Genetic Programming in FPGA Implementation of Addition as a Part of the Convolution

Ernest Jamro, Kazimierz Wiatr

AGH Technical University, Institute of Electronics

Mickiewicza 30, 30-059 Kraków, Poland

tel. +48 12 6173033, fax +48 12 6332398, email: jamro / wiatr@uci.agh.edu.pl

Abstract

In FPGAs, an addition should be carried out in the standard way employing ripple-carry adders (rather than carry-save adders), which complicates search for an optimal adder structure as routing order has a substantial influence on the addition cost. Further, complex parameters of inputs to the adder block have been considered e.g. correlation between inputs. These parameters are specified in different ways for different convoler architectures. Consequently optimisation of the adder tree is a key issue addressed in this paper. Simulated Annealing and Genetic Programming have been proposed, and obtained results compared with the Greedy Algorithm (GrA) and the Exhaustive Search (ES). As a result, the GrA is the best solution when computation time is of great importance. Otherwise, the Simulated Annealing should be employed for the number of addition inputs $N>8$, and the ES is recommended for $N\leq 8$. Employing the Simulated Annealing gives about 10-20% area reduction in comparison to the GrA.

Topics: *image processors, processing arrays and FPGAs, reconfigurable structures*

1 Introduction

Field Programmable Gate Arrays (FPGAs) incorporate dedicated ripple-carry logic, which makes Ripple Carry Adders (RCA) the best solution for FPGAs [1]. For convolution (FIR filters) and other similar operations, addition is a fundamental operation, therefore optimising network of adder tree significantly influences the circuit area and performance. Up to the authors knowledge little research has been done in this area.

For FPGAs different multiplier architectures are recommended for different design options, e.g. Look-Up-Table based Multiplier or Multiplierless Multiplier [12,13]. In order to optimise the adder block independently from the multiplier architecture, complicated parameters of inputs to the adder block have been defined [2], such as input range (not only input width), input shift, kind of operation (addition/subtraction), and correlation between inputs.

Optimisation of the adder tree can be accomplished employing different techniques. A Greedy Algorithm (GrA) has been proposed by [2]. The GrA determines quickly the network of adders, it is the least computationally demanding algorithm from all techniques considered in this paper. Nevertheless, better results can be obtained by employing more sophisticated techniques described in this paper.

The best result can be always found by checking all possible solutions. Nevertheless, the optimisation of the adder block proved to be NP-complete [2], and therefore the Exhaustive Search (ES) can be efficiently implemented only for the number of inputs to the adder block $N \leq 8$, which requires analysing up to 315 combinations. For example, increasing N to 12 results that 467 775 solutions have to be searched through. Consequently more sophisticated techniques are proposed to solve the problem.

In this paper optimisation of the adder tree employing Simulated Annealing (SA) and Genetic Programming (GP) is studied. Different methods for generating new structures of the adder tree for the SA and different crossover and mutation operations for GP are presented. Finally, implementation results for the SA and GP, and the GrA and ES described in [2] are given.

2 Simulated Annealing (SA)

2.1 Principle

The principle behind the SA [3, 4] is analogous to what happens when metals are cooled at a controlled rate. The slowly falling temperature allows atoms in the molten metal to line themselves up and form a regular crystalline structure that has high density and low energy. In the SA, the value of an objective function which we want to minimise, is analogous to the energy in a thermodynamic system. At high temperatures, SA allows function evaluations at faraway points and it is likely to accept a new point at higher energy. At low temperatures, SA evaluates the objective function only at local points and the likelihood of it accepting a new point with higher energy is much lower.

The SA algorithm, implemented for optimising adders structures, employs the following steps:
Objective function calculates the cost C of the circuit for the given adders tree.

Annealing Schedule regulates how rapidly the temperature, T, goes from high to low values, as a function of iteration counts. In our case, the starting temperature T_1 equals the cost of a 2-bit wide adder C_{A2}, the stopping temperature T_S equals ¼ of the cost of a 1-bit wide adder $C_{A1}/4$. In every iteration, the temperature T_i is decreased according to the following equation:

$$T_{i+1} = \eta \cdot T_i \qquad (1)$$

where $\eta = \left(\frac{T_1}{T_S}\right)^{1/S}$, *S- the number of iterations.*

Generating a new adders structure is obtained by randomly selecting two adders on the same layer; i.e. randomly selecting a first adder (input) from all adders and randomly selecting a second adder from adders at the same layer as the first adder. Examples of possible modification are given in Figure 1.

