And vice versa, GAs have also been used to optimize verification (Gao et al., 2015; Cheng and Lim, 2014).

2 Formulating the GA Optimization

The basic principle of GA is that if some randomly generated solutions can produce good results, those solutions can be combined and used as building blocks to generate better solutions. Solutions are evaluated by calculating a fitness function, they are then modified using techniques inspired by natural evolution, such as inheritance, mutation, selection, and crossover. The new generated solutions are then re-evaluated and the procedure is repeated until the target of fitness optimization is achieved or a predefined number of iterations is reached. (Alander, 1992)

2.1 The Problem Encoding

The number partitioning problem is often labeled as the easiest hard problem (Hayes, 2002). While being considered as one of the classical NP-hard problems of combinatorial optimization, it is fairly easy to understand, represent, and evaluate.

According to (Korf, 2009), "The number partitioning problem is to divide a given set of integers into a collection of subsets, so that the sum of the numbers in each subset are as nearly equal as possible". In this work, a four-way partitioning problem is used to verify the functionality and benchmark the performance of our hardware GA implementation. The goal is to split a set of N=128 randomly generated positive integers I_i of length 16 bits into four subsets so that the sums of those subsets are as equal as possible.

Figure 1 illustrates the chromosome (solution trial) representation.

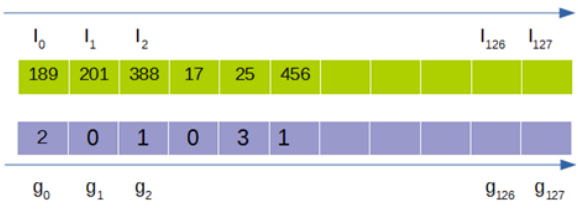

Figure 1. Chromosome representation when N=128

Each chromosome consists of N genes. A binary representation is used, where each gene consist of two bits. Each gene g_i (i=0, .., N-1) can take one of the following

four binary values; "00", "01", "10" or "11" indicating that the corresponding integer I_i belongs either to subset 0, 1, 2 or 3 respectively. A brute force search would require the evaluation of $4^{128} = 1.158 \times 10^{77}$ possible solutions which is intractable.

2.2 The Objective Function (Fitness)

For n number of subsets, the summation of each subset (S_0, S_1, S_2,..., S_n) is computed first. Matlab simulations showed that the standard deviation or variance produced good results. However, to reduce the complexity and cost of hardware implementation, a more simplistic fitness function is implemented. The function adds the difference between every possible permutation pair of the subset sums as illustrated in Equation 1. The objective is to minimize the fitness.

$$\sum_{i=0}^{n-1} \sum_{j=i+1}^{n} |(S_i - S_j)| \quad (1)$$

In case of a 4-way partitioning problem (n=4), unpacking equation 1 yields to Equation 2.

$$Fitness = D_{0,1} + D_{0,2} + D_{0,3} + D_{1,2} + D_{1,3} + D_{2,3} \quad (2)$$

Where $D_{i,j}$ is the absolute value of the difference between S_i and S_j

2.3 Selecting The GA Operators

Since the GA is intended to be implemented and verified in FPGA, the simplest possible set of operators that can provide satisfactory results were investigated using Matlab simulations. In these simulations, the number of generations was fixed at 2000. For each set of operators, the experiment was repeated 1000 times. For each experiment, the error was defined as the difference between the largest and smallest of the four subsets sums divided by the sum of the whole set. If the average error of the 1000 experiment is $\leq 1\%$, this insures that the sum of each of the four subsets is deviating less than 1% from the optimal 25% of the whole set sum. The operators are considered good enough and a simpler set of operators is simulated. After running several simulations, the simplest set of operators that was able to achieve an average error less than 1% (0.23%) was chosen as follows:

- Tournament Selection was used.
- One point crossover was used.
- For Mutation, a positive random integer x is generated for each gene, where $0 \leq x \leq a$, a is a constant that is used to adjust the mutation rate. The random integer x is then compared to a predefined constant integer c, where $0 \leq c \leq a$. If x equals c, the corresponding gene is replaced with two random bits. Otherwise no change is performed. Therefore, the mutation rate can be controlled by changing a, and is defined by $MR = 1/(a+1)$.
- For replacement, the chromosome with worst (largest) fitness value is replaced with the new offspring before the generations counter is incremented.

