

UNIVERSITY OF PORTO

**Reconfigurable Custom Computing for
Population-Based Optimization
Metaheuristics: Accelerating Genetic
Algorithms with FPGAs**

by

Pedro Manuel Vieira dos Santos

A dissertation submitted to the
Faculty of Engineering of the University of Porto
in accordance with the requirements for the degree of
Doctor in Electrical and Computer Engineering

September 2015

Thesis Identification

Title:	Reconfigurable Custom Computing for Population-Based Optimization Metaheuristics: Accelerating Genetic Algorithms with FPGAs
Keywords:	Genetic Algorithms, Metaheuristic, Reconfigurable Computing, FPGA, High-Level Synthesis
Start:	March 2008
Candidate	
Name:	Pedro Manuel Vieira dos Santos
e-Mail:	pedro.vieira.santos@fe.up.pt
Supervisor	
Name:	José Carlos dos Santos Alves
e-Mail:	jca@fe.up.pt
Co-Supervisor	
Name:	João Paulo de Castro Canas Ferreira
e-Mail:	jcf@fe.up.pt
Educational Establishment:	FEUP - Faculty of Engineering of the University of Porto
Research Institution:	INESC TEC - INESC Technology and Science

Statement of Originality

The work presented in this thesis was carried out by the candidate. It has not been presented previously for any degree, nor is it at present under consideration by any other degree awarding body.

Candidate:

(Pedro Manuel Vieira dos Santos)

Supervisors:

(José Carlos dos Santos Alves)

(João Paulo de Castro Canas Ferreira)

Statement of Availability

I hereby give consent for my thesis, if accepted, to be available for photocopying and for interlibrary loan, and for the title and summary to be made available to outside organizations.

Candidate:

(Pedro Manuel Vieira dos Santos)

*“Everything has been thought of before,
but the problem is to think of it again.”*

Johann Wolfgang von Goethe

Acknowledgements

This work could not have been completed without the help and support of many people to whom I would like to thank.

I wish to express my gratitude to my supervisors, Professor José Carlos Alves and Professor João Canas Ferreira, for giving me the opportunity to perform my research work. Their guidance, valuable advices, and hours of thoughtful discussions were undoubtedly a driving source towards the completion of this work.

I would like to thank to Professor João Cardoso and Professor Michael Hübner for the opportunity to work during 2 months at the Institute for Information Processing Technologies in the Karlsruhe Institute of Technology (Germany). I thank to the colleagues at the institute for the fruitful discussions that we have had, and to Ali Azarian for his friendship during this stay.

This work was partially funded by the PhD grant SFRH/BD/41259/2007 awarded by the Portuguese Foundation for Science and Technology (FCT - Fundação para a Ciência e a Tecnologia), to whom I would like to thank for the support.

I also would like to thank to Instituto de Engenharia de Sistemas e Computadores, Tecnologia e Ciência (INESC TEC) and to Faculty of Engineering of the University of Porto (FEUP) for providing me the necessary facilities to my research work.

I thank to my friends and colleagues at FEUP and INESC TEC, especially to Pedro Mota, João Oliveira, Mário Pereira, Luís Pessoa and Paulo Ferreira, for their friendship and continuous encouragement during this work.

Last but certainly not least, I am deeply grateful to my wife, Liliana, for her endless support and comprehension during this long journey. Thank you for always believing in me.

Abstract

This thesis addresses the study of mechanisms to accelerate population-based meta-heuristic search procedures with custom computing machines, in particular genetic algorithms (GAs), implemented in field-programmable gate arrays (FPGAs). This meta-heuristic often requires long execution times to converge to satisfactory results, thus limiting its use in systems where the processing power is limited. To address this issue, a scalable processor array architecture is proposed where the operations of the meta-heuristic are parallelized to improve the execution speed of GAs while ensuring that the quality of the optimization process is not compromised. To customize the architecture to solve different optimization problems, a design flow based on high-level synthesis (HLS) is developed to specify the problem-dependent operations of the algorithm.

A hardware architecture is proposed that effectively accelerates the execution of cellular GAs (cGAs), a variant of a GA where the candidate solutions of the problem (population) are distributed over a regular grid. With such physical distribution, several independent memories can hold subsets of the population so that they are accessed by processing elements (PEs) that compute locally a GA with those solutions. As a result, the architecture can be scaled by changing the number of PEs and memories, without introducing a degradation in performance with the size of the array due to memory access bottlenecks. This architecture is suitable for FPGA implementation, since these devices have a large number of memory blocks to implement the required distributed memory system.

Although GAs are usually known for using a few simple operators, in many problems special constraints apply and dedicated algorithms are required to encode feasible solutions. Therefore, a HLS-based design flow is presented that specifies the problem-specific operations of the algorithm to customize the architecture for the specific requirements of each problem. Additionally, a complete scalable array solution, called cGA processor (cGAP), is presented that implements all the infrastructure necessary to the execution of the algorithm. As a result, the cGAP is specified by defining a set of parameters, while the problem-specific operations are specified in a common programming language (C++) and translated to digital hardware with HLS tools.

Finally, two optimization problems that appear in the context of wireless ad hoc networks are addressed as case studies to demonstrate the effectiveness of the cGAP and its design methodology. These problems are mapped to the proposed processor and implemented in a Virtex-6 FPGA. Results have shown that the acceleration achieved by the cGAP is near to directly proportional to the number of PEs, and that increasing the parallelism level ensures similar (or even better) solution quality. Therefore, the cGA supported by the architecture is efficient as it can improve the execution time of the algorithm and, at the same time, effective as it provides good quality solutions. Additionally, the scalability of the architecture allows exploiting trade-offs between acceleration of the algorithm and hardware resources used to implement it.

Sumário

Esta tese aborda o estudo de mecanismos para acelerar procedimentos de pesquisa meta-heurística baseados em populações, em particular algoritmos genéticos (GAs), usando processadores dedicados implementados em sistemas digitais reconfiguráveis baseados em FPGA. Esta metaheurística requer normalmente tempos de execução longos para atingir bons resultados, o que limita a sua aplicabilidade em sistemas com capacidade de processamento limitada. Em resposta a este problema, é proposta uma arquitetura escalável constituída por uma matriz de processadores idênticos, onde as operações da metaheurística são paralelizadas sem comprometer a eficácia do processo de otimização. Para adaptar a arquitetura de forma a que esta seja aplicada a diferentes problemas de otimização, é proposto um fluxo de projeto baseado em síntese de alto nível (HLS) onde as operações dependentes do problema são especificadas com recurso a uma linguagem de programação convencional.

A arquitetura de processamento proposta acelera a execução de GAs celulares (cGAs), uma variante da metaheurística onde as soluções do problema em evolução (população) são distribuídas de forma regular numa grelha regular abstrata. Tal distribuição possibilita que conjuntos de soluções possam ser mantidos em memórias independentes de forma a alimentar em paralelo vários elementos de processamento (PEs), em que cada um executa um GA com um conjunto local de soluções. Como resultado, a arquitetura é escalável através da alteração do número de PEs e memórias associadas, sem que isso se traduza numa degradação de desempenho devida a estrangulamentos nos acessos ao sistema de memória. Esta arquitetura adequa-se a uma implementação em dispositivos FPGAs, uma vez que estes apresentam um grande número de blocos de memória, conveniente para implementar o sistema de memória distribuída.

Apesar dos GAs serem conhecidos por usarem poucos e simples operadores, muitos problemas apresentam restrições específicas que resultam na necessidade de algoritmos dedicados para codificar e transformar soluções admissíveis. Desta forma, é proposto um fluxo de projeto baseado em síntese de alto nível que permite especificar as operações e procedimentos dependentes do problema numa linguagem de programação padrão (C++). Para além disto, é também apresentado o mecanismo de construção e parametrização da matriz de nós de processamento e de toda a infraestrutura de suporte à execução do cGA.

Para validar a solução proposta e a metodologia de projeto, são abordados como casos de estudo dois problemas de otimização que surgem no contexto de redes sem fio *ad hoc*. Os dois problemas são implementados num dispositivo FPGA Virtex-6 e os resultados demonstraram que a aceleração obtida pela arquitetura é diretamente proporcional ao número de PEs, e que o aumento de paralelismo não degrada a qualidade da solução. Mostra-se assim que a arquitetura paralela proposta é eficiente uma vez que permite acelerar a execução do algoritmo e, ao mesmo tempo, é efetiva uma vez que garante soluções de boa qualidade. Para além disso, a escalabilidade da arquitetura permite explorar o compromisso entre aceleração do algoritmo e recursos lógicos usados para o implementar.

Contents

1	Introduction	1
1.1	Thesis organization	4
1.2	Contributions	5
2	Background and state of the art	9
2.1	Introduction	9
2.2	Genetic Algorithms	9
2.2.1	Canonical genetic algorithm	10
2.2.1.1	Representation of solutions	11
2.2.1.2	Selection	13
2.2.1.3	Crossover	14
2.2.1.4	Mutation	15
2.2.1.5	Replacement	16
2.2.1.6	Fitness evaluation	16
2.2.1.7	Example: OneMax problem	16
2.2.2	Decentralized GAs	17
2.3	Hardware implementations of GAs	19
2.3.1	GA architectures	25
2.3.1.1	Panmictic - generational	25
2.3.1.2	Panmictic - steady-state	27
2.3.1.3	Distributed	28
2.3.1.4	Cellular	31
2.3.1.5	Variants of GAs	31
2.3.2	Where is the bottleneck?	32
2.3.3	Acceleration	34
2.3.4	General considerations	35
2.4	Summary	36
3	A scalable processor array for cGAs acceleration	37
3.1	Introduction	37
3.2	The architecture	38
3.2.1	Comparison with a canonical cGA	43
3.2.2	Application to other population-based metaheuristics	47
3.3	Architecture simulation	49
3.3.1	Toroidal arrays configuration	53
3.3.1.1	Square arrays	53
3.3.1.2	Non-square arrays	56

3.3.2	Non-toroidal arrays configuration	59
3.3.2.1	Square arrays	60
3.3.2.2	Non-square arrays	61
3.4	Hardware implementation: the TSP	63
3.4.1	Processing element	63
3.4.2	Memory access control	66
3.4.3	Implementation and results	67
3.5	Summary	71
4	The cGAP architecture	73
4.1	Introduction	73
4.2	cGA processor (cGAP) overview	74
4.3	cGA array (cGAA)	75
4.4	cGA cell	77
4.4.1	Processing element (PE)	77
4.4.2	Subpopulation memory	78
4.5	Control infrastructure	82
4.5.1	cGA controller (cGAC)	82
4.5.2	Communication infrastructure	83
4.5.3	cGAP interface	87
4.6	RNG infrastructure	88
4.6.1	RNG block	91
4.7	Summary	92
5	The cGAP design methodology	95
5.1	Introduction	95
5.2	Specification for high-level synthesis	95
5.2.1	Processing element	96
5.2.1.1	Algorithm structure	96
5.2.1.2	Interface	98
5.2.1.3	Access arbitration to the subpopulation memory	101
5.2.2	cGA controller	106
5.2.2.1	Algorithm structure	106
5.2.2.2	Interface	107
5.3	cGAP host communication	109
5.3.1	The host interface	109
5.3.2	Software access to cGAP	110
5.4	Design Flow	111
5.4.1	cGAP parameters configuration	112
5.4.2	Hardware	114
5.4.2.1	High-level synthesis	114
5.4.2.2	RTL synthesis	115
5.4.2.3	FPGA implementation	115
5.4.2.4	Verification	116
5.4.3	Software	116
5.4.4	Hardware platform	116
5.5	Summary	118

6	Experimental results	121
6.1	Introduction	121
6.2	Spectrum allocation in cognitive radios	122
6.2.1	Problem definition	122
6.2.2	The cGA operations and control	125
6.2.3	The processing element	126
6.2.3.1	Subpopulation memory organization	127
6.2.3.2	Coding in C++ for HLS	129
6.2.3.3	HLS optimizations	131
6.2.4	cGAP implementation	135
6.2.5	cGAP results	139
6.3	Minimum energy broadcast	145
6.3.1	Problem definition	146
6.3.2	A memetic algorithm for the MEB problem	147
6.3.2.1	Codification of solutions	148
6.3.2.2	Local search heuristic: <i>r</i> -shrink	149
6.3.2.3	The cGA operations	153
6.3.3	cGAP implementation	154
6.3.4	cGAP results	158
6.4	Considerations on the cGAP implementation	163
6.5	Summary	164
7	Concluding remarks	167
7.1	Recommendations for future work	172
	References	185
A	C++ classes to describe the cGAP	187
A.1	Class <code>command_type_cGA</code>	187
A.2	Class <code>request_channel</code>	190
B	cGAP MicroBlaze access	193
B.1	Application programming interface	193
B.2	cGAP control	196
C	Design flow libraries	199
C.1	cGAP HLS library	199
C.2	cGAP RTL library	200
C.3	cGAP API library	201

List of Figures

2.1	Iterative processes of a GA.	11
2.2	Representations for GAs: (a) binary, (b) path and (c) Prüfer sequence. . .	12
2.3	Selection operators for GAs: (a) roulette-wheel and (b) binary tournament.	13
2.4	GA crossovers for binary representation: (a) 1-point and (b) uniform. . .	14
2.5	Maximal preservative crossover (MPX) for path representation.	14
2.6	Mutations operators for GAs: (a) bit-flip mutation for binary representation and (b) swap mutation for path representation.	15
2.7	Example of a fitness evolution of a GA for solving the OneMax problem with 128 elements.	17
2.8	Panmictic GAs, distributed GAs and cellular GAs.	18
2.9	Example of a hardware architecture for a panmictic generational GA. . .	26
2.10	Example of a hardware architecture for a panmictic steady-state GA. . . .	27
2.11	Example of a hardware architecture for a distributed GA.	29
2.12	Example of a hardware architecture for a cellular GA.	30
3.1	Overall architecture of the proposed scalable processor array for cGAs. . .	39
3.2	Example of typical used neighbourhoods in cGAs.	44
3.3	cGA neighbourhood solutions of the processor array for different levels of parallelism.	45
3.4	Application of the processor array architecture to build a distributed GA.	49
3.5	Discrete-event simulation of the cGA supported by the processor array to emulate its time-driven update policy.	50
3.6	Example of the maximal preservative crossover applied to the TSP, and how it implicitly performs a mutation operation.	52
3.7	Fitness evolution with the total number of generated solutions, obtained with the SystemC model for the cGA supported by toroidal and square processor arrays to solve a TSP.	54
3.8	Fitness evolution with number of clock cycles, obtained with the SystemC model for the cGA supported by toroidal and square processor arrays to solve a TSP.	57
3.9	Fitness evolution with the total number of generated solutions, obtained with the SystemC model for the cGA supported by toroidal and non-square processor arrays to solve a TSP.	58
3.10	Fitness evolution with the total number of generated solutions, obtained with the SystemC model for the cGA supported by non-toroidal and square processor arrays to solve a TSP.	61
3.11	Fitness evolution with the total number of generated solutions, obtained with the SystemC model for the cGA supported by non-toroidal and non-square processor arrays to solve a TSP.	62

3.12	Organization of a shared memory in the processor array for solving the TSP with a maximum of 252 cities.	63
3.13	Overview of the hardware pipeline implemented in the PE to compute the MPX and fitness of a TSP solution.	64
3.14	Configuration and organization of the auxiliary memory (a BRAM from a Virtex-6 FPGA) in a PE to solve the TSP.	65
3.15	Sequence of operations executed by a PE during a generation of a new solution for solving a TSP with N cities ($N \leq 252$).	66
3.16	Signals involved in the memory access control among two PEs and a shared memory of the processor array architecture for solving the TSP. .	67
3.17	Fitness evolution obtained with the Verilog HDL model for a toroidal and square processor array to solve a TSP.	69
3.18	Throughput of the processor array for the configurations of 2×2 , 4×4 , and 8×8 , normalized to the throughput of a single PE. Results obtained with the Verilog HDL model for solving the TSP.	70
4.1	Overview of the cGA processor (cGAP).	74
4.2	Hardware configuration of the cGAA with (a) toroidal and (b) non-toroidal shape.	76
4.3	Processing element (PE) hardware interface.	78
4.4	Subpopulation memory hardware interface.	79
4.5	Details of the subpopulation memory hardware block.	80
4.6	Example of the handshake protocol between a PE and a subpopulation memory.	81
4.7	Cellular genetic algorithm controller (cGAC) hardware interface.	83
4.8	Overview of the communication infrastructure hardware used for the communication among the cGAC and the PEs.	84
4.9	Command information used in the communication infrastructure.	84
4.10	Details of the FIFOs used in the communication infrastructure to ensure the passage of commands among FIFOs, PEs and cGAC.	86
4.11	Overview of the interface circuits used to ensure the communication and control of the cGAP by a host computer.	87
4.12	Random number generator (RNG) infrastructure hardware used in the cGAA.	89
4.13	RNG based on a cellular automata, with a 1-dimensional ring topology with connectivity $\{-7,0,11,17\}$ and rule 50745, used to feed the RNG infrastructure of the cGAA.	92
5.1	Algorithmic structure used by Catapult HLS to describe the PE.	97
5.2	Algorithmic structure used by Catapult HLS to implement a selection procedure of solutions in a PE.	103
5.3	Sequence of commands issued by two PEs to a subpopulation memory to access the same solution.	105
5.4	Algorithmic structure used by Catapult HLS to describe the cGAC. . . .	107
5.5	Overview of the host communication with the cGAP. A MicroBlaze soft-core processor accesses to the cGAP via AXI4-Lite protocol.	109
5.6	Overview of the hardware design flow used to build the complete cGAP, that connects to a MicroBlaze soft-core processor running a Linux OS, both embedded in a Xilinx Virtex-6 FPGA.	113

5.7	Set-up of the test bench platform used to evaluate the cGAP. A Virtex-6 FPGA, placed in a ML605 board, integrates a MicroBlaze soft-core processor connected to the cGAP. The processor runs a Linux OS with its file system hosted in a PC.	117
6.1	Example of a SA problem with (a) placement of 1 primary and 3 secondary users, (b) availability of channel A among secondary users and primary user, (c) availability of channel B among secondary users	123
6.2	Memory codification of a solution in the SA problem (matrix A).	127
6.3	Subpopulation memory organization in the cGAP used for the SA problem.	128
6.4	Area cost and computation time for the different PE implementation solutions with Catapult HLS and Precision RTL to solve the SA problem. .	134
6.5	Fitness evolution with the total number of generated solutions, obtained for an instance 20_24 of the SA problem for (a) toroidal arrays and (b) non-toroidal arrays.	141
6.6	Speedup achieved by the cGAP with the level of parallelism for the 32_32 SA instance.	144
6.7	Example of a MEB problem with 3 nodes where (a) source node transmits to all the remaining nodes and (b) source node transmits to the closest node.	145
6.8	Example of a MEB solution with 6 nodes where solid edges represent the transmission costs and dashed edges represent implicit transmissions. . .	147
6.9	GA solution representation and codification used in the MEB problem. . .	148
6.10	Possible local search moves by (a) 1-shrink and (b) 2-shrink in the MEB problem.	150
6.11	Subpopulation memory organization in the cGAP used for the MEB problem.	155