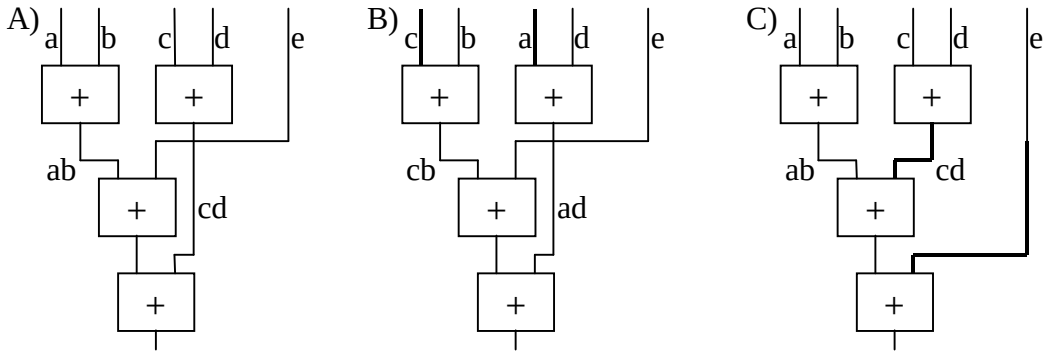

Figure 1. Examples of possible one-step modification: A) an initial circuit, B) C) the modified circuit A.

Modifications of the circuit are constrained by the temperature T_i. In the conventional SA, also known as the Boltzmann machine, the generating function which specifies the change of the input vector, is a Gaussian probability density function [5]. In our approach, the number of possible solutions is finite therefore the Gaussian probability function is useless. An alternative solution is defining a move set [5], denoted by M(x), as a set of legal points available for exploration. However, constructing the move set is rather computationally demanding task thus not implemented. In our approach, therefore, two adders are selected randomly (but at the same adders layer) and then a local acceptance function (LAF), which is further described in the next paragraph, is calculated. The local acceptance function differs from the (global) acceptance function as it takes under consideration only the cost of the two involved adders before and after modification. If modification is not accepted locally, the change is rejected and the next modification is randomly generated (the iteration counter and temperature are not affected in this step).

Acceptance function. After a new network of adders has been evaluated, the SA decides whether to accept or reject it based on the value of an acceptance function *h( )*. The acceptance function is the Boltzmann probability distribution:

$$h(\Delta C, T) = \frac{1}{1 + \exp(\Delta C / T)} \qquad (2)$$

where: $\Delta C = C_{i+1} - C_i$ *- the difference of the adders cost for the previous and current adders tree.*

The new circuit is accepted with probability equal the value of the acceptance function.

2.2 Implementation results

The result for the SA, for different circuits are given in Table 1. It can be seen that for the filter *a*, the result is *103*, the best possible – the same as for the ES, is obtained already for 1000 iterations. For the filter *c*, the cost equals *123*, the same as for the ES, was obtained already for 30k iterations. It should be, however, noted that the computation cost of a single iteration is lower for the ES than for the SA. This holds as for the ES, the change in the circuit is well-defined and usually constrained

only to the upper layer of the adder block and therefore only a part of the circuit has usually to be re-calculated. For the SA, the change is done randomly and on any part of the circuit, therefore cost of the whole circuit has to be calculated again. The lower calculation cost for the ES does not, however, compensate much greater number of iterations required to obtain the same result. Consequently, the overall calculation cost of the ES is usually greater than for the SA, however for small circuits for which the calculation cost is very low, the ES is a good alternative to the SA.

Filter	GrA	SA 1k	SA 30k	SA 1M	ES
a	111	93±0	93±0	93±0	93
c	128	126.9 ±0.3	125 ±1.4	123 ±0	123
d	413	394 ±3	385 ±1	382 ±1	-
d (wlaf)	413	398 ±4	382 ±1	380 ±1	-
e	1358	1346 ±10	1299 ±3	1292 ±4	-
e (wlaf)	1358	1341 ±9	1293 ±4	1283 ±4	-
f	3730	3702 ±20	3338 ±14	3245 ±6	-
f (wlaf)	3730	3706 ±13	3419 ±13	3296 ±8	-

Table 1. The circuit costs for the GrA, ES and different number of iterations for the SA for different filters; wlaf – without local acceptance function.