2.4 Tuning the GA Parameters

Matlab simulations are used to fine-tune the population size, mutation rate, number of generations and the tournament size. Each time, one parameter is varied and the rest are set to fixed values. One thousand experiments, each with a new set of 128 random integers are performed and the best fitness of each experiment is recorded. After 1000 experiments have been performed, the results of those experiments are averaged. The standard deviation is also taken into consideration in order to guarantee a lower probability of getting a bad solution which is highly deviated from the average result.

To evaluate the best size of the selection tournament, The number of generations was set to G=2000, the mutation rate was MR= 1%, and the population size was set to $N_p = 128$. Table 1 shows the averaged best fitness of 1000 experiments and its standard deviation using different tournament sizes. The best results are obtained when the tournament size is 32/128 (i.e. 1/4 of the population size N_p). Since 16 bit numbers produce large sums and therefore large fitness values, the fitness values are illustrated with the k (kilo) metric prefix for convenience.

Table 1. Optimizing tournament size $\mathbf{N_T}$.

$\mathbf{N_T}$	Average Fitness	STD of Fitness
4/128	12.187k	7.068k
8/128	7.201k	4.103k
16/128	7.057k	4.310k
32/128	**6.623k**	**4.158k**
64/128	6.8646k	4.575k

To evaluate the best mutation rate, the number of generations was set to G=2000, the $N_p = 128$ and the tournament size was set to $N_T = \frac{1}{4}N_p = 32$. Table 2 shows the averaged best fitness of 1000 experiments and its standard deviation using different mutation rates. Best results are obtained when MR=1%.

Table 2. Optimizing mutation rate **MR**.

MR	Average Fitness	STD of Fitness
0.25%	10.924k	6.448k
0.5%	8.518k	5.513k
1%	**6.623k**	**4.158k**
2%	6.686k	3.944k
4%	13.656k	6.909k

To evaluate the best population size, The following parameters were fixed; G=2000, the MR=1% and $N_T = \frac{1}{4}N_p$. Larger populations did not provide significant improvements as seen from table 3.

Table 3. Optimizing population size $\mathbf{N_P}$.

N_P	Average Fitness	STD of Fitness
16	**7.026k**	**4.246k**
32	6.892k	4.369k
64	7.085k	4.219k
128	6.623k	4.158k

As for the number of generations, no significant improvement was observed after G=2000. Hence, a fixed number of 2000 generations is selected to simplify hardware implementation.

Based on the simulation results, the following parameters were selected for implementation; N_p=16, N_T=4, MR=1% and G=2000.

3 FPGA Implementation

The simulated GA was implemented using RTL (Register Transfer Level) description in VHDL. Each of the GA operators was implemented in a separate VHDL file to simplify the verification process. A top module called GA instantiates, connects and controls the operation of the GA operators using a finite state machine. Arithmetic operations were implemented using Altera's verified fixed point numbers IP cores. The design is optimized to achieve the best possible performance at the expense of size.

3.1 GA's HDL Entities

3.1.1 Fitness

In order to evaluate the sum of integers in each subset, a separate parallel adder with N=128 inputs is instantiated for each subset. The integers are multiplied with a binary mask before being inputted to the parallel adder to zero the integers that do not belong to the subset in question. This approach is illustrated in Figure 2.

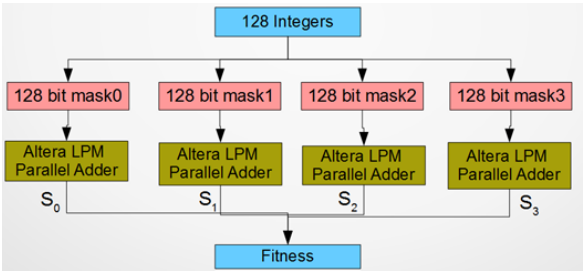

Figure 2. Fitness Circuit

The bit masks are generated from the genes of the chromosome being evaluated. For example, if the fourth gene g_4 in the chromosome has the binary value "11" (decimal three), this means that the integer with index 4 belongs to subset 3. Consequently, bit 4 in mask3 will be assigned '1' while bit 4 in the three other masks will be assigned '0' as shown in Figure 3.