List of Tables

2.1	Review of dedicated hardware implementations of GA.	20
3.1	Configurations of the toroidal and square arrays used in the SystemC model simulations.	53
3.2	Fitness values obtained with the SystemC model for the cGA supported by toroidal and square processor arrays to solve a TSP.	56
3.3	Configurations of the toroidal and non-square arrays used in the SystemC model simulations.	57
3.4	Fitness values obtained with the SystemC model for the cGA supported by toroidal and non-square processor arrays to solve a TSP.	59
3.5	Configurations of the non-toroidal and square arrays used in the SystemC model simulations.	60
3.6	Configurations of the non-toroidal and non-square arrays used in the SystemC model simulations.	62
3.7	Characteristics of the toroidal processor arrays implementations on a Virtex-6 (XC6VLX240T-1) FPGA for solving the TSP.	68
4.1	Request results of the handshake protocol among a subpopulation memory and the two PEs connected to it.	82
5.1	List of commands used during the selection procedure of a solution in a PE to access a subpopulation memory. Commands are used with the methods of a <code>request_channel</code> data type variable.	102
6.1	Genetic operations adopted in the PEs for the spectrum allocation problem.	125
6.2	List of commands implemented in the PE and cGAC for solving the SA problem.	126
6.3	Subpopulation memories accesses adopted by a PE for the different operations of the GA for solving the SA problem.	130
6.4	Catapult HLS and Precision RTL results for implementing a PE for the SA problem.	133
6.5	Optimization of Catapult HLS solution for a PE to solve the SA problem.	136
6.6	Xilinx synthesis results taking 6 PE projects generated by the HLS tools.	137
6.7	Characteristics of the Xilinx implementation for cGAPs ranging from 1×1 to 6×6 PEs and toroidal configurations to solve the SA problem.	138
6.8	Characteristics of the Xilinx implementation for cGAPs ranging from 1×1 to 6×6 PEs and non-toroidal configurations to solve the SA problem.	138
6.9	cGAP arrays with toroidal configurations used to solve the instance 20_24 of the SA problem.	140

6.10	cGAP arrays with non-toroidal configurations used to solve the instance 20_24 of the SA problem.	140
6.11	cGAP time results obtained for different SA problem instances. Array of PEs ranges from 1×1 to 5×5 with a non-toroidal configuration.	143
6.12	Fitness results obtained for different SA problem instances with the cGAP and the CSGC heuristic. Array of PEs ranges from 1×1 to 5×5 with a non-toroidal configuration.	144
6.13	Genetic operations adopted in the PEs for the MEB problem.	153
6.14	Characteristics of the different projects used to implement the cGAP with 1-shrink local search for solving the MEB problem.	157
6.15	Characteristics of the different projects used to implement the cGAP with 2-shrink local search for solving the MEB problem.	158
6.16	Results performance of the cGAP-1s and cGAP-2s on 20 node MEB problems.	160
6.17	Results performance of the cGAP-1s and cGAP-2s on 50 node MEB problems.	162
A.1	Description of the C++ class <code>command_type_cGA</code> methods used in Catapult HLS.	189
A.2	Description of the C++ class <code>request_channel</code> methods used in Catapult HLS.	191
B.1	List of registers and their description used by the MicroBlaze to communicate with the cGAP.	194
B.2	Description of the application programming interface C functions used by the MicroBlaze to access the cGAP as a memory mapped device.	194
C.1	High-level synthesis library developed to describe the PE and the cGAC in C++ with Catapult HLS.	199
C.2	cGAP RTL library used to describe the cGAP.	200
C.3	cGAP API library used to communicate with the cGAP.	201

List of Listings

5.1	C++ description of the PE interface to be used by Catapult HLS.	98
5.2	C++ description of the cGAC interface to be used by Catapult HLS. . . .	108
6.1	Excerpt of the C++ code to describe a PE for solving the SA problem. .	130
A.1	C++ code of the <code>command_type_cGA</code> class definition used in Catapult HLS (header file <i>command_type_cGA.h</i>).	187
A.2	C++ code of the <code>command_type_cGA</code> class implementation used in Cat- apult HLS (file <i>command_type_cGA.cpp</i>).	188
A.3	C++ code of the definition of the <code>request_channel</code> class used in Cat- apult HLS (header file <i>request_channel.h</i>).	190
A.4	C++ code of the implementation of the <code>request_channel</code> class used in Catapult HLS (file <i>request_channel.cpp</i>).	190
B.1	C code of the functions that define the application programming interface used by the MicroBlaze to access the cGAP as a memory mapped device.	194
B.2	C code examples of functions that use the API used by the MicroBlaze to control the execution of the cGAP.	196

List of Abbreviations

AMBA	Advanced Microcontroller Bus Architecture
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
AXI	Advanced eXtensible Interface
BIP	Broadcast Incremental Power
BRAM	Block RAM
CA	Cellular Automaton
cGA	Cellular Genetic Algorithm
cGAA	Cellular Genetic Algorithm Array
cGAC	Cellular Genetic Algorithm Controller
cGAP	Cellular Genetic Algorithm Processor
CPU	Central Processing Unit
CSGC	Colour-Sensitive Graph Colouring
CUDA	Compute Unified Device Architecture
CX	Cycle Crossover
DE	Differential Evolution
dGA	Distributed Genetic Algorithm
DSP	Digital Signal Processing
EA	Evolutionary Algorithm
EWMA	Embedded Wireless Multicast Advantage
FF	Flip-Flop
FIFO	First In, First Out
FPGA	Field-Programmable Gate Array
GA	Genetic Algorithm
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HLS	High-Level Synthesis
HW	Hardware
II	Initiation Interval

IP	Intellectual Property
LESS	Largest Expanding Sweep Search
LFSR	Linear Feedback Shift Register
LSB	Least Significant Bit
LUT	Look-Up Table
MA	Memetic Algorithm
MEB	Minimum energy broadcast
MPX	Maximal Preservative Crossover
MSB	Most Significant Bit
NDE	Node-Depth Encoding
NetKeys	Network Random Keys
NFS	Network File System
NoC	Network On Chip
OS	Operating System
PE	Processing Element
PSO	Particle Swarm Optimization
RAM	Random-Access Memory
RNG	Random Number Generator
RTL	Register-Transfer Level
SA	Spectrum Allocation
SoC	System On a Chip
SW	Software
TGFSR	Twisted Generalized Feedback Shift Register
TSP	Travelling Salesman Problem
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

Chapter 1

Introduction

Genetic algorithms (GAs) are search methods inspired by the evolution of living species, where the principles of natural selection and genetics are applied to solve optimization problems. This algorithm is a metaheuristic, which means that little (or even no) knowledge about the problem being solved is required, and it is based on a population since a set of solutions (the population) is evolved during the evolutionary process. Genetics-inspired operators like selection, crossover, or mutation are iteratively applied to the population, guiding the algorithm towards better solutions.

This metaheuristic has its roots back to the 1950's when ideas like a learning machine based on the principles of evolution, or the use of digital computers to simulate artificial selection of organisms were first introduced [Tur50, Fra57]. However, it was only in the 1970's that GAs start to become popular through the work developed by John Holland, especially with his book *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence* [Hol75].

Since then, and with the development of computers, GAs have become one of the best known and widely accepted metaheuristics used to solve complex optimization problems. Their applications include clustering in data mining and bioinformatics, path planning for autonomous navigation, antenna design, etc. [MBM11, THC11, HGLL06]. Moreover, the ideas behind the algorithm led to new fields of investigation like genetic programming that seeks to find a computer program to perform pre-defined tasks [PK14], or evolvable hardware where logic systems can adapt their architecture dynamically and autonomously [HT11].

The success of GAs is due to the fact that they can effectively explore a huge search space of a problem, as it happens with NP-hard problems, only by examining a small fraction of it. However, GAs usually suffer from long execution times, because large

numbers of generated solutions are required for evolving the population to satisfactory results. Moreover, a trade-off between solution quality and algorithm execution time exists and thus better quality means more time to evolve the population. Therefore, even though the GAs' iterative process is made of simple operations, the time needed to reach a final convergence can be high.

Thus, solving optimization problems where time constraints apply or when the processing power is restricted can limit the use of GAs. To overcome this, dedicated computing platforms that aim to accelerate the algorithm can be developed. Over the last 20 years, continuous research in this domain has been carried out, mainly motivated by the widespread availability of low cost reconfigurable computing platforms like field-programmable gate arrays (FPGAs), as for example in the latest works [GTL14, GTL13, SS12, NBKHM13].

Most of the work in this field has targeted general-purpose engines for GAs, based on the fact that the metaheuristic has a few and simple operators that can be efficiently implemented and applied generally to any optimization problem. This is, indeed, true since metaheuristics do make few assumptions about the optimization problem and use their pre-defined operators to guide the search, regardless of the optimization problem. Although GAs are often associated to a solutions' representation with a binary encoding where a valid solution is represented by a list of bits, others representations exist that can be used, for example, to encode solutions for graph problems. As a result, the genetic operators of a GA must be adapted for different representations. Additionally, it is often the case that specific constraints apply, where the encoding scheme used to represent a valid solution requires the application of specialized algorithms. In addition, the evaluation of the quality of a solution is always problem dependent. Therefore, general-purpose hardware architectures may not be so general since they can only be applied to problems with a straight implementation in GAs.

However, even though GAs require a large number of iterations that must be sequentially processed when emulating their evolution process with a sequential processor, they are inherently parallel [LA11]. This is true since the algorithm uses a population of solutions to evolve and thus, several solutions can be generated at the same time by parallelizing the GA operations. Nonetheless, in dedicated processor implementations this parallelism may result in memory access bottlenecks while accessing the population, thus limiting a potential performance improvement.

Nevertheless, it is possible to decentralize the population of a GA, as happens with distributed GAs (dGAs) or cellular GAs (cGAs) [LA11, AD08]. In a dGA the solutions are kept in different subpopulations that evolve independently from each other and occasionally exchange solutions among them; whereas in a cGA the solutions are distributed

over a regular grid where a solution only interacts with solutions placed in its neighbourhood. Regardless of which decentralized GA is adopted, their solutions can be kept in independent memories which can avoid potential bottlenecks while accessing to them. Therefore, it is possible to have several processing elements working in parallel with a subset of solutions placed in different memories, thus eliminating major bottlenecks during memory access.

Although the decentralization of the population is appealing for hardware implementation, most of the works in this field neglect it and use instead a single population where a solution can interact with any other at any iteration of the algorithm, as for example in [GTL14, GTL13, NBKHM13]. Therefore, the population memory access for ensuring parallelism is often neglected, or only considered for small populations. Although a few works have been carried out that implement dGAs, they do not achieve high levels of parallelism as few processing elements are implemented [THC11, JKFA06, TMS⁺06]. Moreover, and to our best knowledge, there is no other implementation of cellular genetic algorithms in FPGA devices. It is clear that there is room to create new custom computing architectures for dGAs and cGAs. Moreover, these subclasses of GAs have a great potential to improve the acceleration of the algorithm, as it has been often pointed out by practitioners of these metaheuristics [LA11], but somehow ignored by the designers of GAs in dedicated hardware.

The main goal of this thesis is to look into the design of custom computing machines capable of accelerating genetic algorithms with FPGAs. In addition, we target a general-purpose engine that can be used to solve different optimization problems, without being restricted to problem-specific requirements. To allow that, we propose a design flow based on high-level synthesis (HLS) where the problem-specific operations are described in C++ and translated to digital hardware, thus easing the design of new engines. Therefore, the core genetic operators are not part of the computing array because a plethora of operators exist that must be adjusted to a particular optimization problem. Instead, we specify an architecture framework built of processing elements (PEs) and memories conveniently connected that can parallelize the algorithm without introducing heavy memory accesses bottlenecks. Additionally, this work makes hardware development accessible to people with little hardware knowledge, but with knowledge of the application domain.

Our architecture implements specifically a cellular genetic algorithm with a computing performance determined by the number of PEs to meet the desired trade-offs between execution time and hardware resources used. Moreover, since we do not constrain the operations implemented, the infrastructure of PEs and memories can be used for other metaheuristic procedures based on populations.

1.1 Thesis organization

This thesis consists of this introduction chapter, followed by five chapters, concluding remarks, and three appendices.

In Chapter 2, an overview of genetic algorithms is presented. We detail the main operations of this metaheuristic and present the concept of GAs with a structured population. We perform a review of the state of the art about custom computing implementations of these algorithms and discuss the impact that the different subclasses of GAs can have on the hardware performance, mainly the population memory organization. We also make general considerations about the different operations of the algorithm and how they can impact the hardware architecture.

In Chapter 3, a scalable processor array for accelerating cellular GAs is proposed. The main architecture of this hardware engine is explained together with their characteristics. In addition, a comparison with existing (canonical) cellular GAs in the literature is performed so that the algorithm supported by our architecture is classified. To evaluate the convergence of the algorithm, we perform a system-level architecture simulation of the proposed scalable array processor for solving the travelling salesman problem (TSP). We explore results for different configurations of the processor array, and analyse the impact in the algorithm convergence and the potential acceleration. Additionally, a synthesizable hardware model targeting a Virtex-6 FPGA is presented for solving the same optimization problem. Results have shown a promising performance of this particular engine that solves the TSP, with an acceleration proportional to the number of PEs without compromising the quality of the solution found by the algorithm as the parallelism level increases.

In Chapter 4, we present and specify the complete hardware architecture of the proposed scalable array, which we call cellular genetic algorithm processor (cGAP). In particular, the chapter focuses on specifying the hardware infrastructure required to support the array of processors (the PEs), thus not including the blocks whose operations depend on the optimization problem. We detail how the different processor elements can access to the memory contents where the subsets of the population are placed. Additionally, we explain how the algorithm execution can be globally controlled and monitored, as well as the specification of a communication infrastructure used for this purpose. All the hardware blocks specified in this chapter have been described at the register-transfer level (RTL) using Verilog hardware description language (HDL), while the PEs and global controller are built with high-level synthesis, as presented in the next chapter.

In Chapter 5, we explain how the blocks that are problem-dependent can be specified in C++ to use a HLS-based design flow. These blocks are the PE and the algorithm

controller which we call cellular genetic algorithm controller (cGAC). Additionally, we present the interfaces required so that the cGAP can be accessed by a host processor. We provide several examples of how the PE and cGAC can be specified, as well as C language examples to be executed in the host so that the cGAP is accessed as a peripheral. Finally, we define a complete design flow, which takes the blocks described in Chapter 4, together with the PE and cGAC specified and tailored for an optimization problem, for building the cGAP. In addition, this design flow includes a test bench platform that implements in a Virtex-6 FPGA a MicroBlaze soft-core processor running Linux to control and communicate with the cGAP as a peripheral.

In Chapter 6, we use two optimization problems as case studies to be implemented with the proposed design flow. We selected two representative problems related to wireless ad hoc networks: a spectrum allocation problem in cognitive radios and the minimum energy broadcast problem. These problems are thus used to demonstrate how the PE and cGAC are specified with the use of HLS tools. We present details of these implementations and provide examples of how the cGAP architecture can be further exploited to improve the hardware performance. Furthermore, we provide examples of how to conveniently explore the HLS design space by tuning various optimizations of the synthesis tools and improve the quality of the hardware system. Implementation results are presented together with acceleration figures obtained by the cGAP.

Finally, Chapter 7 presents the concluding remarks and identifies future research work for this thesis.

Additionally, three appendices are included. Appendix A presents the C++ classes used to describe the cGAP for the HLS tools. Appendix B provides C language functions used by the host processor (a MicroBlaze) to access the cGAP as a memory mapped device. Appendix C summarizes all the files for the different libraries used in the design flow, mainly the cGAP HLS library, the cGAP RTL library, and the cGAP API library.

1.2 Contributions

The main goal of this research work is to study mechanisms to accelerate population based metaheuristics with configurable custom computing platforms, mainly genetic algorithms with FPGAs. We propose a design flow where the problem-specific operations are specified in HLS, thus easing the design of the hardware architecture for different optimization problems. Specifically, we conceived a scalable architecture where an array of processors execute a cellular genetic algorithm. All the hardware blocks described

throughout this thesis as well as the algorithms have been entirely designed and developed from scratch. Additionally, a comprehensive state of the art about custom computing hardware implementations of GAs has been performed to understand what is required to build an engine that accelerates the algorithm that could be applied to different optimization problems.

The main contributions of this thesis can be summarized as follows:

- A scalable processor array for accelerating cGAs has been proposed. This hardware architecture is constituted of several processing elements that execute the algorithm operations, and are conveniently connected to a set of shared memories where the solutions of the problem (population) are placed. This architecture can be built with different levels of parallelism while ensuring that no memory bottlenecks are introduced as the size of the processor array increases. Although it has been developed to solve cellular genetic algorithms, it can also be applied to other population-based metaheuristics by changing the functionality of the processing elements.
- A system-level simulation model of the proposed processor array has been implemented in SystemC language. This model is used to evaluate the cGA supported by the engine under different configurations of the array of processors.
- A specification at the register-transfer level using Verilog HDL has been done of all the elements required to support our architecture (the cGAP). This comprises the specification of the access to the shared memories where the solutions are placed; the interfaces of the cGAP; a communication infrastructure to support the transmission of commands between the controller (the cGAC) and the PEs; and a global random number generator infrastructure to provide random numbers to the processing elements.
- A design flow has been proposed that combines the modules specified at RTL with the problem-specific modules which are specified in C++ and built with HLS, thus easing the design of the cGAP for different optimization problems. A set of algorithm templates in the C++ language are used to describe the PEs and the cGAC. The interfaces of these two modules have been specified to communicate with all the corresponding blocks described in Verilog HDL while hiding implementation details. The design flow includes all the necessary project constraints, and a small set of parameters that configure the array of PEs and the memories that keep the population. Additionally, a test bench platform is included in the design flow that implements a MicroBlaze soft-core processor running Linux to control and communicate with the cGAP as a peripheral.

- A significant contribution is also the demonstration of the effectiveness of the proposed approach to two relevant problems in the domain of wireless ad hoc networks. In the first problem, a spectrum allocation in cognitive radios, examples are given of how the cGAP architecture, mainly the shared memories that keep the population, can be further exploited to improve the hardware performance. Additionally, we have explained how to conveniently constrain the designs developed by the HLS tool so that the final results are improved. Moreover, we have analysed the convergence of the algorithm running in the cGAP for different levels of parallelism and compared the results with an existing heuristic. Additionally, for the results obtained we have verified that the throughput of the cGAP is (almost) directly proportional to the number of PEs. In the second problem, the minimum energy broadcast, we have considered an existing implementation of a memetic algorithm (i.e. GA with a local search heuristic to improve the solutions' quality) and have ported it to the cGAP. The algorithms used to describe the PEs with HLS are far from the straightforward implementations of typical GA operators, and yet they have been successfully implemented without major concerns using the design flow. Results have shown that the cGAP can achieve superior quality solutions and a faster execution time when compared to the original algorithm running in a personal computer.

The contributions of this research work have led to the following publications:

- P. V. dos Santos, J. C. Alves, and J. C. Ferreira, "An FPGA Framework for Genetic Algorithms: Solving the Minimum Energy Broadcast Problem", 18th Euromicro Conference on Digital System Design (DSD), pages 9-16, Funchal, Portugal, 2015.
- P. V. dos Santos, J. C. Alves, and J. C. Ferreira, "A Custom Computing Machine for Cellular Genetic Algorithms", 1st Doctoral Congress in Engineering (DCE), pages 1-2, Porto, Portugal, 2015.
- P. V. dos Santos, J. C. Alves, and J. C. Ferreira, "High-level synthesis of custom processing elements for genetic algorithms", Proceedings of the XVIII Conference on the Design of Circuits and Integrated Systems (DCIS), pages 74-79, San Sebastián, Spain, 2013.
- P. V. dos Santos, J. C. Alves, and J. C. Ferreira, "A framework for hardware cellular genetic algorithms: An application to spectrum allocation in cognitive radio", 2013 23rd International Conference on Field Programmable Logic and Applications (FPL), pages 1-4, IEEE, Porto, Portugal, 2013.

- P. V. dos Santos, J. C. Alves, and J. C. Ferreira, “A scalable array for cellular genetic algorithms: TSP as case study”, International Conference on Reconfigurable Computing and FPGAs (ReConFig), pages 1-6, IEEE, Cancun, Mexico, 2012.
- P. V. dos Santos, and J. C. Alves, “A cellular genetic algorithm architecture for FPGAs”, VIII Jornadas sobre Sistemas Reconfiguráveis (REC), pages 21-25, Lisboa, Portugal, 2012.
- P. V. dos Santos, and J. C. Alves, “FPGA based engines for genetic and memetic algorithms”, 2010 International Conference on Field Programmable Logic and Applications (FPL), pages 251-254, IEEE, Milano, Italy, 2010.