In our solution, the final circuit (obtained in the lowest temperature) is often not the best one. Therefore, the best-obtained circuit is every time stored as the result; this increases calculation cost insignificantly but allows for substantial savings.

Table 1 shows also the results when local acceptance function (LAF) is not implemented (option: wlaf). Calculating local cost before and after the modification, insignificantly influences the total calculation cost and the LAF usually rejects solutions which are unlikely to generate a good global result. Conversely, the LAF constrains search space and therefore may cause some good solutions to be omitted. This is often the case for relatively small adders circuits and for large iteration numbers. For example, it can be seen in Table 1 that not implementing LAF gives better results for the circuit *d* and for small number of iterations, however spoils results for more complicated circuit *f*.

3 Genetic Programming (GP)

Genetic Programming (GP) [6, 7, 8] is an optimisation method based loosely on the concept of natural selection and evolutionary process. Major components of the GP include: encoding scheme, fitness evaluation, parent selection, crossover operation and mutation operators, these are approached next.

Encoding scheme transforms gene representation into the problem specific representation. In this approach, the adder tree is represented directly using two vectors of integers. Each adder occupies one entry in each vector. The entry specifies the index of the adder or input (from the lower layer) which is connected to the considered adder. For example, the parent 0 in Figure 2 is represented in the following two vectors of integers:

considered adder	*24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13*
vector 0	*22, 19, 18, 13, 14, 16, 0, 3, 2, 4, 9, 7*
vector 1	*23, 20, 21, 17, 15, 11, 6, 8, 1, 5, 12, 10*

It should be noted that initially it might seem that the structure of the adder can be defined giving only the order of adder block inputs (the bottom layer order), as the rest of the structure can be built straightforward by connecting two neighbour adders. This is a special case when the number of inputs to the adder block is a power of two. Otherwise, there is an alone signal which cannot be paired and therefore must be fed directly (without addition) to the next layer of the adder block. In the given example for the parent 0, it is the case for the input 11 for the first layer and signal 18 for the second layer. These alone signals complicate the adder structure and cause that the structure of upper layer adders must be also included into gene coding.

Fitness evaluation is based on evaluation of the cost (area) of a given adder.

Selection - After fitness evaluation, a new generation is produced from the current generation. The selection operation determines which adder will survive, based on the fitness value – the lowest cost of the adder, the greatest survival probability. It our approach the modGA algorithm [6] has been implemented as it has proved to surpass the classical genetic algorithm [6]. In the modGA, in every generation we select independently *(p-r)* chromosomes to survive unchanged with the probability proportional to the scaled fitness f_i' which is obtained as a linear scaling of the area f_i occupied by the adder block:

$$f_i' = a \cdot f_i + b. \tag{3}$$

Parameters *a* and *b* are calculated independently for every generation to satisfy the following equations:

$$a \cdot f_{\min} + b = 1 \tag{4}$$

$$\sum_{i=1}^{p} (a \cdot f_i + b) = p - r \tag{5}$$

where: f_{min} *– the fittest (minimum cost) chromosome, p –population size, r – number of chromosomes determined to die* $r<p/2$*;*

The eq. 4 preserves the fittest individual with the probability equal *1*. Eq. 5 causes that on average *(p-r)* chromosomes are selected to survive in a single wheel spin (a single wheel spin - every chromosome is picked to survive with probability f_i' only once). In our approach, the wheel spins until *(p-r)* chromosomes are selected and on average, a single wheel spin is required.

The *r* chromosomes selected to die are replaced by new ones, which are produced in either crossover or mutation. Consequently, the following equation holds:

$$r = c + m \tag{6}$$

where: c- number of new chromosomes produced in the crossover operation, m- number of new chromosomes produced in the non-overwriting mutation operation (see mutation operation). In our approach p= 12, r= 5, c= 4, m= 1.

Crossover is applied to randomly selected pairs of parents. The structure of the adders tree seems very similar to the commonly used tree graph structure used for scheduling and partitioning [6] or finding the optimal operation tree [9]. However for the addition, the structure of the tree is strongly constrained. Therefore, the crossover operation implements a procedure, which generates only a valid structure of the adder tree (no repair procedure is required).