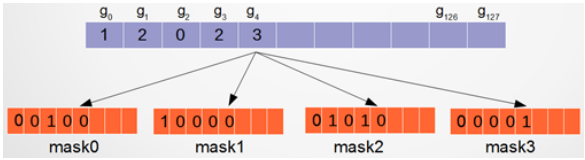

Figure 3. Adder's Masks

Fitness is then calculated from the subset sums using equation 2. The fitness operator is thus purely combinatorial requiring only a single clock cycle to sample and store the result.

3.1.2 Selection

The selection entity accepts 4 random candidates and uses a pair of comparators to output the best two candidates in a single clock cycle.

3.1.3 Crossover

The crossover entity has two parents and a 7-bit random integer (crossover point index) as input. Each clock cycle, it copies half of the genes (higher than the crossover point index) from parent 1 and the other half of genes from parent 2 and outputs a new offspring.

3.1.4 Mutation

For each new offspring, a positive random integer x is generated, where $0 \leq x \leq N-1$. N is the number of integers to be partitioned which is equal to the number of genes in each chromosome. The gene that has an index equal to x is replaced with a new randomly generated gene $\in [0,3]$. As a result, the mutation rate MR=1/128 = 0.78125% which is close to the best mutation rate of 1% obtained from Matlab simulations.

3.1.5 Generating Pseudo Random Sequences

In order to generate random numbers, maximal sequence linear feedback shift registers (LFSRs) are used. Tap locations are obtained from Xilinx tables (Alfke,1996). A separate LFSR is used for each required random number and initialized with a different seed to improve randomness.

The population LFSR is a 256-bit LFSR used to generate the initial population and replace mutated genes with new random values. The selection LFSR is 16 bits wide where each four bits are interpreted as an index from 0 to 15 allowing the selection entity to randomly pick 4 parent candidates each clock cycle. The crossover LFSR is 7 bits wide and it describes the single point crossover location. Similarly, the mutation operator requires a 7-bit long index for selecting which gene will be mutated.

3.1.6 Replacement

In order to identify the index of the individual with the worst fitness, a 16 numbers comparator consisting of 4 stages tree of two number comparators is implemented. The output of each stage comparators named "a greater than or equal b" (ageb) is used to backtrack the tree and identify the index of the individual with the worst (largest) fitness. Figure 4 illustrates the process using, for convenience, a smaller 8 number 3 level comparator. Each level produces 1 bit of the index of the largest fitness.

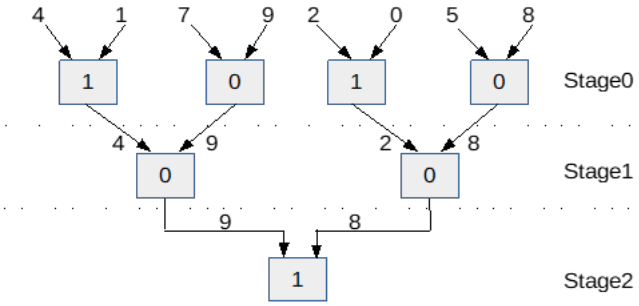

Figure 4. Largest fitness evaluation for 8 values using 3 levels tree

Each of the boxes represents a comparator of two integers. The number inside the box represents the binary result of $A>B$ comparison where A is the left hand side input integer and B is right hand side input integer. Backtracking from stage 2 to stage 0 we obtain the binary sequence "100" which indicates that the largest number exists in index 4 (indexing from right to left). The same procedure can be scaled to compare a larger number of integers adding one extra stage of comparators and one extra bit to the index's LSB each time the number of compared integers is doubled. However, resource utilization and timing constrains must be carefully examined.