Chapter 2

Background and state of the art

2.1 Introduction

In this chapter an overview of genetic algorithms (GAs) is presented, together with a review of their dedicated hardware implementations. It starts in Section 2.2 by presenting the GA in its canonical form and explaining the different operations that constitute the algorithm. Additionally, the concept of structuring the population in a GA, which leads to the distributed GA (dGA) and cellular GA (cGA), is introduced.

With the aim of building a custom computing machine to accelerate GAs, the state of the art about hardware implementations of GAs is presented in Section 2.3. The review focus on the effects of structuring the algorithm's population, and how this impacts on the architecture of the custom hardware.

2.2 Genetic Algorithms

Solving combinatorial optimization problems has always been a research topic of great interest in today's world. With the development of computer science, a myriad of more complex and harder optimization problems are continuously emerging. Exact methods can be used to solve such problems which guarantee to find the optimum solution. Nevertheless, such techniques require an execution time that grows exponentially with the instance size for solving NP-hard problems, becoming this way impractical for handling real-world problems. On the other hand, approximated methods do not guarantee to find the optimum solution; instead they provide sub-optimal solutions (eventually the optimum) in a realistic time. This is the case of *metaheuristic* search methods, where

Algorithm 1 Pseudo-code of a canonical GA.

```

1:  $P \leftarrow \text{Generate\_Initial\_Population}()$ 
2:  $\text{Evaluate}(P)$ 
3: while not  $\text{Termination\_Condition}()$  do
4:    $P' \leftarrow \text{Select\_Parents}(P)$ 
5:    $P' \leftarrow \text{Apply\_Crossover}(P')$ 
6:    $P' \leftarrow \text{Apply\_Mutation}(P')$ 
7:    $\text{Evaluate}(P')$ 
8:    $P \leftarrow \text{Replacement}(P, P')$ 
9: end while
10: return Best solution found

```

a higher-level procedure is responsible to provide an appropriate good solution of an optimization problem through the use of simple heuristic methods.

Metaheuristics can be applied to a wide range of optimization problems as these techniques do not rely on a previous knowledge of the problem. Instead, a high-level strategy is used to guide the search, where a set of operations (variation operators) are applied to explore the search space. A good metaheuristic is capable of providing a trade-off between an exploration of all the search space, while a exploitation occurs in good solutions. This is also known as diversification and intensification, and it represents the key mechanism to ensure good quality solutions (an effective algorithm) in a low computational time (an efficient algorithm) [BR03].

There exist two major classifications for metaheuristics based on the number of candidate solutions of the optimization problem used during the algorithm evolution: trajectory based and population based. In the first, the metaheuristic works with a single candidate solution, whereas in the population-based case, a set of solutions (*population*) are evolved during the search. Examples of trajectory based metaheuristics are simulated annealing [KJV83] and tabu search [GL⁺97]. Evolutionary algorithms [BFM97] and particle swarm optimization [BDT99] are examples of population-based metaheuristics.

2.2.1 Canonical genetic algorithm

The genetic algorithm (GA) is a population-based metaheuristic that belongs to the class of evolutionary algorithms where a population goes through an evolutionary process inspired in the biological evolution of living species [Hol75]. Therefore, this metaheuristic relies on operations that can be found in nature and consist, for example, in natural selection and genetics-inspired operations like crossover and mutation. The population of solutions compete and cooperate among them so that during the evolutionary process the better solutions survive which, in turn, will propagate their genetic information to future generations.

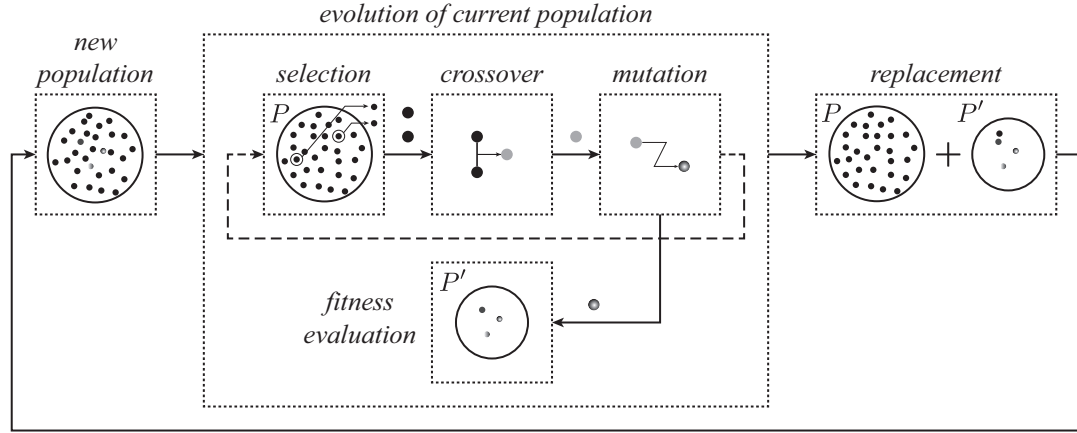


FIGURE 2.1: Iterative processes of a GA.

Algorithm 1 presents the canonical genetic algorithm. It starts with a population P which is a set of tentative solutions of the optimization problem to be solved. All the solutions are evaluated by a *fitness* evaluation function that quantifies their quality according to the objective function of the optimization problem. After that, the algorithm proceeds with an iterative process where genetics-inspired operations are applied as illustrated in Figure 2.1. First a *selection* of solutions in P is performed to elect solutions that will undergo some transformations to create new solutions. Typically, the selected solutions are called *parents* and are combined (usually in pairs) to generate new ones through a *crossover* operation. Then, the new solutions suffer a *mutation* operation that induces small changes to them. The generated solutions P' are then evaluated and the population for the next generation is elected among the solutions in P and P' , according to an evolution strategy that takes into account the fitness values to promote the evolution towards a better population¹.

2.2.1.1 Representation of solutions

The first step for using a genetic algorithm to solve a given optimization problem is to encode the decision variables into a convenient format that codes a tentative solution to the problem. A solution encoded in such way is called *chromosome* (or individual), and its elementary data is a *gene*. The practical results of a GA implementation strongly depend on the way a solution is encoded. Figure 2.2 illustrates examples of possible encodings, although the universe of possibilities is vast and dependent on the optimization problem. In addition, it may be required to process/decode a given chromosome so that it represents a valid solution to fulfil specific constraints of the problem.

¹In this context, a better population means that in average the fitness values in the population tend to improve during the evolution of the algorithm. Nevertheless and to promote diversity of solutions, it is perfectly admissible that solutions with a worst fitness value are also included for future generations.

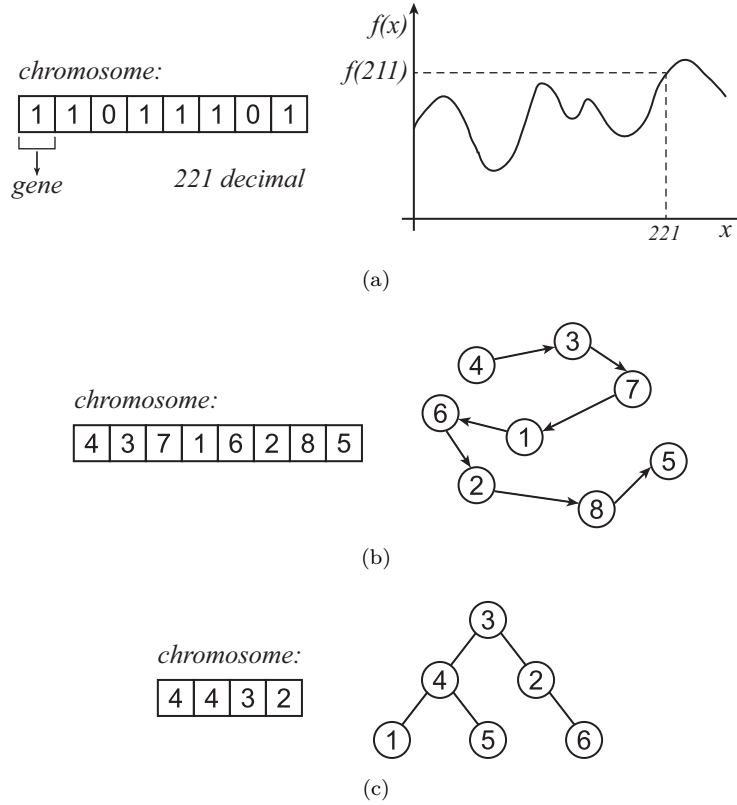


FIGURE 2.2: Representations for GAs: (a) binary, (b) path and (c) Prüfer sequence.

Figure 2.2(a) depicts an example of a chromosome with binary representation where the different bits (genes) form a binary number, representing a solution of the problem. A fitness value provided by a function $f(x)$ quantifies the quality of the solution. The binary representation can have many different interpretations depending on the encoding scheme used. For instance, a gene can represent directly a decision variable of a problem with binary variables.

Although the binary representation is usually the most referred encoding scheme in genetic algorithms, there exist others like the path representation, where the chromosome is a set of ordered and unique symbols [LKM⁺99]. In Figure 2.2(b) the list $\{4, 3, 7, 1, 6, 2, 8, 5\}$ represents the order from which the different elements, in this case nodes of a directed graph, are traversed. This encoding scheme can be used for instance in minimum path problems, like the travelling salesman problem (TSP) where a given number of points must be visited exactly once in the shortest possible path [Ale05].

The case of solutions of a given problem formed by a tree (a connected graph with no cycles) is another example where specific representations of solutions in genetic algorithms can be explored, for example, to solve the degree constrained minimum spanning

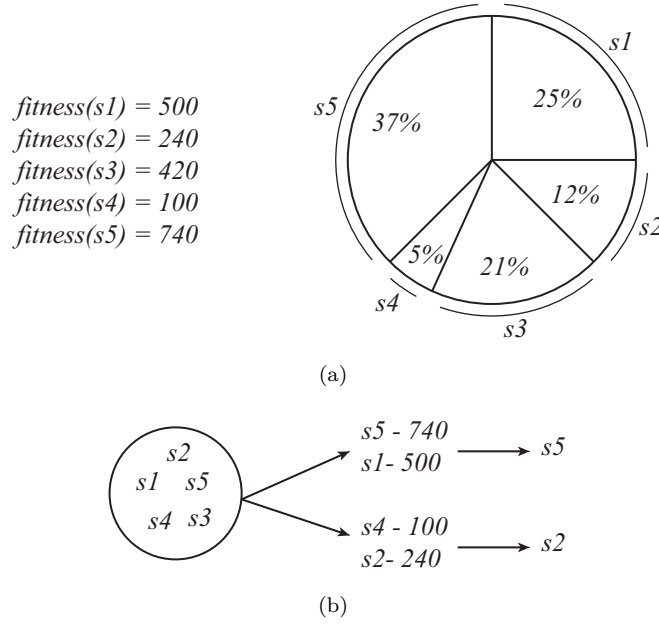


FIGURE 2.3: Selection operators for GAs: (a) roulette-wheel and (b) binary tournament.

tree problem [KES01]. There are many different encoding schemes for tree representations and graphs in general, like characteristic vector [DOCQ93], network random keys (NetKeys) [RGH02], and node-depth encoding (NDE) [LLHD05] each with its own characteristics. One of the best known tree codification is the Prüfer numbers, where a unique sequence is associated with a tree [PK94]. In Figure 2.2(c) the sequence $\{4, 4, 3, 2\}$ is the chromosome that represents the Prüfer sequence of the shown tree. Essentially, to construct this sequence with a known tree we select the lowest numbered node of degree² 1 in the tree (in our example, node 1) and the node's number that is connected to it goes to the Prüfer sequence (node 4). The selected node with degree 1 is removed from the tree and the step described above is repeated till only 2 nodes are left in the tree. Details of the algorithms to construct a tree from the Prüfer sequence and vice versa can be found in [Rot06].

2.2.1.2 Selection

The genetic algorithm starts by a selection procedure where better individuals of the population are preferred to be combined to generate new solutions. It is expected that the solutions with best fitness values can potentially create better solutions than the worse ones since they have better genetic information, helping this way the evolution of the population during the iterative process. The roulette-wheel method [SGK05] is one of the most used in the genetic algorithm context and it consists of assigning a

²In graph theory, the degree of a node of a graph is the number of edges connected to it.

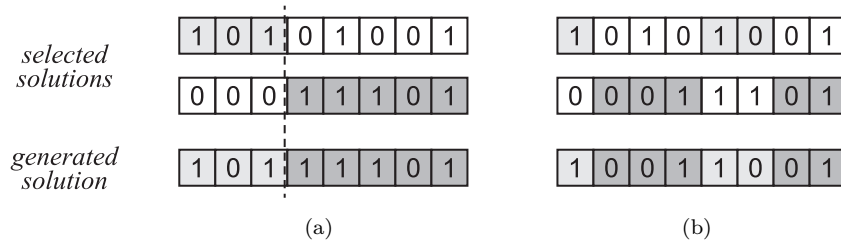


FIGURE 2.4: GA crossovers for binary representation: (a) 1-point and (b) uniform.

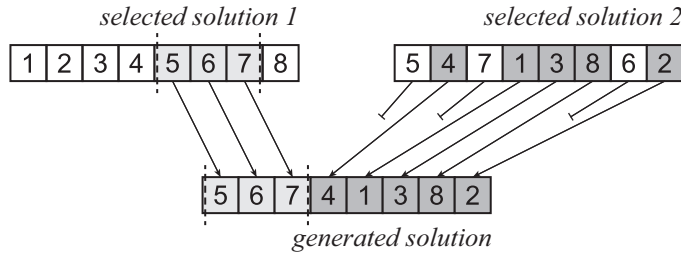


FIGURE 2.5: Maximal preservative crossover (MPX) for path representation.

probability to each solution of being chosen that is proportional to its fitness value as depicted in Figure 2.3(a). This way, better solutions are likely to be chosen, whereas the worse ones are selected with a lower probability which helps to prevent premature convergence of the algorithm. Another approach is a tournament selection where k individuals are randomly chosen to enter into a tournament against each other. The element with the best fitness value wins that tournament and is elected as one parent. The process is repeated to choose the other parent. Figure 2.3(b) illustrates an example of a binary ($k = 2$) tournament.

2.2.1.3 Crossover

After selecting the solutions, these individuals (or parent solutions) go through a cross-over operation where two solutions are combined to generate a new solution. Different representations of chromosomes may require specific crossover operators to ensure that the genetic information is passed to the new solutions (children) and, at the same time, to ensure that they generate feasible solutions.

Figure 2.4(a) depicts an example of a 1-point crossover applied to binary chromosomes. This operator splits the chromosomes' information over a randomly chosen point in a way that the generated solution receives information from both parent solutions. The uniform crossover is another example that can be applied to a binary codification of solutions. In this case, each gene of the new solution is randomly chosen from one of the two selected solutions as it can be seen in Figure 2.4(b).



FIGURE 2.6: Mutations operators for GAs: (a) bit-flip mutation for binary representation and (b) swap mutation for path representation.

Although the 1-point and uniform crossovers present an appropriate approach concerning the passage of genetic information from parents to children, they cannot always be applied because this transformation can result in unfeasible or even meaningless solutions. In the case of the path representation (see Figure 2.2(b)), a direct copy of parts of the chromosome parents to create a new solution can easily generate an invalid solution as duplicate genes can occur. Therefore, specific crossover operations exist to deal with the path representation, like cycle crossover (CX), partially matched crossover (PMX), maximal preservative crossover (MPX), among others [LKM⁺99]. As an example, Figure 2.5 presents the MPX crossover [MGSK88]. This operator chooses randomly a set of consecutive genes from the first solution that will be copied to the beginning of the new solution's chromosome. After this, the missing genes of the new solution are filled in the same order as they appear in the second solution, ensuring this way that no duplicated genes appear.

Different representations of chromosomes may require specific crossover operators to ensure that the genetic information is passed to the children solutions and, at the same time, to ensure that they generate feasible solutions. However, to fulfil specific problem's constraints it may not be possible for the crossover to generate a valid solution, and thus a dedicated procedure must be added to correct the solution.

2.2.1.4 Mutation

Once the new solution has been created with the crossover operation, a mutation operator can be applied to it that induces random changes in the solution and thus it fosters diversity. Figure 2.6(a) shows an example of a bit-flip mutation operator applied to a chromosome with binary representation, where a given gene's information is changed (a 0 is converted to 1 or vice versa). As it happens with the crossover operators, different chromosomes representations require specific mutation operators so that they generate valid solutions. Figure 2.6(b) depicts a swap mutation operator applied to a chromosome with a path representation, where two randomly chosen genes are exchanged. Usually the crossover operation is performed with a low probability (usually below 10 %) and it performs a random walk in the vicinity of the solution.

2.2.1.5 Replacement

After the new solutions are created with the selection, crossover and mutation operations, it is required to introduce them in the present population. There exist several techniques to accomplish this, like the delete-all where all the solutions of the population are replaced by an equal number of generated new solutions. Another approach is when n new solutions replace n old solutions, keeping the population size constant. In this case, it must be decided the criterion to select the solutions to be deleted, which can be by selecting the worst ones, a random choice or the selected parents [SGK05]. Despite of the technique chosen, it is expected that in average the population improves gradually its fitness values during the evolution of the algorithm.

2.2.1.6 Fitness evaluation

As stated previously, the fitness evaluation operation measures the quality of the solutions during the evolutionary process of the GA. Therefore, it is an essential operation in the algorithm that allows the choice of better solutions over worse solutions, so that the algorithm evolves as desired. This operation is unique for each different optimization problem, and thus it cannot be a generic operator.

2.2.1.7 Example: OneMax problem

As an example of an implementation of a GA, Figure 2.7 presents the fitness evolution during the execution of the algorithm for the OneMax problem, consisting in maximizing the numbers of ones in a binary string [AD08]. In this example, a solution for the problem is represented by a binary vector with 128 elements and a GA chromosome representation is a straightforward copy of such vector (binary representation). The selection operation consists in a binary tournament as explained previously, and two experiments were performed with the crossover: one with the uniform crossover and the other with 1-point crossover. The mutation operator is the bit-flip where each gene (a bit) of the generated solution has a probability of being flipped of 2%. At each generation of the algorithm, a single solution is created by the genetic operators to replace a randomly chosen solution from the population. This replacement only occurs if the fitness of the new solution is better than the solution chosen to be deleted. A total of 100 individuals exist in the population of the GA.