Two different options of crossover have been implemented. The first one (option A) attempts to copy as much as possible from the parents (considering the actual parents structure and disregarding the indices) and then employs a greedy algorithm (simplified version of the GrA implemented in [2]) to route unconnected adders. In the second option (option B), the offspring copies the structure of the first parent and implements changes similar as for the SA, however changes are applied according to the structure of the second parent. In this option only indices are considered. These options are described below.

3.1 Option A

In this option, an offspring inherits one (or all but the one) branch of adders from the first parent. For the example given in Figure 2, the offspring 0 inherits from the parent 0, the adder structure: 20, 14, 15, 9, 12, 4, 5. Then, the offspring inherits as much as possible from the second parent. In the given example, the offspring 0 can copy only adders 13, 2, 11; 16, 6, 7 and 17, 8, 0 from the parent 1. Unfortunately, the whole adder structure may not be obtained directly from the parents; as some connections copied from the first parent, conflict with connections in the second parent. For the given example for the offspring 0; from the parent 0, the adder structure: 14, 9, 12 is copied; this make impossible to copy the adder structure 14, 1, 9 from the parent 1 as the input 9 is already connected.

It should be noted that the crossover algorithm considers only indices of inputs on the bottom layer (in the given example: inputs 0-12) and how they are connected going up to the top layer (the indices of the adders of the upper layer are disregarded). Consequently, pairing of the upper layer adders can be achieved provided that all adders on the lower layers can also be paired. Therefore, a single connection that cannot be achieved on the bottom layer, prevents the adders on the upper layers from being connected. This causes that the structure of adders on the upper layer is seldom inherited from the parents. Nevertheless, the offspring inherits only connections which existed in either of the parents.

This approach causes that the effectiveness of the algorithm strongly depends on crossover points; that is how many adders are copied from the first parent. Consequently, a crossover parameter is included into the gene-coding scheme. This parameter is optimised together with the adders structure during the evolutionary process. The crossover parameter defines from which adders layer a randomly chosen branch of adders is copied from the first parent to the offspring. For example, on Figure 2, offspring 0 inherits adder structure: 20, 14, 15, 9, 12, 4, 5; i.e. the branch of adders beginning from the adder 20 (adder 20 is on the layer 2). Besides, crossover parameter defines if one branch of the adder is inherited or all but one branches are inherited. For the given example, for the offspring 0, only one branch of adders is inherited from the parent 0; for the offspring 1, all but the one branches of adders are inherited from the parent 1, i.e. except adders: 20, 14, 1, 9, 3.

The implementation results shown that in old generations, the crossover parameter is the same for all chromosomes and is equal: copy one branch beginning from the next to the last adder, i.e. copy half of the adder structure. Consequently, the crossover parameter is not longer included into gene-coding scheme and the crossover point is fixed to the half-adder copping.