3.2 Performance Evaluation

A SystemVerilog *testbench* was created to simulate the proposed GA in Modelsim. A state named report is added to the state machine to be executed after reaching the maximum generation to send the results before resetting the GA.

The *testbench* uses a class that generates N random positive integers from a user defined seed and stores them in an array. Those random integers are buffered serially to the GA. The GA then initializes the population and runs for 2000 generations. After that, the GA buffers thirteen 32-bit words to the *testbench*. The first 8 words are the best solution (chromosome) with the most significant word buffered first. The next 5 words are the best fitness, S_0, S_1, S_2, S_3 and finally a counter of the number of clocks since the GA has started receiving the input.

The SystemVerilog *testbench* then uses the reported best solution to split the integers into 4 subsets and calculate the sum of each subset. The fitness value is calculated using equation 2. The *testbench* will then assert if the calculated sums and fitness match the ones reported by the GA implemented in VHDL. In case of a mismatch, an error message is printed and simulation is interrupted. Otherwise, results are printed and a new set of random integers is generated and buffered to the GA.

Initializing the integers requires 128 clock cycles (in case there are no wasted clock cycles by the input between successive integers). N_p number of clock cycles are required to initialize the population, one item per cycle. In each generation, 4 clock cycles are required to sequentially calculate selection, crossover, mutation, and finally the replacement of the worst individual with the new offspring. Additional N_p clock cycles are required to compare the fitness values of the final generation and extract the best solution. Thirteen clock cycles are consumed reporting the results. Finally, 4 extra clock cycles are required for switching between other internal states in the state machine. The total number of required clock cycles thus becomes (equation 3).

$$CLK(G,N_p) = 4G + 2N_p + 145 \tag{3}$$

ModelSim simulations report a clock count of 8177 when G=2000 generations and N_p=16, which matches exactly Equation 3. Using a clock with 25MHz, the required time per experiment is 8177/25MHz = 0.32708 milliseconds. In other words, the expected performance of the implemented GA is 3057 experiments (each consisting of partitioning N=128 integers) per second.

3.3 Functional Verification

The verification is performed with SystemVerilog assertions and covergroups. We use random inputs and require that certain coverage for each input will be achieved. At each iteration step we check that the results of the operator under verification are correct. Measuring the coverage is an essential step in verification (Wile et al., 2005). In addition to verifying the results, we verify that certain intermediate results inside the modules are correct. For this purpose we need to create new output ports to the modules. The verification is performed by a person other than the designer, but the approach is still that of grey-box verification where the verifier is aware of certain features of the source code of the system.

3.3.1 Fitness

In the verification of the fitness module we verify that the sums and the final fitness value are correct. Since the amount of possible values for the input chromosome is large (2256), we divide the possible values into 216 bins and require that each of them will be covered. In addition, we require that the four special cases where all the genes of the chromosome are the same (000000_2, 010101_2, 101010_2 or 111111_2) will be covered. We also require that at least 6 of the cases where all the numbers belong only to 2 groups will be covered, one for each combination of groups. The same is required for 3 groups, where 4 cases are required, one for each combination of

groups. Also a case where all 4 groups are present must be covered. The numbers to be partitioned are 16-bit and every possible value for them must be covered. However, since there are 128 numbers, not each value for every number have to be covered, but each value must occur at least once among the numbers. In addition we require that the case where all the numbers to be partitioned have the maximum value (216), will be covered. This case must also be cross covered with the cases where all the numbers belong only to 1, 2 or 3 groups. In this way we verify that no overflow occurs in the fitness calculation.

In the calculation of the sums the fitness module utilizes masks that tell which group each number belongs to. Each number should belong to exactly one group. To verify this we sum the masks pointwise. The sum should be 1 at each index. In this test we use the same coverage requirements as when verifying the results.

3.3.2 Selection

The selection module selects the parents for the next crossover. We verify that the operator selects the correct parents specified by the input *selection LFSR* and the fitness values, and reports the correct worst individual. There are sixteen 25-bit fitness inputs, and for each of them we create 216 bins which must be covered. Every value of the 16-bit *selection LFSR* must be covered. In addition we require that at least one of the cases where all the fitness values are the same will be covered.