The results obtained show the convergence curve during 6000 new generated solutions (or iterations of the algorithm) for the best solution in the population and the average fitness value of the population. As it can be seen from Figure 2.7, the uniform crossover

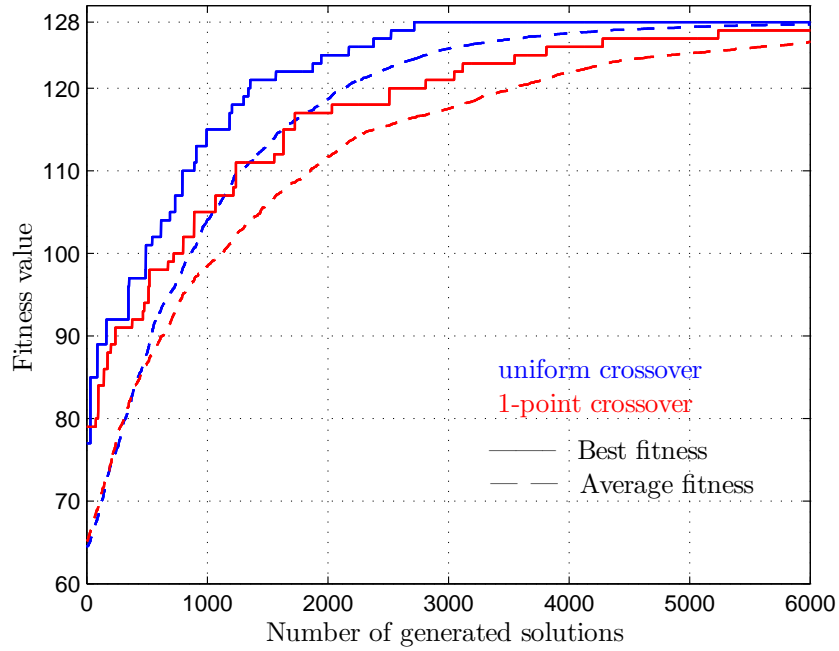


FIGURE 2.7: Example of a fitness evolution of a GA for solving the OneMax problem with 128 elements.

converges to the optimum solution (128) faster than the 1-point crossover. Also, the average fitness value tends to the optimum solution value, meaning that all the solutions will eventually achieve the optimum point. Although in this simple example the uniform crossover shows a superior performance over the 1-point crossover, this may not be true for different problems. Knowing the best genetic operators for solving an optimization problem requires an extensive experimentation where the different operators (selection, crossover, mutation and replacement) are verified for their quality. Moreover, a trade-off must also be performed between an efficient algorithm that converges faster towards an optimum value, and an effective algorithm that converges to better solutions even if it takes a longer time.

2.2.2 Decentralized GAs

Most genetic algorithms implement a global selection procedure, meaning that any solution can potentially mate with any other present in the population (see Figure 2.8(a)). This type of GA is called *panmictic* (means random mating) and it can be essentially categorized in two classes concerning the reproductive step (or replacement operation) [LA11]. The first one, called *generational*, a whole new population is generated that replaces the old one at each iteration of the algorithm. The second type is called *steady-state*, as one new individual replaces an existing solution in the present population

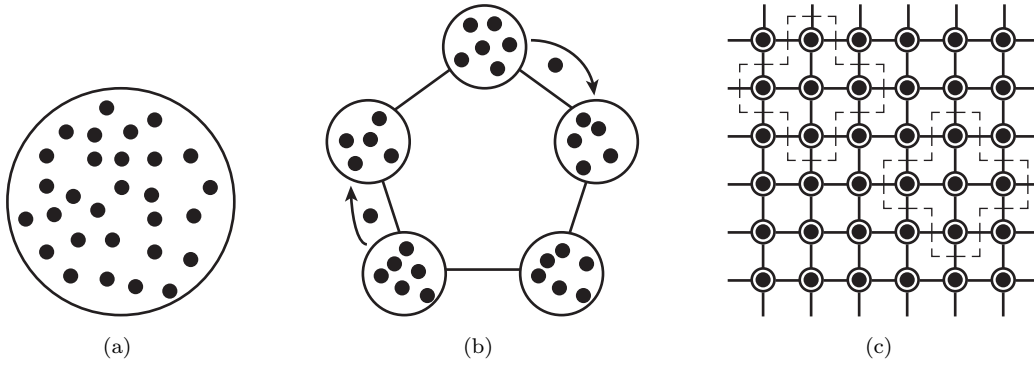


FIGURE 2.8: A panmictic GA (a) has all the solutions (black points) in the same population. With a structured GA the population is partitioned, as it happens with the (b) distributed and (c) cellular GA.

and, therefore, coexisting with its parents for the next iteration of the algorithm. These classifications are identical to the replacement techniques of delete-all and delete n solutions ($n = 1$) at each generation of the algorithm presented previously in Section 2.2.1.5. Nevertheless, between these two opposite classes of panmictic GAs, a different number of solutions can be generated to replace existing solutions in the population during a reproductive step.

In contrast to the panmictic GA, in a *structured* GA the population is somehow decentralized, resulting that a given solution can only be combined with a limited set of other solutions. It has been observed that this class of GAs is often faster and can lead to better solutions than a panmictic GA, as it provides a better sample of the search space [AT02]. Two major classes of structured GAs exist: the *distributed* and *cellular* genetic algorithms.

In the distributed GA (dGA), the complete population is partitioned into smaller and independent subpopulations. During the evolution of the algorithm, solutions that belong to a certain subpopulation sporadically migrate to another subpopulation. Figure 2.8(b) illustrates this concept with a distributed GA with 5 subpopulations that can migrate solutions according to a given connection topology. Besides the GA operations that must be performed in each subpopulation, the distributed GA requires the execution of a migration strategy which depends on the number of generations between two consecutive exchanges, the number of solutions to be migrated during a migration step, and the selection/replacement criterion of the migrant solutions.

With a cellular GA (cGA) model, the solutions of the population are distributed over a regular grid and a solution can only mate with solutions that exist in its neighbourhood. This approach relies on the concept of structuring the population into overlapped

neighbourhoods, where a single solution belongs to more than one of those neighbourhoods. This way, the genetic information of a given individual spreads throughout the whole population without the need to implement explicit migration of solutions. Figure 2.8(c) illustrates this model where a given solution can interact with solutions that are positioned in the grid at north, south, east and west from itself, although different neighbourhoods can be used [AD08]. As it can be seen, two neighbour solutions share solutions (two for the example) from their neighbourhoods. Usually, the distribution of the solutions over a grid follows a toroidal shape which ensures that all the solutions have exactly the same neighbourhood structure [AD08].

As both dGA and cGA have several (almost) independent subpopulations or solutions running during the iterative process, these algorithms have excellent properties to be implemented in parallel computing platforms. The dGA presents a more coarse-grained architecture when compared to the cGA and it is suitable to be implemented in multiprocessor architectures, allocating a processor to one subpopulation or a set of subpopulations. Similarly, the cGA offers the possibility to explore its regular distribution of the solutions to be implemented in more fine-grained computing platforms like graphics processing units (GPUs) or field-programmable gate arrays (FPGAs).

2.3 Hardware implementations of GAs

This section presents a review of the state of the art about implementations of genetic algorithms with dedicated hardware. Since the 1990's there has been a continuous research activity in this subject, motivated by the widespread computing platforms like FPGAs, GPUs or multiprocessors. Nevertheless, the majority of these works have as main goal the acceleration of the operations of a genetic algorithm, like selection or crossover. Although these operations are actually the common operators in GAs, they are not in several cases the bottleneck that constrains the overall performance of the algorithm. Therefore, there has not been paid much attention to the architecture of the algorithm regarding the decentralization (or not) of the population, and how this can be advantageous for implementing GAs as custom hardware specialized processors.

In Table 2.1 it can be seen an overview of the main works that implement GAs in dedicated hardware reported in the last years. In this review, we will not address implementations of GAs that run exclusively in a central processing unit (CPU) of a computer with the aim of accelerating the algorithm. Instead, we will focus on implementations that target dedicated hardware platforms like FPGAs or application-specific integrated circuits (ASICs). The table is organized according to the architecture of the GA: panmictic (generational and steady-state), distributed and cellular. It refers the operations

TABLE 2.1: Review of dedicated hardware implementations of GA.

Work	Optimization problem	GA architecture	Population size	Chromosome size	Genetic operations	HW platform	HW clock frequency	Speedup	Observations
<i>Panmictic generational GAs (some works are only classified as panmictic since they do not provide further details)</i>									
[GTL14]	Locating problem; Set covering problem (117×27); Mathematical functions	Panmictic (generational)	128; 32		Sel: roulette-wheel, tournament Cx: 1-point, multi-point, blending Mut: bit-flip, real-valued	FPGA Xilinx Virtex-6 (Maxeler's MAX3 card)	75 MHz	$30\times$ in average (vs. PC @ 2.67 GHz)	Proposes an automated framework for creating and executing general-purpose GAs in FPGAs. Hardware is created by a HLS tool (Maxcompiler by Maxeler).
[GTL13]	Set covering problem (35×15 ; 32×32)	Panmictic (generational)	128		Sel: roulette-wheel Cx: 1-point Mut: bit-flip	FPGA Xilinx Virtex-6 (Maxeler's MAX3 card)	75 MHz	$60\times$ (vs. PC @ 2.67 GHz)	Uses Maxeler system to describe the hardware (written in Java language).
[WS12]	Generating daily activity plans in an Artificial Transportation System	Panmictic (generational)	512		Sel: random Cx: 1-point Mut: swap	nVidia graphic cards Tesla C2050	1.15 GHz	$23\times$ and $32\times$ (vs. PC @ 2.1 GHz)	Solves in a GPU the problem of generating daily activity plans for individual and three-person household agents.
[SS12]	Mathematical functions	Panmictic	32; 16; 5	8; 4	Sel: tournament, elitism Cx: 1-point, 2-point Mut: bit-flip	FPGA Xilinx Spartan-3, Virtex-4, Virtex-5		Authors claim that there is a shift from seconds to nanoseconds between SW and HW.	
[CW11]	Parameters of PID controller	Panmictic	40	48	Sel: tournament Cx: 3-point crossover Mut: bit-flip	FPGA Altera			Presents a GA to compute the parameters of a PID controller. Uses Altera's IP core generation tool for building the hardware (only simulation).
[PSJ10]	Knapsack problem (4 to 40)	Panmictic (distributed GA with 1024 nodes but no migration).	256	4; 40	Sel: tournament Cx: uniform Mut: bit-flip	nVidia graphics card GTX 260	1.24 GHz	$1340\times$ (4-bit) and $134\times$ (40-bit) (vs. PC @ 2.66 GHz; values considering number of solutions produced in 1024 nodes)	
[FKK ⁺ 10]	Mathematical functions	Panmictic (generational)	255	16	Sel: roulette-wheel Cx: 1-point Mut: bit-flip	FPGA Xilinx Virtex-II Pro	50 MHz	$5.16\times$ (vs. processor PowerPC embedded in the FPGA)	Reports a general-purpose GA (several parameters can be changed). Architecture includes a specialized module for different fitness functions. Uses a HLS tool for the HW description.

continued ...

Table 2.1 – . . . continued

Work	Optimization problem	GA architecture	Population size	Chromosome size	Genetic operations	HW platform	HW clock frequency	Speedup	Observations
[DDT08]	Mathematical functions; TSP (8)	Pannictic (generational)	32	16; 32	Sel: roulette-wheel Cx: 1-point, 2-point, uniform Mut: single-point, masked, uniform; special crossover for TSP	FPGA Xilinx Spartan-3	92 MHz; 91 MHz (TSP)	11× (vs. PC @ 3.2 GHz; Matlab)	A parameterised GA core is presented where several parameters can be changed in the RTL description of the GA.
[NdMM07]	Mathematical functions	Pannictic (generational)			Cx: 2-point Mut: bit-flip	FPGA Xilinx Spartan-3	12.5 MHz	5× less number clock cycles compared to other hardware architecture	Uses a neural network hardware to compute the solutions' fitness.
[SF02]	TSP (≤ 1024)	Pannictic			Cx: Partially-mapped	FPGA Xilinx XCV812E (ADM-XRC PCI board)	40 MHz	13× to 50× (vs. PC @ 800 MHz; only crossover)	Only the crossover is implemented in hardware (represents between 15% to 60% of the execution time in software).
[MB97] [MB98]		Pannictic (generational)			Sel: roulette-wheel Cx: uniform Mut: bit-flip				Proposes a systolic design of a GA providing high throughput and unidirectional pipelining. Fitness function calculation not included in the architecture.
[SSS95]	Mathematical function	Pannictic (generational)	16	3	Sel: roulette-wheel Cx: 1-point Mut: bit-flip	3 FPGAs Xilinx XC4005 (BORG board)		14.7× less number clock cycles than software version	Proposes a general-purpose GA engine that can be reprogrammed (using VHDL)
<i>Pannictic steady-state GAs</i>									
[NBKHM13]	Mathematical functions; Parameter tuning in finger-vein biometrics	Pannictic (steady-state)	100	256	Sel: random, tournament Cx: 1-point, 2-point, multiple Mut: bit-flip, gene type	FPGA Altera Stratix II	50 MHz	102× (vs. soft-processor Nios II @ 50 MHz embedded in the FPGA; mathematical functions)	Proposes a hardware/software co-design methodology. Problem-dependent operations run in an embedded processor (Nios II).
[ATAH11]	Mathematical function	Pannictic (steady-state)			Sel: roulette-wheel, tournament Cx: 1-point, 2-point Mut: bit-flip	FPGA Xilinx Spartan-3		218× (vs. PC)	Uses neural networks to compute the fitness function.
[EPDP09]	Mathematical functions	Pannictic (steady-state)	32	10		FPGA Xilinx Virtex-II Pro	127 MHz	1.4× compared to a similar HW architecture	Architecture allows for the objective function to be updated through partial reconfiguration technology of the FPGA

continued . . .

Table 2.1 – ... continued

Work	Optimization problem	GA architecture	Population size	Chromosome size	Genetic operations	HW platform	HW clock frequency	Speedup	Observations
[VPP09]	Mathematical functions	Panmictic (steady-state)	2044	10	Sel: roulette-wheel	FPGA Xilinx Virtex-II Pro	100 MHz	$1.2 \times$; $3.3 \times$; $60 \times$ compared to other hardware implementations in the same FPGA family (in $60 \times$ only the fitness is in HW)	Implementation with 6 different optimization functions.
[PD05]	Mathematical functions	Panmictic (steady-state)	50	10	Sel: stochastic scheme;			$13 \times$ to $53 \times$ less number clock cycles compared to a serial uniprocessor	A pipelined hardware platform is described (not implemented). Uses the concept of postfix function evaluation for the fitness.
[VRGAR ⁺ 05]	TSP (≤ 100)	Panmictic (steady-state)	197		Sel: random Cx: edge recombination Mut: inversion	FPGA Xilinx Virtex-E	11 MHz to 15 MHz	$< 1 \times$ (vs. PC @ 1.7 GHz)	Hardware described in Handel-C (a HLS). 13 different hardware versions of the GA are implemented with increased performance.
[SSC ⁺ 01]	Set covering problem (94×520); 36-residue protein folding problem	Panmictic (steady-state)	256; 512	94; 70	Sel: random Cx: multi-point, uniform Mut: bit-flip	6 FPGAs Altera EPF81188A (Aptix AXB-MP3 board)(set covering); FPGA Xilinx XVC300 (protein folding)	1 MHz; 66 MHz	$2200 \times$ (vs. PC @ 100 MHz; set covering); $320 \times$ (vs. PC @ 366 MHz; protein folding)	GA operations are pipelined to achieve a throughput of 1 new chromosome per clock cycle. Complex fitness functions must be further pipelined to maintain throughput.
[KAM ⁺ 99]	Mathematical function; Knapsack problem	Panmictic (steady-state)	8; 16	8; 30	Sel: roulette-wheel Cx: 1-point; uniform Mut: bit-flip	FPGA ATT2C40 by Lucent Technologies	33 MHz; 8.95 MHz	$730 \times$; $250 \times$ (PC @ 333 MHz)	Hardware with a pipeline that never stalls.
<i>Distributed GAs (some works also include implementations of panmictic GAs)</i>									
[Jar12]	Knapsack problem (10000)	Distributed GA (≤ 14 nodes)	128 to 2048 per node	10000	Sel: tournament Cx: uniform Mut: bit-flip	14 nVidia graphic cards GTX 580	1.54 GHz	$< 781 \times$ (single CPU core @ 2.6 GHz); $< 35 \times$ (24 CPU cores @ 2.6 GHz)	Implementation in a multi-GPU cluster. Each node of the distributed GA evolves in a single GPU.
[THC11]	Path planning	Distributed GA (2 nodes)	50		Sel: tournament Cx: 1-point Mutation adapted to the problem	FPGA Altera		$90 \times$ (vs. PC @ 3.4 GHz)	Presents a GA to solve a global path planning for autonomous mobile robots navigation. Hardware/software co-design (fitness implemented in Nios II processor). Checks feasible solutions; special mutation.

continued ...

Table 2.1 – . . . continued

Work	Optimization problem	GA architecture	Population size	Chromosome size	Genetic operations	HW platform	HW clock frequency	Speedup	Observations
[VLM10]	Mathematical function	Distributed GA (≤ 2048 nodes)	128 per node		Sel: tournament selection Cx: 2-point Mut: change 1 gene	nVidia graphics card GTX 280	1.3 GHz	$< 2074\times$ (vs. PC @ 2.4 GHz; depends on problem size and number of nodes)	Several considerations are addressed about memory limitations of GPUs for implementing a distributed GA.
[PJS10]	Mathematical functions	Distributed (≤ 1024 nodes)	≤ 131072 (depends on the card and configuration of GA)		Sel: tournament Cx: arithmetic	nVidia graphics cards 8800 GTX; GTX 285 and GTX 260		$< 7000\times$ (vs. PC @ 2.66 GHz; 1 core; no migration; depends on number of nodes)	Presents a distributed GA implemented in GPUs using the CUDA framework. Acceleration is measured not considering the migration operation required in a dGA.
[JKFA06]	OneMax problem; Mathematical function	Pannictic (steady-state); Distributed (2 nodes)	128	64; 24	Sel: modified tournament Cx: uniform Mut: bit-flip	FPGA Altera Stratix	50 MHz	$50\times$ (OneMax); $20\times$ (function) (vs. soft-processor Nios embedded in the FPGA)	Fitness function implemented in software (Nios processor in the FPGA).
[TMS ⁺ 06]	TSP (51); Knapsack problem (64)	Pannictic (steady-state); Distributed (4 nodes)				FPGA Altera Cyclone			Proposes a pipelined based architecture for the GA and a parallel consisting of multiple concurrent pipelines.
[TY04]	Mathematical function	Pannictic; Distributed (4 nodes)	256	16	Sel: roulette-wheel Cx: multi-point, Mut: one-bit, multi-bit, masked, random	FPGAs Altera FLEX 6000 and FLEX 10k (per GA node).		$10.7\times$ (vs. PC @ 2.4 GHz; pannictic GA). Linear speedup for the dGA up to 4.	GA processor in a single board (with 2 FPGAs); dGA requires additional boards.
[CC00]	Set covering problem (19×63)	Distributed (2 nodes)	256		Sel: modified tournament Cx: 1-point, 2-point, uniform	2 FPGAs Altera EPF10K100A (PCI9030 board)	20 MHz		Distributed model with 2 processing nodes that achieves a $2\times$ speedup to find the optimum solution when compared to 1 node.
[GN95]	TSP (24, 120)	Pannictic; Distributed (4 nodes)	128, 256		Sel: roulette-wheel Cx: order-based Mut: swap	4 FPGAs Xilinx 4010s (per GA node) (Splash 2 board)	11 MHz	$< 11\times$ (vs. PC @ 125 MHz; pannictic GA)	dGA has shown a faster convergence when compared to the pannictic GA and with a dGA where the nodes do not interact.
[SWSS95]	Chip partitioning problem (100 to 500 components)	Pannictic (generational); Distributed (3 nodes)			Sel: roulette-wheel; Cx: uniform	several FPGAs Xilinx XC4010 (Armstrong III multicomputer)		$3\times$ (vs. PC @ 60 MHz; pannictic GA); $8.3\times$ (distributed GA)	Only the fitness function (that represents 95 % of the time in a software implementation) is implemented in the FPGAs.

continued . . .