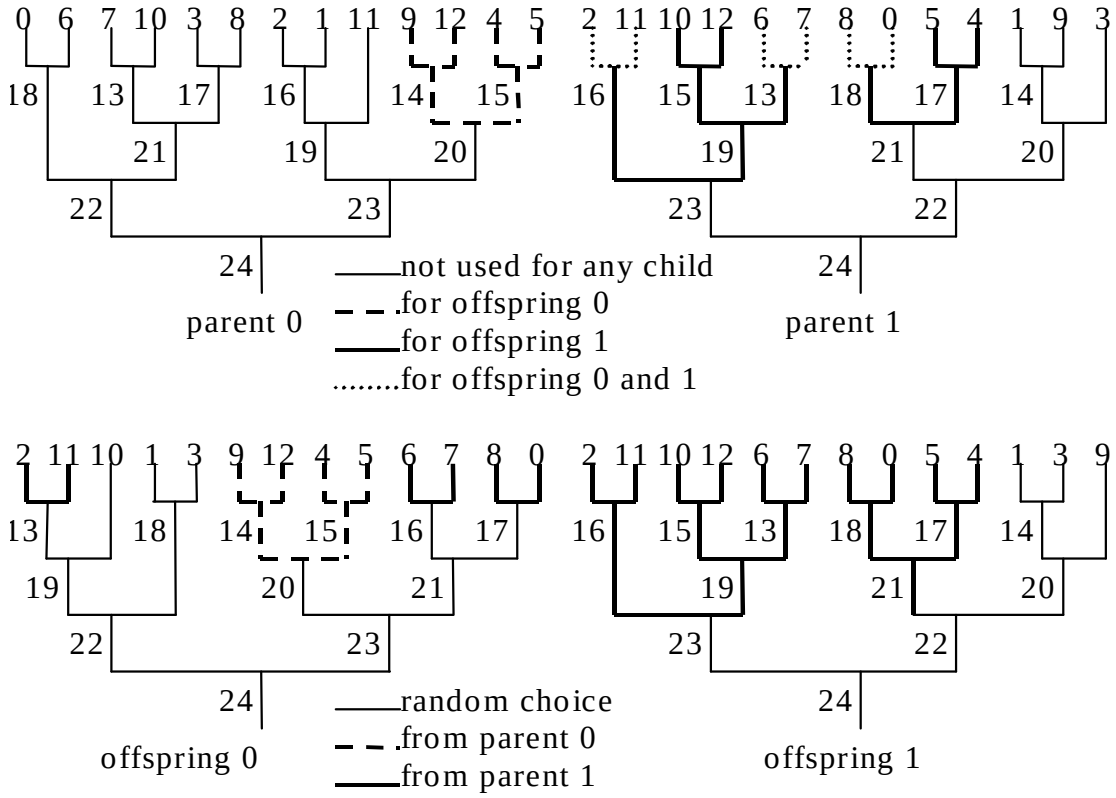

Figure 2. An example of the crossover operation for Option A.

3.2 Option B

In this option, the crossover operation is similar as for the Simulated Annealing presented in the previous section. The difference is that for the SA a modification is made in a random way, however for the GP, the modification is carried out with respect to the structure of the second parent. An example of the crossover operation is given in Figure 3. The crossover operation consists of three steps:

1. Randomly selecting a common crossover signal (in the given example: signal 0).
2. Finding swapped signals which are paired with the common signal (signal 1 for parent 0 and signal 2 for parent 1). In the case when the common signal is an alone signal (is not paired), the alone signal is chosen.
3. Swapping signals found in the previous point.

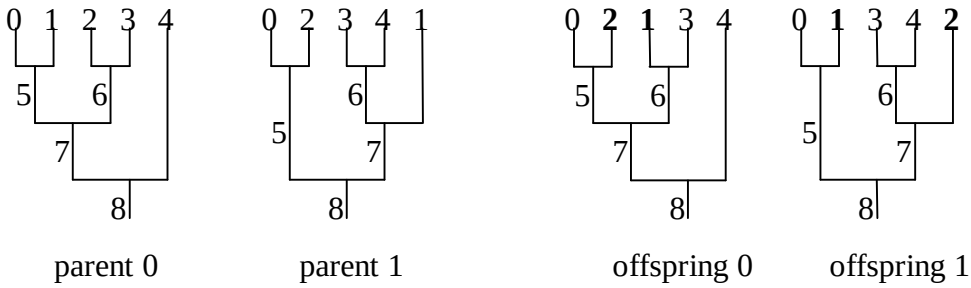

Figure 3. An example of a crossover operation for Option B.

It should be noted that the common crossover signal can be selected on any layer of the adder (except the top layer, which is a trivial case and therefore skipped). Besides, indices of signals on the upper layers for different parents may not correspond to each other, in the sense the real adders structure. For example, in Figure 3, signal 5 in the parent 0 ($S_5 = S_0 + S_1$) is different from the signal 5 in the parent 1 ($S_5 = S_0 + S_2$). This means that swapping the upper layer adders often disregards the real connections of the parents as indices of these adders are assigned more or less in a random way. Therefore, to improve the algorithm the indices of the upper layer adders are assigned (sorted) according to the increase of the input index (the lower index of two inputs). For example, for the parent 1 in Figure 3, the adder 5 has the lowest index on the layer 1 because the input 0 is the lowest input index on the bottom layer. Sorting adder indices improves correlation between parents, nevertheless, the index of the upper layer adder in one parent often represents different addition than in the second parent. This means that swapping is often achieved in a random way, especially when structure of parents differs significantly. It should be noted that for large adder structure the relationship between index number and its structure is decreasing, therefore for large adders tree this crossover method is not recommended (see Table 3).