3.3.3 Replacement

The selection module also contains the replacement functionality. We verify that the replacement works as intended. For this purpose we add a new port which contains the array of fitness values that are stored inside the module. We require that each possible value for the new fitness input will be covered. We also verify that the selection works correctly after the replacement. For that purpose we use the same coverage requirements as described in the Selection section.

3.3.4 Crossover

The crossover module is a rather simple, one-point crossover. The crosspoint is a 7-bit input ranging from 0 to 127. We require that every possible value of the 7-bit crosspoint will be covered. We especially focus on the case where the value is 0 or 127. The crossover component should change the value 0 to 1 and the value 127 to 126, because otherwise the result would comprise only of parent1 or parent2. For both 256-bit parents we create 216 bins which must all be covered. Because the area near the crossing point is the most error-prone, we require that per each index, every combination of 6 gene values around the index is covered, 3 genes for both parents. We require that a case where the parents are the same is covered.

3.3.5 Mutation

The inputs of the mutation module are the 256-bit child to be mutated, the 7-bit index to the gene to be mutated and the random 256-bit individual, the *population LFSR*, which will provide the new content for the gene to be mutated. We verify that the operator will mutate the correct gene with correct value specified by the inputs. We require that for the child and the *population LFSR*, 216 bins will be covered. We also require that all 27 values for the index will be covered. No cross coverage requirements will be set.

Each gene of the child has a probability of P=1/128 to proceed into mutation phase, where the content of the gene will be replaced with the content from the corresponding gene from the *population LFSR*. We call the probability P as the internal mutation rate. The probability that the new content is different than in the original gene is C=3/4. Thus the overall probability that the content of a particular gene will change is $CP = C \times P = 3/512$. We call C as the external mutation rate and CP as the overall mutation rate.

We verify that the mutation rate of the component is correct. We use the same coverage requirements as when verifying the results. We examine one gene and check if the content will change in CP of the cases. We repeat this for each gene. If the actual mutation rate differs from the correct rate, it may be because the *testbench* does not generate purely random but pseudo random values for the index and *population LFSR* inputs. To verify that this is the reason we make a test where the effect of the external mutation rate is eliminated, by keeping the *population LFSR* always different than the child to be mutated. Now the result should be close to P. If the result still differs, it may be due to the fact that the index input is also pseudo random. However the difference should be smaller than when the effect of the external mutation rate is present.

3.3.6 LFSRs

The sequence generated by an LFSR is a binary numeral system just like natural binary code or Gray code. Eventually the shift register enters a repeating cycle. To obtain good pseudo random numbers the cycle should be as long as possible This is called maximal length sequence. The maximal cycle length is $2^n - 1$ where n is the number of bits in the LFSR.

We wanted to verify that the sequences generated by the LFSR module were maximal-length. For the shortest outputs this is possible by using covergroups. During a cycle of $2^n - 1$ executions every possible value except all ones should be covered. With the 256-bit population LFSR we cannot do this. For this output we will simply verify that the sequence is correct for the first 216 cases. We can do this because we know the seed and the random sequence generating function. We verify the correctness of the sequence also in the case of the smaller LFSRs. At least 216 first numbers are checked for each output sequence.

3.3.7 Comparator

The system also contains a comparator module for determining the largest of sixteen 25-bit numbers. For each input we create 216 bins which must be covered. Also

at least one of the cases where the numbers are the same must be covered.

4 Integrating GA into the SoC

In this work, GA was implemented and tested on Terasic's SoCKit hardware, which contains the 925 MHz, Dual-Core ARM Cortex-A9 MPCore Hard Processor System (Terasic, 2015).