Table 2.1 – . . . continued

Work	Optimization problem	GA architecture	Population size	Chromosome size	Genetic operations	HW platform	HW clock frequency	Speedup	Observations
<i>Cellular GAs</i>									
[VA10]	6 benchmark of discrete and continuous problems	Cellular	$\leq 512 \times 512$		Sel: roulette-wheel Cx: 2-point	2 nVidia graphic cards GTX 285	1.48 GHz	$8 \times$ to $771 \times$ (vs. PC @ 2.67 GHz)	Presents a multi-GPU desktop platform (2 PCs) for implementing a cellular GA.
[TA95]	Disc scheduling problem	Cellular			Sel: order-based Mut: swap	ASIC (1.0 μm)			System capable of solving the problem in real-time (2 ms). Details of the cellular structure are not provided.
[TAH94]	Image processing in a vision system	Cellular			Sel: tournament Cx: 2-point	ASIC (1.5 μm)	8 MHz		Implementation of a cellular GA in a ASIC. Only details of a processing element are provided (not from the cellular structure).
<i>Other algorithms based on the GA</i>									
[JC08]	OneMax problem (32); Mathematical function	Compact GA		32		FPGA Xilinx Virtex-5	280 MHz (2×2 configuration)	$> 2 \times$ compared to others HW implementations of compact GA (only one node)	Proposes an hardware architecture with a cellular like structure for the compact GA.
[ZMC07a] [ZMC07b]	Mathematical functions	Novel algorithm called OIMGA				ASIC (0.13 μm)	300 MHz		A novel GA is proposed (OIMGA), that includes a global and local search. Work targets a algorithm optimized to hardware by using a single optimum solution thus reducing memory.

of the GA and the population size used in the implementation as well as the optimization problem to be solved. Additionally, it mentions the hardware platform and the acceleration achieved by the algorithm. Empty fields in the table mean that those details are not provided by the authors.

Besides FPGAs and ASICs implementations, Table 2.1 also includes a few works that use GPUs as a hardware platform to execute GAs. Although these devices use a processing unit to execute the algorithm, which is programmed through software, they have been successfully used to implement GAs by exploiting their highly parallel computing structures.

We analyse the GA custom hardware architectures of the existing implementations mainly concerning the organization of the population (panmictic and decentralized and their variants). The implementations of the operations of the GA are problem-dependent and, in general, straightforward to implement in hardware as they require simple bit manipulation. Additionally, more specific operators required by a particular optimization problem must always be designed to target that application and thus a review of those operators is impractical. Nevertheless and despite the operators that constitute the GA, the organization of the population plays an important role in the performance of the hardware. The next subsection presents an overview regarding this topic.

2.3.1 GA architectures

As discussed previously, the population of a GA can be organized and accessed in different ways which impacts in the hardware organization of the computing, control and memory elements of a custom computing machine for genetic algorithms. The following subsections present an analysis of such architectures and draws considerations about their advantages and disadvantages from the review performed in Table 2.1.

2.3.1.1 Panmictic - generational

Figure 2.9 presents a generic hardware architecture for a panmictic GA with a generational reproductive step. At each iteration (or generation), the GA processor must be able to access any solution present in the population memory to build a new population that will replace the existing one. Therefore, the processor must apply the genetic operations on the existing solutions to build a complete new population that will replace the entire old population. Afterwards and in a single step, the population memory is updated with the new population, thus eliminating the previous one.

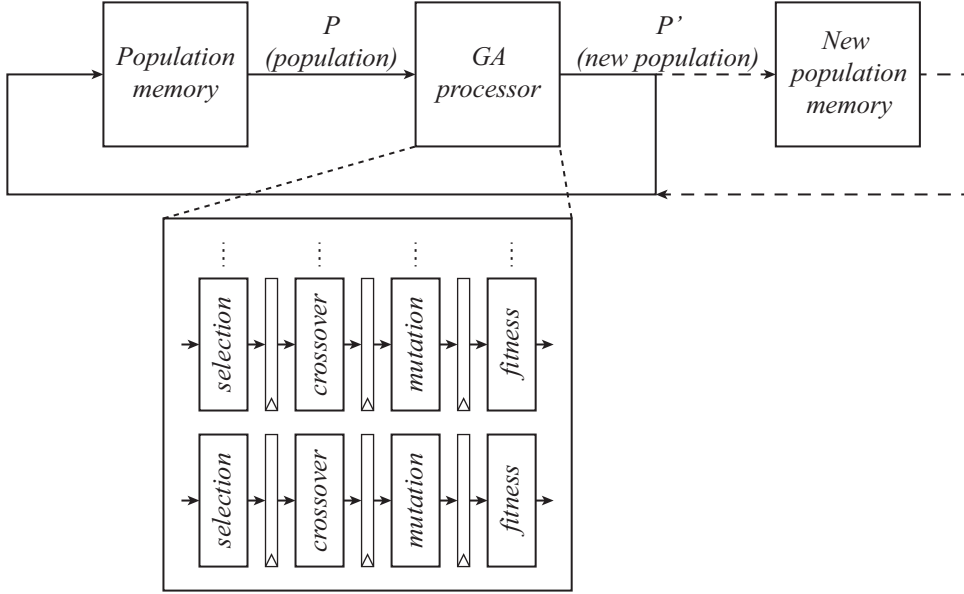


FIGURE 2.9: Example of a hardware architecture for a panmictic generational GA. In a generation step, a GA processor creates a new population P' by processing the current population P . The genetic operations can be parallelized and pipelined.

As the operations executed by the GA processor are repeated several times in a single iteration, it is possible to parallelize them to improve the performance of the hardware. Furthermore, these operations can be pipelined as shown in Figure 2.9. This concept is explored in [GTL13] and [GTL14], where the GA processor can have different levels of parallelism and the operations are pipelined.

Although increasing the parallelism is appealing, a clear drawback appears in what concerns the population memory access. Since the memory bandwidth is limited, the level of parallelism will be limited by the data that this memory provides to keep all the parallel elements working. An architecture with a parallelism level capable of producing a new population in a single step is possible as it is shown in [NdMM07]. Nevertheless, all the memory elements must be accessed simultaneously which may be possible to implement only for a small population where the solutions can be kept in registers. While the authors of this work do not provide the population size, it is clear that for realistic GA implementations, where dozens or hundreds of solutions may be required, the memory access bottleneck imposes a practical limit in the parallelism level.

Another issue with this architecture is that a second memory is required to keep the new population since the GA processor requires several steps till it builds the new population completely (a parallelism level to generate the entire population in a single step is unrealistic). Therefore, an operation that replaces the population memory contents with the new generated solutions must be performed as exemplified in [GN95]. To avoid

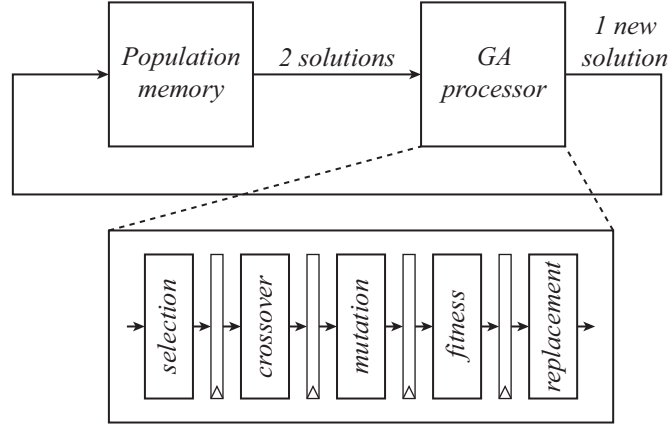


FIGURE 2.10: Example of a hardware architecture for a panmictic steady-state GA. In a generation step, a GA processor creates a single solution by processing 2 solutions in the population. All the operations can be pipelined.

this, the work proposed in [VPP09] uses a single memory divided in two parts to store both populations. From one generation to another on the GA, the hardware swaps the accesses to the memory avoiding this way the need to explicitly copy the contents from one memory to another.

With a generational GA it is clear that it is possible to build an hardware architecture that exploits parallelism of the operations of the algorithm. However, to be effective in terms of performance, the memory architecture must be designed in a way to match the bandwidth requirements of the processing datapath.

2.3.1.2 Panmictic - steady-state

Figure 2.10 depicts an example of a hardware architecture of a panmictic GA with a steady-state reproductive scheme. In this case, in a generation of the algorithm a single solution is created by the GA processor that will replace one existing solution in the population memory. Therefore, and by contrast with the generational GA, the GA processor can write the new solutions directly in the population memory and thus no additional memory is required. In addition, this can take advantage of pipelined implementations to improve performance.

One of the main concerns with this architecture is the need for a high memory bandwidth to exploit the pipelined circuit. In a single step, and to avoid stalls of the pipeline, this memory would require three simultaneous accesses: reading the two selected solutions and writing the new one. To overcome this, several works propose that one of the selected solutions in a given generation of the GA comes from the previous generation [SSC⁺01,

TMS⁺06, CC00]. This way, only one solution is required to be read from the memory instead of two.

Additionally, it is also common in this architecture to use simple algorithms in the selection operation to cope with the single clock cycle requirement. Example of such algorithms are the random selection [SSC⁺01, TMS⁺06] and tournament selection [CC00]. Moreover, often with this selection schemes the replacement operation follows a strategy where the worst selected solution may be replaced by the new solution if this has a better fitness value. This technique is performed to improve the convergence of the GA. More complex selection algorithms, like the roulette-wheel, are possible to implement while preserving a good pipeline performance [KAM⁺99]. However, this would require a considerable increase in the hardware resources as the population size increases.

The remaining operations of the GA must also be implemented in hardware as fast hardware blocks to improve the pipeline performance. The crossover and mutation applied to a binary representation of solutions are efficiently implemented in hardware. Nevertheless, the fitness evaluation function, which is problem-dependent, may compromise the performance of the pipeline. The work proposed in [SSC⁺01], presents a general-purpose steady-state GA hardware engine with a 6-stage pipeline where 1 stage is dedicated for the fitness evaluation. For implementing a fitness evaluation circuit for the set covering problem, the hardware resources available in the FPGA are sufficient to maintain the pipeline without stalls. Nevertheless, for solving a 36-residue protein folding problem, the fitness evaluation requires a much more elaborated function, which results in a pipeline initial interval of 36 for this circuit. Although several parallel fitness circuits could be used to avoid the stall of the GA pipeline, more hardware resources would be required.

Although we have illustrated a pipelined hardware architecture for the steady-state GA, some works do not follow this idea [VRGGAR⁺05, EPDP09]. Instead, the operations of the algorithm are executed sequentially by different custom hardware blocks. This approach may be interesting as a trade-off between hardware project complexity and overall performance of the circuit.

2.3.1.3 Distributed

In a distributed GA the population is divided into small subpopulations that evolve independently among them; during the evolution of the algorithm, these subpopulations must occasionally exchange solutions. Figure 2.11 shows a generic hardware architecture for a distributed GA with 4 subpopulations. As expected, the processing is easily parallelized by several identical processing nodes (dGA nodes), each one holding its own

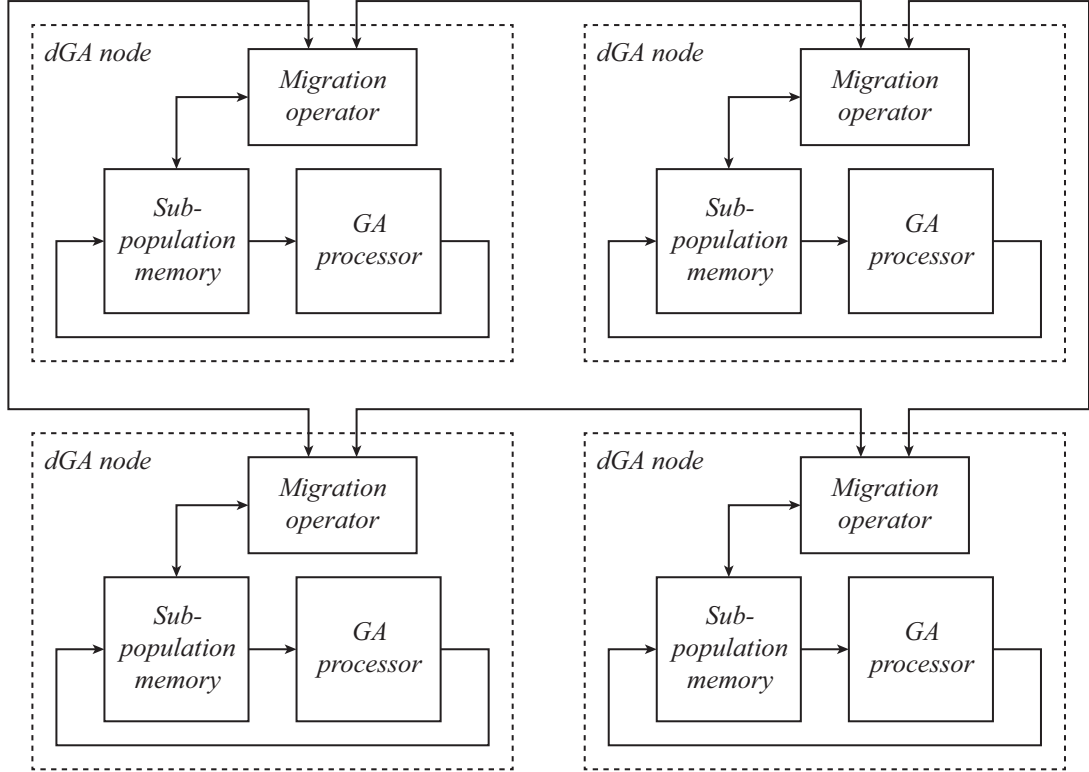


FIGURE 2.11: Example of a hardware architecture for a distributed GA (dGA) with 4 nodes connected in ring. Each dGA node is responsible to evolve a subpopulation of the algorithm. The dGA nodes must occasionally migrate solutions among them.

subpopulation memory and GA processor. Additionally, the dGA node has a migration operator that ensures the exchange of solutions among the dGA nodes. In this example, the 4 nodes are connected in a ring topology, although other topologies are possible.

It should be noted that the GA processor and the subpopulation memory of a dGA node are identical to a panmictic GA (either generational or steady-state). Therefore, the same ideas discussed for the panmictic GAs are also applicable to the implementation of a dGA node.

Several works that implement a panmictic GA, also implement a distributed GA as essentially this can be built by replicating several panmictic GAs. By comparison with a panmictic GA (or a dGA with 1 node), these works show a linear speedup with the number of dGA nodes [CC00, TY04], or an almost linear speedup [SWSS95] (for this particular example, $2.8\times$ for 3 nodes). All of these examples are implemented in FPGA devices and do not exceed more than 4 dGA nodes.

More recently, GPUs devices have been used with great success to implement distributed GAs. By comparison with FPGAs, these implementations usually deal with considerable larger populations that reach dozens of thousands of solutions [VLMT10, Jar12, PSJ10],

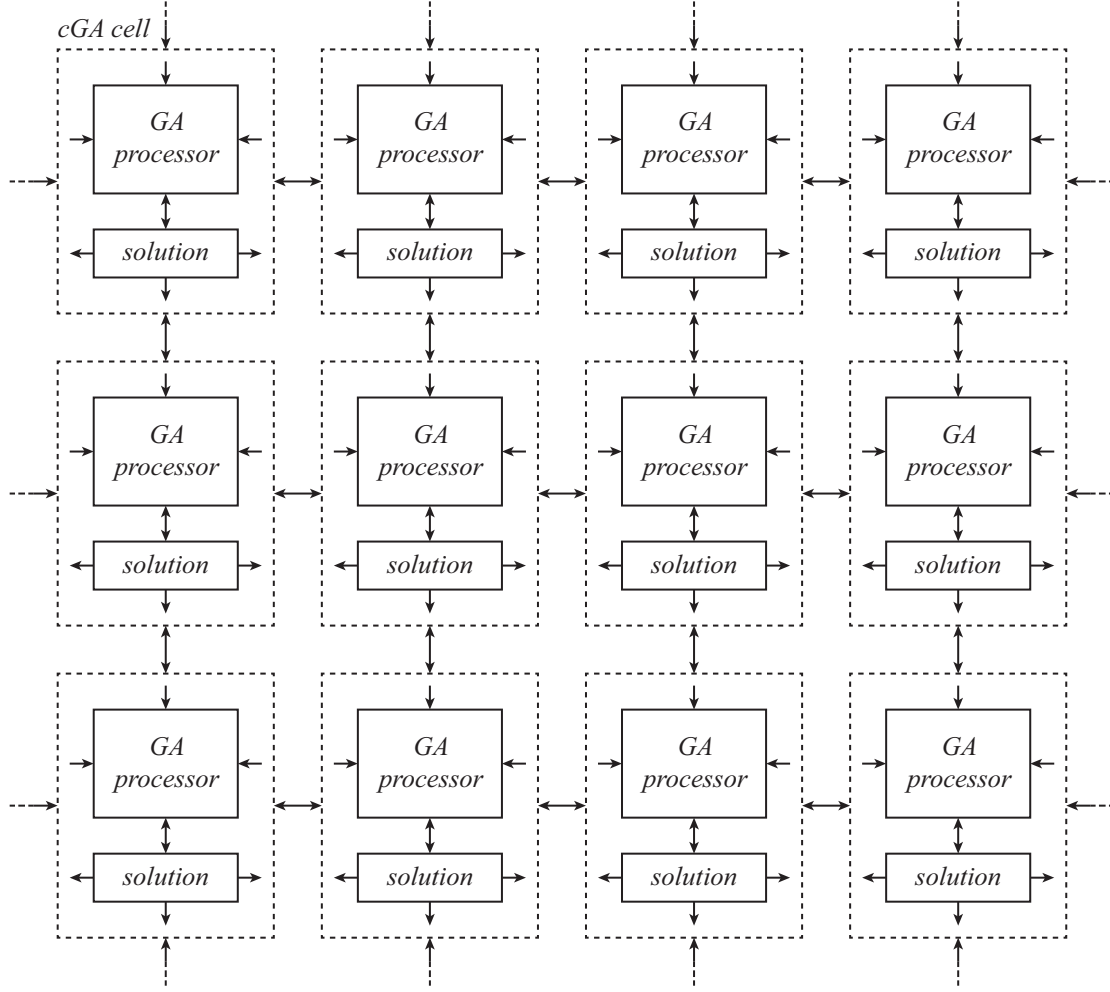


FIGURE 2.12: Example of a hardware architecture for a cellular GA (cGA). Each cGA cell only processes solutions present in itself and in cells directly connected to it.

or solutions that require more memory for their representation [Jar12]. Moreover, these works easily achieve a high level of parallelism with a few hundreds of dGA nodes in a single GPU device [VLMT10, PJS10]. The speedup obtained with these implementations easily reaches a few thousands over a single CPU.

It should be stressed that in a dGA the nodes must exchange solutions among them, otherwise the algorithm will not behave as expected. In [PJS10] and [PSJ10] it is presented a dGA with a maximum of 1024 nodes. Nevertheless, the acceleration of this algorithm is measured not considering the migration operation and thus the results cannot be considered as accurate for the distributed GA (but instead for a panmictic replicated 1024 times).

2.3.1.4 Cellular

Figure 2.12 presents an example of a possible hardware architecture of a cellular GA where the solutions are distributed over a regular grid. A cGA cell, which in this example is constituted by a single solution and a GA processor, is replicated several times to form a 2D shape as depicted in the figure. The GA processor generates new solutions by applying the operations of the GA to solutions present in the vicinity cells. Therefore, each processor can access for selection five different solutions: one in the cell where the processor is and four in the neighbour cells which the processor can communicate with. If selected by the replacement strategy, the generated solution is written by the processor in the solution memory present in the cGA cell, therefore deleting the existing solution.

Although the cGA is a very interesting metaheuristic that can be exploited in a hardware implementation of a GA, especially in what concerns the parallelization of the operations, it has not been widely adopted and few works can be found in the literature. The works proposed in [TAH94, TA95] present a custom ASIC that implements cGAs to solve a image processing problem in a vision system and a disc scheduling problem. Nevertheless, these works, which are from the 1990's, do not provide much detail about the parallelism achieved and how this improves the acceleration of the algorithm in comparison with other architectures.