The change made by a single swapping is rather insignificant therefore, usually 1-3 similar swapping operations are performed to obtain an offspring.

The idea behind the modGA is that the algorithm avoids leaving the exact copies of the same chromosomes in the new population, which may still happen accidentally by other means but is very unlikely [6]. However, experiments proved that both Option A and B can produce an offspring identical to its parents especially if the parents are very similar. This causes that several copies of the same parents exist in the population and therefore the modGA algorithm deteriorates the result. Consequently, to improve the modGA results in the case when an offspring is an exact copy of the parent(s) (or differs insignificantly), the mutation is performed on the offspring. Therefore, in this approach, the additional mutation operation prevents from obtaining the exact copy of the parent during crossover operation and prohibits super-individuals from dominating the population. It should be noted that in the nature, a specimen avoids to mate with its relatives in order not to produce similar gene offspring. Moreover, similar solution has been proposed by Maudlin [10], where the mutation rate is changed according to the degree of homogeneity of the chromosomes. The disadvantage of Maudlin's approach is that it requires additional computation time to evaluate the degree of the homogeneity. In our approach, however, detecting crossover diversity increases computation time insignificantly as it is associated directly with the crossover operation.

3.3 Mutation

Crossover operation can only explore the current gene potential therefore, a mutation operation is included to spontaneously generate new chromosomes. In our approach, mutation is carried out in a similar way as for the SA, i.e. by swapping two adders on the same layer.

Two different mutation options has been implemented:

1. parent non-overwriting mutation (NOM)
2. parent overwriting mutation (OM)

The NOM is associated with the modGA selection operation as the number of new chromosomes r generated in each generation, includes the number of new chromosomes created during mutation m. Therefore, randomly picked chromosomes (from the surviving chromosomes) are copied and the mutation is performed on the copy of the chromosomes.

The OM is carried out in the standard way, i.e. every unchanged chromosome is mutated with probability $p_{m.}$

Two different mutation options have been implemented to allow proper development of the population. In the case when only the OM is implemented, the high mutation ratio prohibits super-individuals to grow as often probability of generating an offspring which fitness is comparable to the parent is very low – lower than mutation rate. Therefore, the best solution is often generated rather in a random way then based on the genetic algorithm properties and the fitness of the latest generations is very often far from the best solution fitness. Conversely, low mutation ratio causes mutation to have insignificant influence on the result and therefore deteriorates the result. Employing only NOM causes that super-individuals are always copied to the new generation without any change (the fittest chromosome is selected) and therefore the population is dominated by the super-individuals which may be in a local minimum. Consequently, the best solution is a combination of the NOM and OM. Implementation results, given in Table 2, confirm this assumption.

filter	p_m= 0, m= 1	p_m= 0.2%, m= 1	p_m= 10%, m= 0
d) iter= 6k	393 ±2	392 ±5	391 ±2
d) iter= 200k	392 ±4	388 ±5	381 ±1
e) iter= 6k	1302 ±3	1303 ±2	1317 ±4
e) iter= 200k	1297 ±3	1292 ±2	1297 ±2
f) iter= 6k	3580 ±7	3580 ±6	3616 ±8
f) iter= 200k	3454 ±5	3455 ±4	3508 ±16

Table 2. Implementation results for different mutation solutions: only NOM, combination of the NOM and OM, and only OM; for crossover: option A.

3.4 Implementation results

Table 3 shows implementation results for the different algorithms. The number of iterations for the GP and SA is chosen so that the calculation cost was roughly the same. It can be seen from Table 3 that the SA solution gives usually the best results and the crossover Option A is a better solution in comparison to Option B, especially for more complicated circuits. Furthermore for the Option B and filter f, the implementation results are even so poor that the GrA initial solution is the best-found solution for up to 6000 iterations.