4.1 Interfacing GA with the HPS

In order to communicate with and control the GA from the HPS, Xillybus IP core was used. As the authors describe it in their website, "An FPGA IP core for easy DMA over PCIe with Windows and Linux" (Xillybus, 2017). Xillybus comes with a Linux distribution called Xillinux that runs on the FPGA embedded ARM processor and communicates with the Xillybus IP core via a device driver. The driver allows the developers to easily communicate with the FPGA using C language's *read()* and *write()* commands with FIFO buffers. Xillybus allows users to create online accounts to customize then download IP cores. The IP core factory is available to try for free for researchers (Xillybus, 2017). However, it requires licensing for commercial usage. Using Xillybus provides several advantages:

- Streaming data to and from FPGA is done using DMA without impacting the performance of the operating system.
- Xillybus supports a wide range of both Altera and Xilinx FPGAs and is part of the official Linux Kernel device drivers making the design more portable.
- The same design can also be used by connecting the FPGA as a coprocessor via *PCI Express* with an external host running either Microsoft windows or Linux.

Xillybus bus clock runs at 100 MHz supporting a maximum bandwidth of 400 M-Bytes/sec (32-bit interface). To analyze timing requirements for the GA clock, Altera's Time Quest Timing Analyzer was used. A timing netlist based on the "Slow 1100mV 85C Model" with 0 IC delays option unchecked was used to reflect the worst possible case scenario. Altera's tool reported an F_{max} of 36.99 MHz. The GA was tested practically on the device and found to operate correctly at 50 MHz. However, the GA clock was restricted to 25 MHz to ensure reliable operation in the worst conditions. This clock is derived from the Xillybus bus clock using Altera's Phased Locked Loop IP core. In order to enable cross clock domain communication between GA and Xillybus, two Altera *DCFIFO* (dual clock FIFO buffers) entities were instantiated and connected to the GA inside a new top module. The result top module has only standard FIFO ports plus the required clocks and resets and can be directly port mapped to any HDL design that contains standard 32-bit FIFO buffers without any further modification.

4.2 Controlling GA from the Linux host

The Xillinux is a standard Ubuntu desktop for ARM processor that comes with the gcc compiler and a rich preinstalled collection of packages and libraries.

In order to test the GA on the SoCKit, a C program that performs the same functionalities as the SystemVerilog *testbench* was written and compiled using gcc running on Xillinux. The C program opens Xillybus drivers which exist under the Linux /dev directory using the C *open()* command. It then uses the C write() command to write random integers to the Xillybus *write* FIFO buffer and waits for the GA report. When GA is finished, it buffers the results to Xillybus *read* FIFO buffer and resets waiting for new integers. The results are then read by the C program using the C *read()* command and verified.

If no errors are detected, the C program visualizes the results by using the Linux system API to call *GNUplot* and pass the sums of the initial partitions as well as the optimized sums reported by the FPGA. *GNUplot* generates and displays two charts stacked horizontally to visually compare the optimized and non-optimized partitions.

On the SoC FPGA, the reported clock count averaged at 200000 $\pm$ few thousand clock cycles per experiment. Running 1000 experiments required about 10 seconds. However, this is much slower than the reported clock counts by Modelsim's simulations. After examining the buffers interfaces using Altera's Signal Tap 2 Logic analyzer, it was found that the C function *write()* is causing large delays. A quick remedy is to increase the size of the FIFO buffers and write all the required integer for 1000 experiments (1000*128*4=512000 bytes) as one block. This indeed reduced the delay of performing 1000 experiments to less than 2 seconds.

For performance comparison, the same GA was implemented using C language and compiled using *GNU GCC* "gcc (tdm64-1) 5.1.0" with *CodeBlocks* on a desktop computer running Windows 10. The PC has has an Intel core i-7 4770 @ 3.4 GHz with 8 GB DDR3. Ten experiments were perfomed. In each experiment, 1000 number partitoning problems are solved. The average time was calculated at 9.8063. These results show and improvement of 500% in performance when using FPGA versus a PC due to the massive parallelism of the FPGA implementation. However, in order to utilize the full performance of the GA, it must be either controlled by a real time operating system, or at least by a C program running as a kernel module.