Recently, a cGA has been implemented in a multi-GPU platform for several benchmark optimization problems [VA10]. In this implementation, each solution of the algorithm is associated to a thread of the GPU, thus achieving a high level of parallelism. Results show that by comparing a implementation of the cGA in a CPU to a single-GPU platform, the speedup ranges from 8 to 771. Nevertheless, the lowest value of this acceleration is for a population of 64 solutions, whereas the highest value is for a population of 262144 solutions. Therefore, it is clear that the acceleration achieved by the GPU does not scale in proportion to the number of solutions (more threads); an increase of $4096\times$ in the population size leads to a best value acceleration less than $100\times$.

2.3.1.5 Variants of GAs

There are other algorithms that somehow resemble the GA and can be efficiently implemented in hardware. The best known is the compact GA that requires a single vector to keep the complete information present in a population of a canonical GA using a binary representation. In this vector, it is kept a list of probabilities values for each gene of the (non-existing) solutions in the population. The algorithm starts with a vector with 0.5

of probability for each gene and, as it evolves, these values will converge to 0 or 1 which represent the final solution. Details of the compact GA can be found in [HLG99].

By comparison to the GA, the compact GA needs less memory as it only requires a single memory vector (for the probability vector) instead of a complete population of solutions. Therefore, the memory accesses required are much easily handled by a GA processor since less data needs to be accessed. As it happens with the decentralized GAs, in the compact GA the population can also be divided in smaller subpopulations (in the case, the probability vector) each one assigned to a processing node. In [JC08] a hardware architecture for a compact GA with a cellular like structure (with 4 nodes) is built that resembles a distributed GA.

Although the compact GA has been widely adopted, especially in the research field of evolvable hardware, it does not provide the same search power as the GA even with several improvements [GVK04]. This happens since this algorithm was developed to mimic the panmictic steady-state GA with a binary tournament selection and a uniform crossover operator for solutions with binary representation [HLG99]. Therefore, the algorithm is a particular case of the GA, where different operators cannot be used.

In [ZMC07a, ZMC07b] it can be found another algorithm inspired in the GA, named optimal individual monogenetic algorithm (OIMGA), that targets a hardware implementation. This algorithm uses a single solution to be evolved, thus also reducing the memory when compared to the GA, and it is based on an interaction of a global and local search. Although this algorithm has the word ‘genetic’ in its name and the authors claim that is a *new* GA, there is not a clear association with this algorithm and the population-based metaheuristic GA.

The goal of this work is to develop a custom hardware architecture that supports the execution of GAs in its canonical form, or variations of it as it is the case of the cellular GA, and are well-known from the literature. Therefore, the ideas presented in this section are not further exploited.

2.3.2 Where is the bottleneck?

The main reason to implement a GA in dedicated hardware is to accelerate it. Therefore, it becomes important to know which of the algorithm’s operation is the most computationally demanding so that it can be further improved in hardware.

From the literature review, we cannot conclude that there is a particular operation in the GA that is more demanding than the others. For instance, a software profile to the GA implemented in [SWSS95] reveals that 95 % of the execution time is in the

fitness function evaluation for a chip partitioning problem. On the other hand, the work presented in [SF02] reveals an execution time between 15 % to 60 % for the crossover operation for the travelling salesman problem. In both works, only the critical operation has been implemented in hardware with the goal to accelerate the algorithm.

As we have already discussed, in [SSC⁺01] it is presented a general-purpose hardware architecture for executing a panmictic steady-state GA, with a specialized pipelined architecture. Implementing the hardware block for the fitness evaluation function for the set covering problem requires a simple circuit that does not compromise the pipeline performance. Nevertheless, for a protein folding problem, a much more complex function to compute, the pipeline has a clear stall due to the fitness evaluation block (which takes several clock cycles to finish).

Another example of how the complexity of a operator can impact in the performance of the hardware is presented in [KAM⁺99]. In this case the selection operator chosen is the roulette-wheel algorithm which needs to unroll its main loop and parallelize it to achieve a pipelined circuit without stalls. Once again, this imposes limits in the hardware needed to implement the circuit where a larger population would require more hardware resources.

Therefore, the question *where is the bottleneck in a genetic algorithm?* does not have a straight answer. It is highly dependent on the operators used by the GA that in turn depend on the optimization problem. Many different representations and operators exist for the GA, as we have discussed in Section 2.2.1. Although it is common to associate a genetic algorithm to solutions coded with a binary representation where simple crossover and mutation operators are applied, this is not always the case. Additionally, it may be even necessary to include special operators in the algorithm so that the GA provides feasible solutions or to improve the quality of the solutions (e.g. combine a GA with a local search operator). In this thesis we will see two examples of real-world problems that have these requirements.

Regardless of what has been discussed previously, it is clear that a hardware architecture that exploits parallelism of operations will increase the potential to accelerate an algorithm, although requiring more hardware resources. In this regard, the decentralized models of GAs (distributed and cellular) are strong candidates for parallelizing all the operations of the algorithm as we have discussed in Section 2.3.1.

2.3.3 Acceleration

As stated before, the implementation of GAs in specialized hardware is motivated by the need to accelerate the search metaheuristic and provide solutions in less time than conventional CPUs. Therefore, the acceleration value is a common figure provided by hardware implementations of GAs, which tries to quantify the quality of the implementations (see Table 2.1). However, a direct comparison of these data to understand which may be the best implementations is not advisable as there are many factors that influence these figures.

In the majority of the works, the hardware execution time is compared to a software counterpart of the GA running in a personal computer (PC) [DDT08, GTL13, GTL14, SSC⁺01, KAM⁺99, VRGGAR⁺05, ATAH11, TY04, GN95, SWSS95, VLMT10, Jar12, VA10]. Nevertheless, some works execute the software in an embedded processor which exists in the same hardware platform as the GA engine (typically an FPGA) [FKK⁺10, NBKHM13, JKFA06]. In this case, it is expected to achieve a higher acceleration as these processors execute in a substantially lower clock frequency. Besides a software to hardware execution time comparison, others methods are adopted like counting the number of clock cycles in both software and hardware, irrespective of the clock frequency used [SSS95, PD05], or a direct comparison with other hardware implementations [NdMM07, EPDP09, VPP09].

Although most of the hardware architectures (not considering GPUs) are described at the register-transfer level (RTL) using a standard hardware description language (HDL), high-level synthesis (HLS) tools were also used to build the hardware [VRGGAR⁺05, FKK⁺10, GTL14, GTL13]. The same happens with the software implementations, where different languages with distinct performances are used like C/C++ [SSC⁺01, JKFA06] or Matlab [DDT08]. In both hardware and software implementations the different tools and techniques used to implement the algorithm can lead to incomparable acceleration figures among all the works.

Even though it is difficult to compare the hardware performance of different works, it is possible to draw relevant conclusions from Table 2.1. A steady-state GA usually shows better accelerations when compared to a generational GA. From these works, clearly the ones exploiting a high-performance pipeline circuit (each pipeline stage representing an operation of the GA) are the ones that achieve the best results with acceleration values of a few hundreds or even more [SSC⁺01, KAM⁺99]. Nevertheless, these architectures have limitations in the operations of the GA as discussed in Section 2.3.1.2. The distributed and cellular GAs, that can have several processing nodes running in parallel, clearly increase the acceleration as the parallelism also increases. This is evident in the works

that implement a GA in FPGA and a direct comparison is made between one and several processing nodes [CC00, TY04, SWSS95]. Moreover, recent GPUs implementations exploit a very high-level of parallelism in these GAs with thousands of threads processing in parallel [VLMT10, Jar12, VA10]. The results show accelerations values of a few thousands.

Decentralized GAs offer the possibility of having several processing nodes, thus improving the algorithm execution time. More interestingly, although the generational GAs can explore parallelism, there is not evidences that they provide a better performance than steady-state GAs (that do not parallelize the GA operations) or decentralized GAs. We can only speculate about this since although parallelism exists, the population is in a single memory which leads to a bottleneck in the memory access required to feed all the parallel units. Distributed and cellular GAs can easily overcome this problem by having several memories to keep the population.

2.3.4 General considerations

There have been several hardware implementations of general-purpose GAs that claim that can be used generically to solve any problem. However, it is clear that at least one of the GA's operations needs to adapt to the optimization problem: the fitness function evaluation. To overcome this, neural network techniques have been used to estimate efficiently the fitness of solutions [NdMM07, ATAH11]. Nevertheless, this approach requires a training phase of the network for each different instance of the problem and it has been applied only to simple optimization problems (mathematical functions). Another approach, although obvious, is to develop a new hardware block to calculate the fitness function for each different optimization problem [FKK⁺10, SSC⁺01]. Therefore, the concept of general-purpose architecture for a GA often fails as it is not possible to define *a priori* the operators of the algorithm. For instance, in [DDT08] it is proposed a general IP core for the GA, but for implementing the crossover operator for solving a travelling salesman problem (that requires a path representation) new hardware needs to be specified.

A GA relies heavily in random numbers. The most common implementations of random number generators (RNG) in hardware GAs are linear feedback shift registers (LFSR) [JKFA06, NdMM07] and cellular automata (CA) [CC00, SSC⁺01, FKK⁺10]. These RNGs structures are simple to implement in hardware and provide fairly good quality random numbers. Although other works implement better RNGs, like a true RNG based on jitter effects [NBKHM13], it is not demonstrated that they improve the GA.

It is often the case that hardware implementations of GAs use very small populations and solutions that require just a few bits (c.f. Table 2.1). Nevertheless, a GA is a particular useful metaheuristic to solve complex optimization problems that possess a huge search space. This means that a large number of bits are required to represent a solution in the GA and the population size must be large enough (e.g. hundreds of solutions) to create diversity of the search space. Although small problems can be used to demonstrate a GA hardware architecture, it is fundamental that these architectures can deal with problems requiring a large solution representation.

2.4 Summary

In this chapter we have introduced the GA, a population-based metaheuristic. Different classes of GAs were distinguished with respect to the structure of the population of the algorithm. Subsequently, we have reviewed generic hardware architectures for the different classes of the algorithm.

With a panmictic GA (both generational and steady-state) the population is kept in a single memory. We have discussed that although it is possible to parallelize the operations of this algorithm (generational GA), the memory access required to feed all the processing nodes is likely to represent a bottleneck. In addition, we have also seen that generating a single solution per iteration of the algorithm (steady-state GA), an efficient pipelined circuit is possible to obtain. Nevertheless, in this case the hardware implementation of the algorithm's operations must be balanced not to compromise the pipeline.

Decentralizing the population in a GA (distributed and cellular GAs) has a great potential for building parallel hardware architectures capable of accelerating the algorithm. Additionally, since the population is structured, several independent memories can hold subsets of the population that can be accessed by parallel processing units. Therefore, no memory access bottlenecks are introduced as the level of parallelism increases in these architectures, as it happens with a panmictic GA. In particular, cellular GAs have the potential for the highest parallelism due to their physical distribution of the solutions, where a processor can exist for each solution.

In Chapters 3 and 4 we will present a novel cellular GA architecture suitable for dedicated hardware implementation, mainly an FPGA. Furthermore, Chapter 5 will present a design flow based on a C++ specification of the problem-dependent operations of the algorithm so that the architecture can be rapidly customized to solve different optimization problems.

Chapter 3

A scalable processor array for cGAs acceleration

3.1 Introduction

In the previous chapter genetic algorithms (GAs) were introduced together with a review of dedicated hardware implementations of this metaheuristic. It was concluded that parallelizing the operations of the algorithm can be achieved by using decentralized populations of the algorithm, where subsets of solutions are kept in independent memories that are accessed by processing units. Therefore, the algorithm can be accelerated by increasing the parallelism without compromising the performance of the hardware architecture.

In this chapter a scalable processor array architecture is presented that accelerates the execution of cellular genetic algorithms (cGAs). Section 3.2 starts by presenting the hardware architecture followed by a classification of the cGA supported by the hardware. Additionally, it is clarified how the processor array can be used to support the execution of other population-based metaheuristics. In Section 3.3 an architecture simulation with a system-level model of the hardware is performed to study the behaviour of the algorithm with different levels of parallelism and configurations of the processor array. Finally, in Section 3.4 a hardware implementation in an FPGA is presented to solve the travelling salesman problem.

3.2 The architecture

The goal of this work is to propose an infrastructure capable of creating dedicated processing units, with a high level of parallelism, to support the execution of genetic algorithms. Among the categories of GAs analysed during Chapter 2, we have concluded that the ones where the population of solutions is decentralized, allow to exploit the parallelism afforded by dedicated hardware implementations, without severe bottlenecks in the memory access to the population. Essentially, by decentralizing the population it is possible to have several processing units, each one accessing to its subset of solutions that are placed in different hardware memory blocks. Between the distributed GA and cellular GA, the last one provides a higher distribution of the solutions where, in the limit, a processing unit can exist to each solution. Additionally, the target hardware device of our architecture is an FPGA device, which currently (like Xilinx Virtex UltraScale [Xild]) have up to a few thousands of independent memory blocks. Therefore, we have elected the cGA to be implemented in dedicated hardware, where subsets of the population are kept on those memory blocks, while processing units access to them to compute the algorithm.

This section presents the architecture proposed in this thesis to implement cellular genetic algorithms in parallel computing platforms, targeting digital reconfigurable technologies. Figure 3.1 depicts the overall structure of this architecture, which is constituted by replicating and connecting two different blocks: memories (MEMs) and processing elements (PEs). Each memory holds a subset of solutions (or subpopulation) of the algorithm and is shared between two adjacent PEs. In turn, each PE can access to four memories, thus accessing to the solutions, and it is responsible to apply the operations of the genetic algorithm to those solutions or other procedures necessary for the execution of the cGA on the proposed infrastructure.

This architecture implements a cGA since each solution can only interact with a predefined number of solutions in its neighbourhood. Additionally, throughout the structure there is an overlap of different solutions' neighbourhoods which imposes an implicit mechanism of migration of solutions.

The complete hardware architecture presents a toroidal shape where both top/bottom and left/right sides of the structure are connected, so that all the memories connect to two PEs, thus leading to a complete regular structure. Nevertheless, it is also possible to build a non-toroidal configuration, where the memories placed in the borders of the array only connect to one PE.

As it is evident, each PE can work independently from the others while it processes its solutions. Therefore, this architecture relies essentially on increasing the number

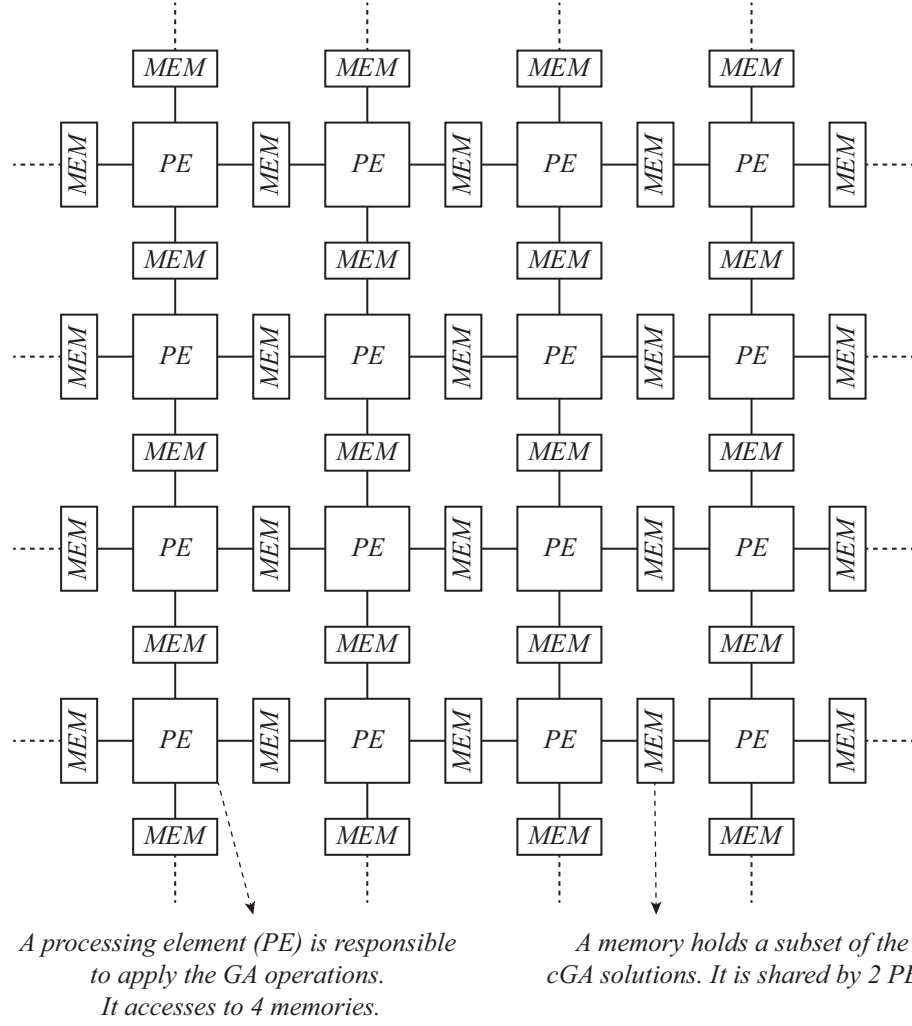


FIGURE 3.1: Overall architecture of the proposed scalable processor array for cGAs.

of PEs to achieve more parallelism, thus accelerating the algorithm execution time by increasing the rate of evolution of the population. Considering a regular shape in the cGA architecture, the number of PEs (N_{PE}), which is the potential acceleration of the algorithm, is defined by

$$N_{PE} = N_{PE_{rows}} \cdot N_{PE_{cols}} \quad (3.1)$$

where $N_{PE_{rows}}$ and $N_{PE_{cols}}$ denote, respectively, the number of PEs in the rows and columns of the array that builds the cGA architecture.

The number of memory blocks required to build the architecture is given by

$$N_{MEM_{toroidal}} = 2 \cdot N_{PE} \quad (3.2)$$

$$N_{MEM_{non-toroidal}} = 2 \cdot N_{PE} + N_{PE_{rows}} + N_{PE_{cols}} \quad (3.3)$$

for respectively a toroidal or non-toroidal configuration.

It is well established that in a GA (or cGA) increasing the population size leads to a convergence to better solutions, although slower because a larger solution space needs to be explored. The number of solutions used by a given cGA is a parameter that is previously studied to ensure that the algorithm obtains a solution with a desired quality within a given execution time. Moreover, as our hardware architecture targets an FPGA, the population size may be constrained by the available memory blocks in the device. In the hardware architecture, as the number of PEs increases the number of memories must also increase, which in turn leads to a growth in the total number of solutions. Consequently, there is a maximum number of PEs that our architecture can have to apply a cGA with a predefined population size. The following equations show the relationship between the total number of solutions in a, respectively, toroidal and non-toroidal configuration of the cGA architecture and the number of PEs

$$\begin{aligned} N_{solutions_{toroidal}} &= k \cdot N_{MEM_{toroidal}} \\ &= 2 \cdot k \cdot N_{PE} \end{aligned} \quad (3.4)$$

$$\begin{aligned} N_{solutions_{non-toroidal}} &= k \cdot N_{MEM_{non-toroidal}} \\ &= k (2 \cdot N_{PE} + N_{PE_{rows}} + N_{PE_{cols}}) \end{aligned} \quad (3.5)$$

where k denotes the number of solutions in each memory. For a given number of solutions, the maximum level of parallelism is achieved by a toroidal configuration with one solution per memory ($k = 1$), and it is equal to half the total number of solutions. Therefore, we can say that our architecture can have a maximum of a PE for each two solutions in the population.