Filter, technique	# iterations (GP/SA)		
	200/1k	6k/30k	200k/1M
d) GP Option A	399 ±3	392 ±5	388 ±5
d) GP Option B	412 ±2	392 ±4	387 ±5
d) SA	394 ±3	385 ±1	382 ±1
e) GP Option A	1341 ±6	1303 ±2	1292 ±2
e) GP Option B	1358 ±0	1357 ±1	1297 ±4
e) SA	1346 ±10	1299 ±3	1292 ±4
f) GP Option A	3713 ±7	3580 ±6	3455 ±4
f) GP Option B	3730 ±0	3730 ±0	3645 ±26
f) SA	3702 ±20	3338 ±14	3245 ±6

Table 3. Implementation results for different options of the GP and SA, for different number of iterations.

4 Conclusions

The Simulated Annealing and Genetic Programming have been employed to optimise the adders tree. These techniques proved to be good alternatives to the Greedy Algorithm and Exhaustive Search proposed in [2]. The structure of the adder block is strictly defined. Therefore for the GP, the crossover procedure has to copy parts of the parents in such a way that the child has a proper structure. This, however, is difficult to be achieved and therefore some part of the offspring has to be routed to satisfy the adder block constrains rather than to copy a structure of the parents. Two

different crossover procedures have been implemented as a part of the research. Nevertheless, at the same computation time, the GP usually gives worse results than the SA. This conclusion is similar as presented by McMahon [11] for scheduling problems and shows that for some problems the SA is beneficial in comparison to the GP.

Addition block is a part of the convolution and therefore all procedures, described in this paper, are included in the Automated Tool for Convolution in FPGAs (AuToCon). The AuToCon [12, 13, 14] generates an optimised VHDL code of the convoler for the given coefficient values.

References

[1] Xing S., Yu W.W.H., *FPGA Adders: Performance Evaluation and Optimal Design*, IEEE Design & Test of Computers, pp. 24-29, Jan.-Mar. 1998.

[2] Jamro E., Wiatr K., *FPGA Implementation of Addition as a part of the convolution*, Proc. of the IEEE Int. Conf. Digital System Design, Warszawa, Poland, 4-6 Sep 2001.

[3] Aarts, E.H., Korst, J. *Simulated Annealing and Boltzman Machines*, Wiley, Chichester, UK, 1989

[4] Kirkpatrick, S. Gelatt, C.D., Vecchi, M.P. *Optimisation by simulated Annealing*, Science, 220 (4598): 671-680, May 1983.

[5] Jang J.S.R., Sun C.T., Mizutani E., *Neuro-Fuzzy and Soft Computing*, Prentice-Hall, London, UK, 1997.

[6] Koza, J.R. Genetic Programming: On the Programming of Computers by Means of Natural Selection. Cambridge, MA: The MIT Press, 1992.

[7] Michalewicz Z. *Genetic Algorithm + Data Structure = Evolutions Programs*, Spinger-Verlag, Berlin, 1992.

[8] Goldberg D.E. *Genetic Algorithms in Search Optimisation & Machine Learning*, Addison-Wesley, Massachusetts, 1989.

[9] Aytekin T., Korkmaz E.E. Guvernir H.A. *An Application of genetic Programming to the 4-Op Problem using Map-Trees*, pp.28-40 in Xin Yao *Progress in Evolutionary Computation, Selected Papers on AI'93 nad AI'94 Workshops on Evolutionary Computation*, Springer, Berlin, 1995.

[10] Maudlin, M.L. Maintaining Diversity in Genetic Search, AAAI Proc. National Conference on Artificial Inteligence, 1984, pp. 247-250.

[11] McMahon G. Hadinoto D. *Comparison of Heuristic Search Algorithms for Single Machine Scheduling Problems*, pp. 293-304 in Xin Yao *Progress in Evolutionary Computation, Selected Papers on AI'93 nad AI'94 Workshops on Evolutionary Computation*, Springer, Berlin, 1995.

[12] Wiatr K., Jamro E. Constant Coefficient Multiplication in FPGA Structures, Proceedings of the 26th Euromicro Conference, Maastricht, The Netherlands, Sep. 5-7, 2000, Vol. I, pp. 252-259.

[13] Wiatr K. Jamro E. *Implementation of Multipliers in FPGA Structures*, ISQED March 2001 San Jose, California

[14] Wiatr K. Jamro E. *Implementation of image data convolutions operations in FPGA reconfigurable structures for real-time vision systems*. International IEEE Conference on Information Technology: Coding and Computing, Nevada 2000, pp. 152-157.