5 Conclusion and Future Work

In this work, we have designed, simulated and verified the functionality of a genetic algorithm to solve a 4-way NP complete partitioning problem of 128 16-bit unsigned integers. GA operators and parameters were selected based on Matlab simulations. The GA was implemented using VHDL and Altera's IP cores and verified using SystemVerilog with ModelSim. The design was then tested on Terasic's SoCKit as a coprocessor to accelerate the algorithm's execution on the embedded ARM Cortex-A9 MPCore processor. Test results demonstrate that using the massive parallelism of FPGAs, it is possible to achieve multiple times higher performance while using a fraction of the size and energy consumption of modern desktop computers. We are currently working on the comprehensive functional verification of the proposed design. In the future, we are planning to use a combination of standard functional coverage and rigorous formal verification techniques to meet industrial verification standards. GA could also be used in software testing (Mantere and Alander, 2005) and the objective could be e.g. the verification of the recently proposed flexible floating point numbers called unums (Gustafson, 2015).

References

Jarmo T. Alander. Indexed bibliography of genetic algorithms genetic algorithms and evolvable hardware and FPGAs. *Report 94-1-FPGA*. University of Vaasa, Dept of Information Tech and Production Econ, 2008. http://www.uwasa.fi/ TAU/reports/report94-1/gaFPGAbib.pdf

Jarmo T. Alander, Mikael Nordman, and Henri Setälä. Register-level hardware design and simulation of a genetic algorithm using VHDL. *In Proceedings of the MENDEL'95*, 10-14, 1995.

Jarmo T, Alander. On optimal population size of genetic algorithms. *In Proceedings of Comp Euro 92, Computer Systems and Software Engineering*, 65-70, 1992. doi:10.1109/CMPEUR.1992.218411.

Peter Alfke. *Efficient Shift Registers, LFSR Counters, and Long Pseudo Random Sequence Generators*. Xilinx Corporation, 1996.

Adriel Cheng and Cheng-Chew Lim. Optimizing system-on-chip verifications with multi-objective genetic evolutionary algorithms. *Journal of Industrial and Management Optimization*, 10(2):383-396, 1996. doi:10.3934/jimo.2014.10.383

Shi-Yi Gao, Xiao-Hua Luo and Yu-Feng Lu. Functional coverage convergence technique based on genetic algorithm. *Journal of Zhejiang University (Engineering Science)*, 49(8):1509-1515, 2015. doi:10.3785/j.issn.1008-973X.2015.08.015

John L. Gustafson. *The End of Error: Unum Computing*, 2015. CRC.

Brian Hayes. The Easiest Hard Problem. *American Scientist*, 90(2):113-117: 2002. doi:10.1511/2002.2.113.

Richard E. Korf. Multi-Way Number Partitioning. *In Proceedings of IJCAI'09 the 21st International Joint Conference on Artifical Intelligence*. 538-543, 2009.

Timo Mantere and Jarmo T. Alander. Evolutionary software engineering, a review. *Applied Soft Computing*, 5(3):325-331, 2005. doi:10.1016/j.asoc.2004.08.004

M. Saqib Nawaz , M. IkramUllah Lali and M. A. Pasha. Formal verification of crossover operator in Genetic Algorithms using Prototype Verification System (PVS), *In Proceedings of the 2013 IEEE International Conference on Emerging Technologies (ICET)*, 2013. doi:10.1109/ICET.2013.6743532

S. Owre, J. M. Rushby and N. Shankar. PVS: A Prototype Verification System. *11th International Conference on Automated Deduction (CADE)*, Lecture Notes in Artificial Intelligence, Springer-Verlag, 1992. ISBN 978-3-540-55602-2.

Terasic. *SoCKit User Manual*, 2015.

Bruce Wile, John Goss and Wolfgang Roesner. *Comprehensive Functional Verification: The Complete Industry Cycle (Systems on Silicon)*. Morgan Kaufmann Publishers Inc, 2005. San Francisco, CA, USA.

Xillybus. *Xillybus IP cores and design services*, 2017. <http://xillybus.com/>.

Xillybus. *Licensing*, 2017. <http://xillybus.com/licensing>