Since the proposed scalable processor array for cGA targets FPGA devices, the memory blocks are implemented with the built-in memory blocks of these devices (BRAMs) which have a predefined capacity. Therefore, depending on the size of the representation of a solution and how many solutions a subpopulation has, the number of BRAMs and their utilization can vary in the implementation of the architecture. We define the memory usage of a solution ($solution_{usage}$) as the amount of space that a solution requires to be kept in a single BRAM. For instance, if a BRAM has a total of 1024 words and a solutions requires 4 words, the $solution_{usage}$ is equal to $4/1024$; instead, if a solution requires 1100 words, the $solution_{usage}$ is now equal to $1100/1024$ which represents more than one BRAM to keep a single solution. The amount of BRAMs required to keep a complete subpopulation is given by $\lceil k \cdot solution_{usage} \rceil$, where k represents the number of solutions per subpopulation. It should be emphasized that the $solution_{usage}$ must include complete memory words. For instance, if a solution requires 40 bits and the BRAM is configured to work with 1024 words each with 32 bits, the $solution_{usage}$ is

equal to $(1 + 1)/1024$; whereas in a configuration of the BRAM of 2048 words each with 16 bits, the $solution_{usage}$ is now equal to $(2 + 1)/2048$.

In an FPGA implementation, the number of BRAMs required to build the architecture is given by

$$N_{BRAM_{toroidal}} = 2 \cdot N_{PE} \cdot \lceil k \cdot solution_{usage} \rceil \quad (3.6)$$

$$N_{BRAM_{non-toroidal}} = (2 \cdot N_{PE} \cdot N_{PE_{rows}} + N_{PE_{cols}}) \cdot \lceil k \cdot solution_{usage} \rceil \quad (3.7)$$

for respectively a toroidal or non-toroidal configuration.

One of the main advantages of the proposed architecture is its scalability since the number of PEs can be adjusted, within certain limits, to fulfil the desired requirements. From one side, a higher number of PEs leads to an improved throughput but it requires more hardware resources. Situations where harsh real-time constraints are imposed may take advantage of our architecture by increasing the parallelism. On the other side, if there are limitations in the hardware resources available, it is possible to decrease the parallelism to a level where both computation time and used hardware resources are satisfied. For example, a cGA with 128 solutions can be implemented with a 8×8 configuration (64 PEs) where each memory holds a single solution (toroidal shape), or by a 2×2 configuration (4 PEs) with 16 solutions per memory.

Below we summarize the main characteristics of the proposed cGA architecture.

- *Scalable:* The number of PEs can be varied to achieve different levels of parallelism. Nevertheless, since the number of memory blocks grows in proportion to the number of PEs, the amount of solutions per memory must be decreased as the parallelism increases so that the population size is kept near constant (Equation (3.4) and (3.5) must be satisfied, respectively, for a toroidal and non-toroidal configuration). This imposes a maximum limit in the level of parallelism (number of PEs) equal to half the population size, for a toroidal configuration, where only one solution exists per memory. Scalability of the architecture is limited by the number and configuration of the PEs in the structure's array, and by the number of solutions per memory so that the equations aforementioned are satisfied.
- *Regular:* A cGA with a toroidal structure is built by replicating a regular structure where one PE connects to 4 memories and one memory is shared by 2 PEs. Therefore, no additional complexity exists for building a cGA with different levels of parallelism since only these two blocks need to be replicated as desired. For cGAs with a non-toroidal structure, the memories placed in the borders only connect to one PE, which can be easily constructed starting from a toroidal configuration.

- *High memory bandwidth:* Changing the level of parallelism does not introduce any potential memory access bottleneck in the architecture. A single PE always has access to four independent memories, regardless of the number of PEs. Therefore, since the number of memory blocks increases in proportion with the number of PEs, the global memory bandwidth of the hardware is naturally adapted to the parallelism level. Additionally, the hardware architecture of a PE can be designed so that the memory bandwidth provided by the four memories is used to efficiently feed their circuits (e.g. parallel structures or pipelined circuits) thus improving the overall performance of a PE.
- *Shared memories:* Although the main goal of the shared memories of the cGA architecture is to keep the problem's solutions, which is essential for the algorithm, they can also be used to store additional data. A typical example is the problem instance parameters (often a large amount of data) required to compute the fitness function. Nevertheless, as each memory block is seen by the two PEs connected to it in different positions (e.g. for one PE is the top memory and for the other PE the bottom memory), consecutive PEs in the array see the same memory contents in different memories' positions. To overcome this, the data can be simply duplicated for each top/bottom and left/right pairs of memories so that all the PEs behave similarly while reading this data. Additionally, the shared memories can also be used exclusively by a single PE to keep its own data, or by passing information to the adjacent PEs if, somehow, this is required.
- *Independent PEs:* The operation of the array of processing elements is globally asynchronous and thus each PE works at its own pace and does not need to synchronize with other nodes. As, in general, the processing time for an iteration in a PE will depend on the solution data, the asynchronous operation is the natural expected behaviour. One important advantage of an asynchronous operation of the array of PEs is the potential reduction of simultaneous memory accesses and consequently the reduction of stall states necessary to overcome memory access collisions.
- *Toroidal or non-toroidal:* With a toroidal configuration of our cGA we guarantee a complete regular structure where all the solutions possess the same neighbourhood structure to interact with. Nevertheless, to build this configuration in hardware long wires are required to form the toroid, which becomes more critical as the parallelism increases. To avoid this, a non-toroidal configuration can be built that ensures that critical data paths are not introduced as the cGA is scaled in number of PEs. As a result, the solutions placed in the border memories present a different neighbourhood structure from the remaining. Despite a toroidal or non-toroidal

Algorithm 2 Pseudo-code of a canonical cGA.

```

1:  $P \leftarrow \text{Genetate\_Initial\_Population}()$ 
2:  $\text{Evaluate}(P)$ 
3: while not  $\text{Termination\_Condition}()$  do
4:   for  $cell \leftarrow 1, \text{Population\_Size}$  do
5:      $neighbours \leftarrow \text{Calculate\_Neighbourhood}(\text{position}(cell))$ 
6:      $parents \leftarrow \text{Select\_Parents}(neighbours \text{ and solution in position}(cell))$ 
7:      $new\_solution \leftarrow \text{Apply\_Crossover}(parents)$ 
8:      $new\_solution \leftarrow \text{Apply\_Mutation}(new\_solution)$ 
9:      $\text{Evaluate}(new\_solution)$ 
10:     $\text{Replacement}(\text{position}(cell), aux\_P, new\_solution)$ 
11:   end for
12:    $P \leftarrow aux\_P$ 
13: end while
14: return Best solution found

```

configuration, all the PEs have the same configuration, that is, they are connected to four memories.

- *Configurable:* The cGA hardware can be easily configured by defining a small number of parameter to control the size of the shared memories, the number of PEs in the rows and columns of the cGA array, and the toroidal or non-toroidal shape of the structure.
- *Applied to different optimization problems and different metaheuristics:* By changing the functionality of the PEs, we can apply our architecture to solve different optimization problems. Additionally, although the architecture has been conceived for cellular GAs, other population-based optimization techniques can also be mapped to this architecture, with appropriate PEs.

3.2.1 Comparison with a canonical cGA

So far, we have presented our proposal for a architecture that supports the execution of cGAs and can be efficiently implemented in hardware. However, how does the changes that we have introduced in the algorithm that is supported by the architecture compare with existing and well-known canonical cGAs?

Essentially, in a cGA the solutions are distributed over a regular grid and for each solution (or *cell*) the interactions needed to generate a new solution only take place within a given neighbourhood of solutions that is centred in the cell. Algorithm 2 presents the pseudo-code of a canonical cGA as it is described in [AD08]. For each solution in the population, the algorithm checks the neighbourhood of solutions that will be used during an iteration, that is, the solutions that can interact with the selected solution

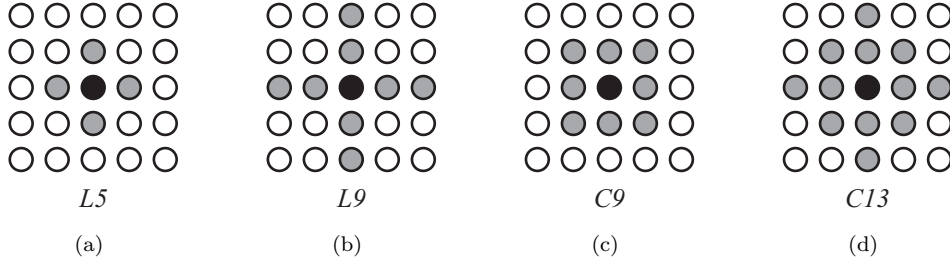


FIGURE 3.2: Example of typical used neighbourhoods in cGAs. Black point represents the elected cell (a solution) and the grey points the solutions in the neighbourhood of the cell.

(the cell). After this, the typical genetic operators (selection, crossover, mutation and replacement) are applied to generate a new solution that is placed in that cell position (or in one of the positions defined by the neighbourhood) to build the next population. After all the cells have been covered, a generation of the algorithm is complete and the new population replaces the old one. The procedure is repeated till a stop criterion is met. We should emphasize that the neighbourhood range is not a fixed parameter that is applied to all optimization problems and, therefore, it must be defined by the user.

Figure 3.2 illustrates four examples of the most common neighbourhoods in cGAs, where the solution in the centre (the black point) is the cell, which is surrounded by the corresponding neighbours (the grey points). The label L_n (linear) for these neighbourhoods represents the $n - 1$ nearest solutions in vertical and horizontal axis, while the label C_n (compact) represents the $n - 1$ nearest solutions in vertical, horizontal and diagonal axis [SDJ96]. It is evident that any two consecutive cells in a cGA possess shared solutions in their neighbourhoods, which is essential for the correct behaviour of the algorithm.

The neighbourhoods of the solutions placed in the memories of the proposed processor array does not match exactly the regular organizations shown in Figure 3.2. Figure 3.3(a) shows an example of a processor array with 8 solutions per memory where two solutions are elected as cells (the black points) to interact with their respective neighbourhoods (grey points). As it can be seen, a given solution has two distinct possible neighbourhoods depending on which PE selects it, represented in the figure by light or dark grey points. The memory shared by both PEs holds solutions belonging to both neighbourhoods.

Figure 3.3(b) shows a modified version of the array of Figure 3.3(a) with 2 times more PEs in both rows and columns of the array, while the solutions are kept identical. This increment by a factor of 4 in the total number of PEs results that each memory keeps now 4 times less solutions, so that the same population size is preserved. The cells

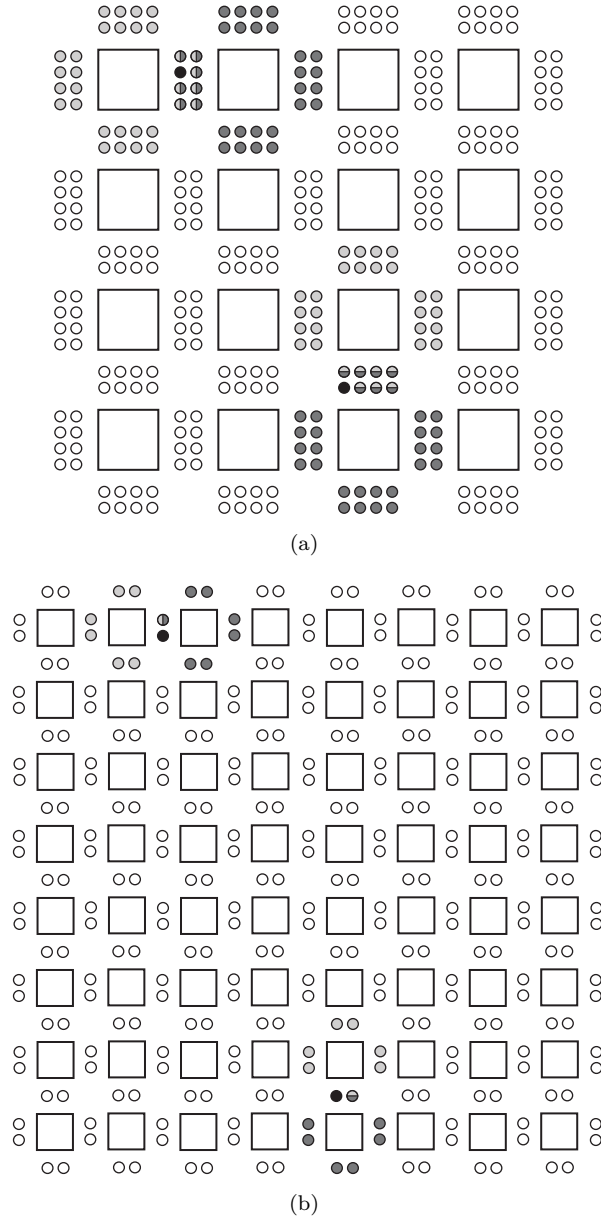


FIGURE 3.3: cGA neighbourhood solutions of the processor array for different levels of parallelism: (a) with 8 solutions per memory and (b) with 2 solutions per memory. Black points: elected cells (solutions); dark and light grey: two distinct possible neighbourhoods.

elected in both figures that interact with their neighbourhoods are the same, but the neighbourhoods are now different.

Although the examples provided above consider that a PE selects all the solutions in its 4 memories for the neighbourhood, it is possible to select only a subset of these solutions by defining the PE functionality to do that. Therefore, a PE with a larger number of solutions per memory can emulate the same neighbourhood that an equivalent cGA with less solutions per memory, but not the other way around.

Another characteristic of the canonical cGA presented in Algorithm 2, is that the new population is generated considering only solutions of the current population. This corresponds to a *synchronous* cGA where all the cells in the population, together with their neighbour solutions, generates concurrently a new population that will replace the previous one in an atomic step. By contrast, in an *asynchronous* cGA as soon as a new solution is generated it immediately replaces (subject to the criterion used) an existing solution of the current population. Therefore, in this class of cGAs it is required to define an update policy that represents the sequential order that each cell is updated. The most common policies are [ADGT06]:

- *Line Sweep*. It is the simplest method where each cell in the cGA is sequentially and row by row updated.
- *Fixed Random Sweep*. The next cell to update is selected with a uniform probability and the same cell cannot be selected more than once for a given generation. A fixed random order is used for all the generations.
- *New Random Sweep*. Identical to the fixed random sweep, with the difference of creating a new random order for all the generations.
- *Uniform Choice*. The next cell is chosen with a uniform probability among all the solutions. A cell can be visited more than once per generation.

Our cGA hardware architecture is designed to have all the PEs executing in parallel and independently from each other. This means that each PE starts to compute a new solution as soon as the last one has been finished, without any kind of synchronization with the remaining PEs. Therefore, this algorithm cannot be classified as a synchronous cGA, which would require a synchronous replacement step for updating a new population.

The update policies presented above for an asynchronous cGA cannot also be applied to our cGA hardware, since they assume that all the solutions are sequentially updated. In the hardware implementation, a subset of solutions (eventually equal to the number of PEs) of the population can be processed at the same time, and updated according to the time that this processing requires to be computed.

All these classifications for cGAs are based on studies in the field of cellular automata performed by [SdR99], where asynchronous methods are divided between *step-driven* or *time-driven*. A step-driven updating method is characterized by the absence of a time variable and thus it is the most natural for using in a cGA as this will be implemented in software where each elected cell is executed one by one. The aforementioned update policies belong to this class. Clearly, our cGA belongs to the class of time-driven where

the updating of a cell (a solution) happens by a process that is independent from the other cells. This process is characterized statistically by the time that a given solution takes from being updated till the next time it is updated again. An example of a time-driven method is the *exponential waiting times* where the occurrence of an event follows an exponential distribution [SdR99]. Nevertheless, in the array of processors we do not know *a priori* the time or the statistics that a solution takes to be updated. Therefore, we classify our cGA as asynchronous with a time-driven update policy.

3.2.2 Application to other population-based metaheuristics

Although the hardware architecture proposed in this work has been thought to solve cellular GAs, it can be applied to other population-based metaheuristics. In its essence, the architecture can simply be seen as a set of processing elements that are capable to compute a given algorithm; each PE is connected to memories that can hold candidate solutions of an optimization problem. In addition, information necessary by the algorithm can be spread throughout the PEs by using the shared memories. Therefore, by distributing the candidate solutions of a problem over the memories, each PE can work simultaneously in their solutions while ensuring the necessary communication with the remaining PEs.

A possible application of the processor array is the particle swarm optimization (PSO) algorithm that is a metaheuristic inspired by the behaviour of a bird flock [KE95]. In this algorithm, to each candidate solution of the problem (called *particle*), it is assigned a given velocity that is calculated according to the best solution (or *position*) found by that particle and the best global solution among all particles. Then, the particle is updated according to its velocity and position. This process is repeated for all the particles.

In [SLGZ11] it is proposed a cellular PSO where the particles are distributed over a regular grid. During the evolutionary process of the algorithm, each particle evolves by the rules defined by the original algorithm, but the best global solution information is only exchanged within neighbour cells. Therefore, there exists a slow information diffusion through the complete population, which may promote diversity while exploitation occurs in each particles' local information. A similar approach is followed by [HM09] where a cellular PSO is applied with success to solve problems whose global optimum value and shape of the fitness function may change over time.

Another algorithm that can be easily adapted to be executed in our hardware architecture is the differential evolution (DE) [SP97]. This metaheuristic, like the GA, belongs

to the class of evolutionary algorithms (EAs) that uses mechanisms inspired by the biological evolution to evolve a population of candidate solutions. The DE is used originally to solve problems over continuous spaces. The algorithm starts by creating a new solution (*vector*) by adding a weighted difference between two population vectors to a third vector, all selected randomly. After that, a solution from the population is elected and it is mixed with the previously generated vector by applying a crossover operation, where the different dimensions of the vector are mixed. Then, the resulting final solution may replace the previously elected solution according to a given criterion.

Since the DE algorithm belongs to the class of EAs, where solutions interact with each other so that a population evolves, it is possible to decentralize the population like it is done in a GA. Therefore, our architecture can be used to implement DE algorithms exactly in the same way as it is done with GAs [NI10]. Moreover, recent studies have found that using a cellular version of a DE algorithm can improve its performance to solve dynamic optimization problems [NHM11]. This finding is consistent with several other works that claim that a cGA is superior to a GA since a slow diffusion of solutions is promoted by distributing them over a cellular grid [AT02].

A local search procedure is often applied to a GA that tries to improve each new solution generated by the classical operations of the GA. This algorithm is known as memetic algorithm (MA) and it mimics both the biological evolution, as it is done by a GA, and cultural evolution by applying a local search [Mos89]. Therefore, the MA can be implemented in the hardware architecture only by changing the functionality of a PE to perform an additional local search operation to the generated solution. Cellular versions of MAs have been successfully implemented to solve complex problems like the batch job scheduling on grid systems [XAD⁺08], or the satisfiability problem [ATDA05].

Figure 3.4 illustrates a possible application of the hardware architecture to implement a distributed GA (dGA). This can be achieved simply by assigning to each PE the task to evolve exclusively a subset of solutions of the population (a subpopulation or island of the dGA). Therefore, the solutions of each subpopulation start to evolve independently from each other and, occasionally, each PE migrates solutions to the neighbour islands which can be accomplished by using the shared memories as it is depicted in the figure. This dGA presents a fixed connection topology among the islands that is imposed by the hardware architecture.

Although we have discussed several applications of our architecture to different population-based metaheuristics, we do not claim it can be, in a general way, applied to all of them. Indeed, by distributing the solutions of a metaheuristic over several memories that can be accessed independently by PEs, it is possible to work with several solutions at the same time, thus improving the number of generated solutions per unit of time.

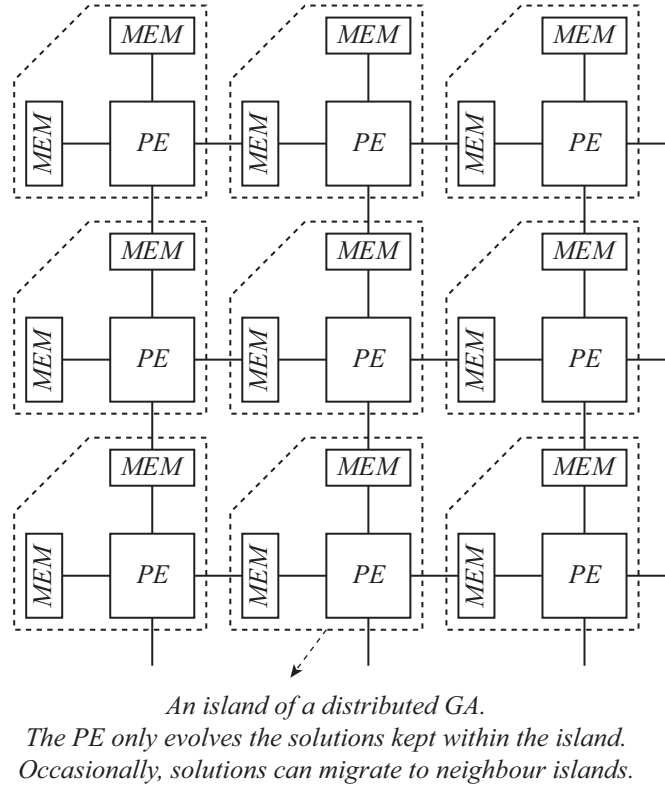


FIGURE 3.4: Application of the processor array architecture to build a distributed GA.

Moreover, the cellular structure of our architecture, where neighbour cells can interact with each other (through the shared memories) makes this architecture conveniently suited for cellular EAs. Additionally, any global data required by a metaheuristic can be diffused throughout the structure by using the shared memories.

3.3 Architecture simulation

We have seen that the proposed architecture can support the execution of asynchronous cGAs with a time-driven update policy since a set of solutions are generated (and updated) in parallel with no relation among them. In addition, the neighbourhood structure of the cGA is imposed by the architecture and changes with the number of solutions per memory. These features are specific of our cGA implementation and distinguish it from others cGAs, and thus a simulation of the algorithm is required to study its convergence for different configurations of the hardware.

In this section we analyse the convergence of the cGA considering different sizes and shapes of the array of processors, reflecting this way possible hardware implementation scenarios. Therefore, we will explore the scalability of the architecture to solve a given

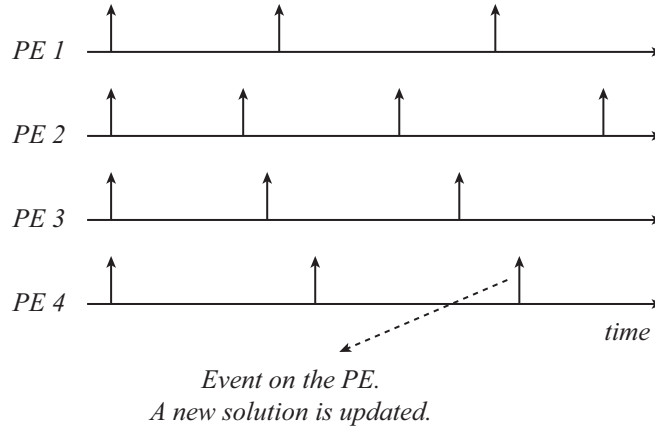


FIGURE 3.5: Discrete-event simulation of the cGA supported by the processor array to emulate its time-driven update policy.

optimization problem by fixing its population size, which results in a smaller number of solutions per memory as the number of PEs increases. Moreover, the algorithm can be easily compared with a panmictic GA by using a single PE.

As discussed in Section 3.2.1, our cGA has a time-driven update policy. Therefore, each solution in the population is updated (or processed) according to a given statistical process that defines the time between two consecutive updates. Thus, to simulate the cGA running in the processor array it is required to perform a discrete-event simulation to emulate the sequence of events over time. Figure 3.5 presents an example of events thrown by 4 PEs that concurrently process their solutions. As it can be seen, although the PEs may start simultaneously, they are not synchronized as the time between two events is not a constant value. We should emphasize that this model of simulation is not equal to randomly select a PE to process a solution. In such situation, for instance, a given PE could be chosen two or more times consecutively while the remaining PEs will not be updating solutions. This scenario does not reproduce the minimum time that a PE takes to process a solution.

In our simulation model we emulate events in the PEs that, in turn, will emulate the evolution of solutions present in the cellular array of the cGA. Therefore, the statistical process that characterizes an event in a PE is different from the statistical process of a solution in the cGA. However, these two processes are related and the time that a solution takes to be updated is equal to the time that this solution takes to be updated by any of the two PEs that have access to it. It is clear that the update policy of solutions in our cGA is a consequence of the capacity that a PE has to compute its algorithm and the sequence it uses to update its solutions.

For the discrete-even simulation we have chosen the SystemC language which is a set of C++ classes capable of providing this kind of simulations [Ini]. This language mimics in

some aspects the hardware description languages (HDLs) and is oriented for system-level modelling.

Our simulation model has been described to emulate the hardware architecture, where the number of PEs in a row and column of the array, the number of solutions per memory, and the toroidal and non-toroidal configuration of the array are parameters. Each PE is capable to execute the typical operations of a GA. We have decided that when a PE is called to compute a new solution (an event), it only updates the solution to the cGA memory in the following event of the same PE. Therefore, an event starts by updating the previously generated solution in that PE and afterwards by generating a new solution. This resembles the real operation of a PE in hardware, where a new solution is only updated to the memory at the end of that generation.

However, for simplicity, we do not simulate access conflicts when two PEs are accessing to the same data in the same memory (a solution). The access to these memories is done instantaneously (in the concept of a discrete-event simulation) and thus no conflicts of data occur. This simplified model allows faster simulation times, and it is sufficient for a first validation of the behaviour of the cGA. Later in Section 3.4, we will present a synthesizable clock cycle accurate model described in Verilog HDL, thus representing accurately a simulation model of its hardware implementation.

We have elected the travelling salesman problem (TSP) to evaluate the algorithm supported by the hardware. The TSP objective is to find the shortest possible route of a salesman that, starting in a given city, has to visit every city of a list exactly once and return to the starting city. This problem has $(n - 1)!/2$ possible solutions where n is the number of cities, and it is a NP-hard problem [LKM⁺99]. Therefore, this optimization problem has a huge search space, ideal to be solved by a metaheuristic search method as the cGA.

The chosen genetic operations of the algorithm are a random selection of two solutions in the neighbourhood of a PE; the maximal preservative crossover (MPX); and a replacement of the worst previously selected solution (the parents) only if the new solution has a better fitness value. Figure 3.6 illustrates an example of the MPX operator for a 5-city problem. This operator, that we have already presented in Section 2.2.1.3, can perform, besides the crossover, also an implicit mutation operation [LKM⁺99]. The example provided in the figure shows that the generated solution presents edges (consecutive cities to be traversed in the TSP) of the graph that are not present in both selected solutions, thus acting also as a mutation. Therefore, we do not use an explicit mutation operator in our cGA. Additionally, the MPX operator has been chosen since it is a simple algorithm capable of providing good results for the TSP [NYYY07].

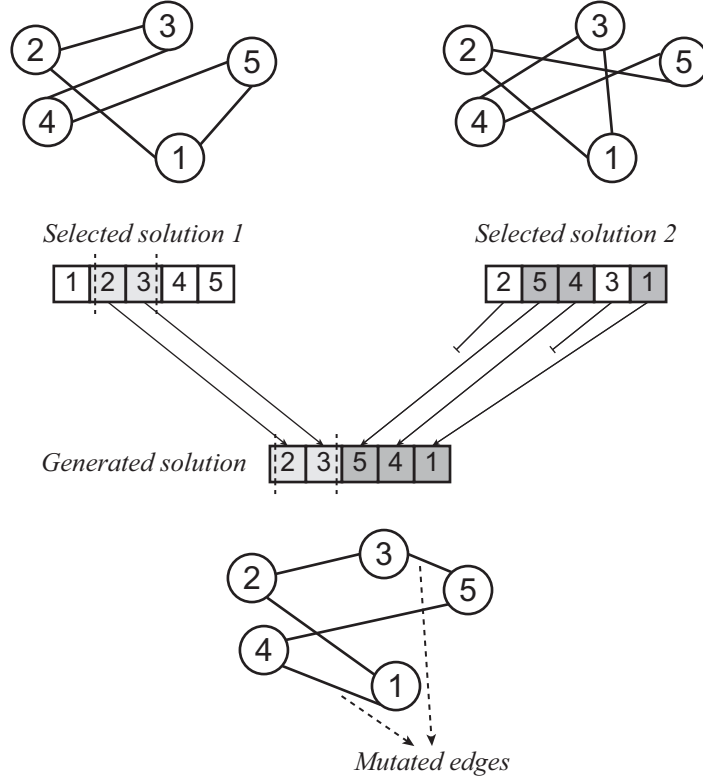


FIGURE 3.6: Example of the maximal preservative crossover applied to the TSP, and how it implicitly performs a mutation operation.

In the next subsections we will present several results for different configurations of the proposed hardware architecture, mainly for square arrays with different levels of parallelism and non-square arrays with equal parallelism. Additionally, toroidal and non-toroidal structures of the array will be analysed. A TSP instance with 280 cities, named *a280* [TSP], will be used for all the experiments that are averaged over 50 independent runs. We have run all the simulations till a maximum of 200×10^6 generated solutions in all the PEs of the array. However, in some graphs we show less information for clarity.

Each PE takes between 35 to 45 time units (distributed uniformly) to generate a new solution, which corresponds to the time of a new event for each PE in our simulation model. This random interval has been chosen to emulate the real behaviour of the MPX algorithm which takes different times to be executed as the length of the copy of the first selected solutions is random (see Figure 3.6). Moreover, in a real implementation of the architecture, memory access collisions will occur that must be solved, resulting this way in an additional random waiting time. The values 35 and 45 have been chosen considering that the selection, fitness, and replacement operations take 10 time units each, and the crossover 5 to 15 time units. More importantly, the total time to generate a solution in a PE ensures that no synchronization occurs among the PEs of the array.

TABLE 3.1: Configurations of the toroidal and square arrays used in the SystemC model simulations.

	Processor array			
	1×1	2×2	4×4	8×8
PEs	1	4	16	64
Memories	2	8	32	128
Solutions/memory	64	16	4	1
Population size	128	128	128	128

It should be stressed that we do not claim that the cGA operators chosen in these experiments are the best for solving the TSP, nor that a GA is a good metaheuristic to solve this problem. The main goal is to evaluate the cGA supported by the processor array with different sizes and organizations, while using the same operators for solving a given optimization problem, so that we could take conclusions about the scalability of the architecture and how this affects the convergence of the algorithm.

3.3.1 Toroidal arrays configuration

In the following subsections we present simulation results of various organizations of a toroidal configuration, both for square and non-square arrays, of the proposed hardware architecture.

3.3.1.1 Square arrays

The evaluation of the cGA running with square arrays, where the number of PEs in both rows and columns is the same, has as main objective to understand the impact that different levels of parallelism brings to the algorithm. For that, we have chosen a population size of 128 solutions that are distributed over the memories of the architecture for the following array sizes: 1×1 , 2×2 , 4×4 and 8×8 PEs. Table 3.1 presents the details of these configurations.

It is important to notice that the different configurations have been chosen to ensure that the population size is kept the same in all the cases. This way, we can compare the behaviour and performance of the algorithm without the interference of different population sizes. In addition, the configuration with 1×1 PE is equivalent to a panmictic GA, since the neighbourhood is the entire population.

Figure 3.7(a) presents the evolution of the best fitness value over 10×10^6 generated solutions in all the PEs. This graph represents in the x-axis the total number of generated

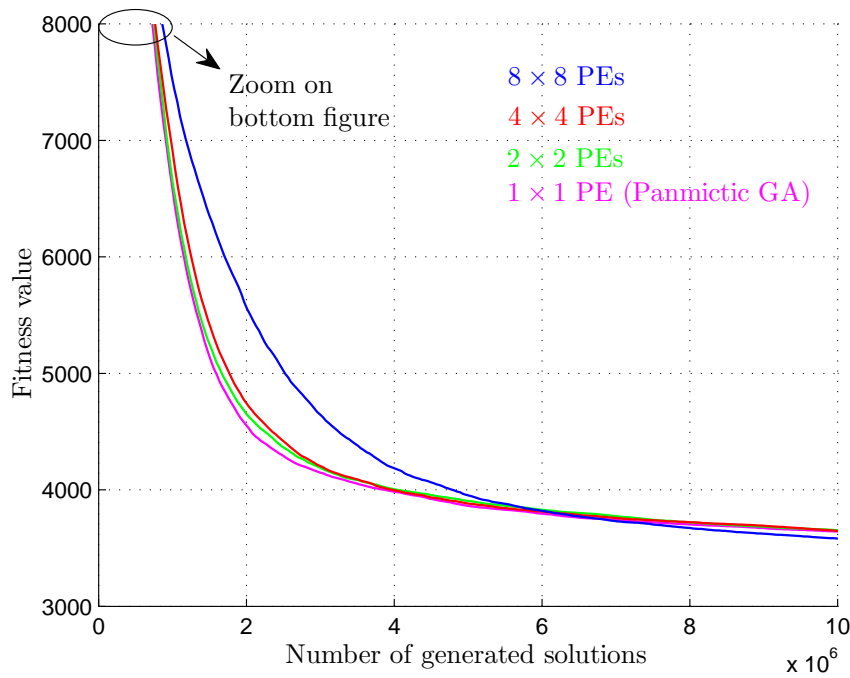
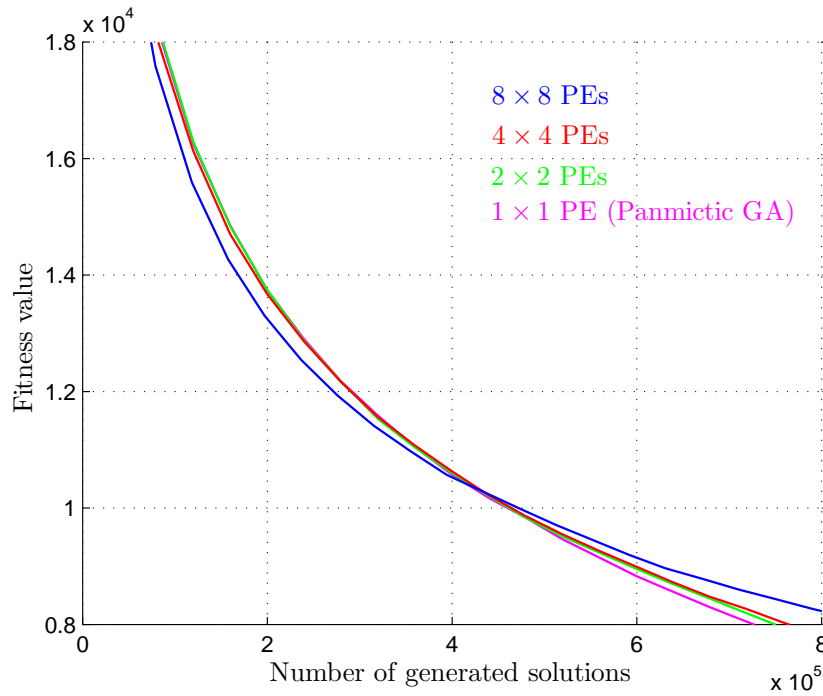
(a) 10×10^6 new generated solutions(b) Zoom on the first 800×10^3 new generated solutions

FIGURE 3.7: Fitness evolution with the total number of generated solutions, obtained with the SystemC model for the cGA supported by toroidal and square processor arrays to solve a TSP.

solutions and thus it does not reflect the execution acceleration achieved by arrays with

a higher number of PEs. This way, we can compare how the algorithm evolution changes for the different neighbourhoods structures (number of solutions per memory) that are a consequence of the change in the number of PEs.

As it can be seen, for processor arrays with a higher number of PEs, the convergence rate of the algorithm is lower in a first phase of the evolutionary process (less than 4×10^6 generated solutions), when compared to arrays with fewer PEs, leading thus to worst quality solutions found. However, these architectures recover, and even show to achieve a superior quality solution as the algorithm continuous to evolve. This is especially notorious in the 8×8 architecture that has 1 single solution per memory.

Interestingly, in a very early phase of evolution of the algorithm, the arrays with higher parallelism have a faster convergence rate than arrays with less PEs. Figure 3.7(b) illustrates this by zooming in the first 800×10^3 generated solutions. This situation happens since the architectures with a larger number of PEs have smaller neighbourhoods in the cGA, which leads to a more intensive search in each PE as each one of them sees a smaller number of solutions. Therefore, the algorithm running in a PE, at this phase, behaves somehow as a panmictic GA with a very small population, leading thus to a faster convergence rate.

Table 3.2 shows several fitness values during the evolution of the algorithm for the maximum number of generated solutions in the complete simulation (200×10^6), highlighting the best fitness for each evolution point found for the different array configurations. As it can be seen, the final value found by the different configurations is consistently better as the number of PEs increases.

Note that the neighbourhood of the cGA changes according to the level of parallelism of the architecture. A higher parallelism leads to a smaller number of solutions in the neighbourhood of the cGA, whereas a smaller parallelism leads to a higher number of solutions in the neighbourhood of the cGA (cf. Figure 3.3). Indeed, the different convergence rates for the different levels of parallelism are a result of this neighbourhood change and how long a solution takes to spread its genetic information throughout the array [SDJ96].

It is clear from the results that our cGA does not degrade the quality of the final solution found when compared to the panmictic GA (1×1 PE). Additionally, the quality of the final solution improves as the neighbourhood size gets smaller. This fact happens since there exists a smooth diffusion of the solutions' information thorough the population, which provides a better sampling of the search space that leads to better results [AD08].

Figure 3.8 shows the same results as the previous graphics, but now reflecting the real gain obtained by the parallelism of the architecture. Therefore, the x-axis represents the

TABLE 3.2: Fitness values obtained with the SystemC model for the cGA supported by toroidal and square processor arrays to solve a TSP.

Generations	Best fitness value			
	1×1	2×2	4×4	8×8
5×10^3	27969	27984	28028	28235
10×10^3	26233	26343	26236	26416
20×10^3	24183	24360	24169	24077
50×10^3	20561	20624	20460	19864
100×10^3	17178	17245	17000	16394
200×10^3	13735	13746	13670	13227
300×10^3	11861	11830	11861	11602
400×10^3	10574	10601	10631	10523
500×10^3	9647	9673	9698	9797
800×10^3	7560	7703	7814	8227
1×10^6	6585	6651	6911	7524
2×10^6	4553	4653	4745	5562
3×10^6	4148	4190	4205	4646
4×10^6	3984	4006	3997	4185
5×10^6	3862	3905	3881	3953
6×10^6	3795	3828	3814	3820
7×10^6	3741	3775	3758	3731
10×10^6	3642	3654	3648	3582
20×10^6	3508	3507	3482	3395
50×10^6	3402	3364	3343	3255
100×10^6	3362	3322	3303	3215
200×10^6	3338	3310	3287	3200

number of abstract time units used in the context of our discrete-event simulation. As expected, the time that each configuration takes to execute the algorithm is inversely proportional to the number of PEs. This means, for example, that the 8×8 configuration (64 PEs) is 4 times faster than the 4×4 configuration (16 PEs) and 64 times faster than the configuration with a single PE. It is evident from the figure that any slower converge rate previously observed by the arrays with higher parallelism, clearly vanishes with the speedup achieved. Therefore, the cGA supported by the architecture not only is efficient as it provides a mean to accelerate the algorithm, but also it is effective as it provides good quality results for the optimization problem.

3.3.1.2 Non-square arrays

To analyse the behaviour of the cGA with non-square arrays, we have used the same population size as previously, but now changing the arrays from 1×64 to 8×8 PEs as shown in Table 3.3. In all these configurations we have the same parallelism and the same number of solutions per memory which leads to the same neighbourhood structure

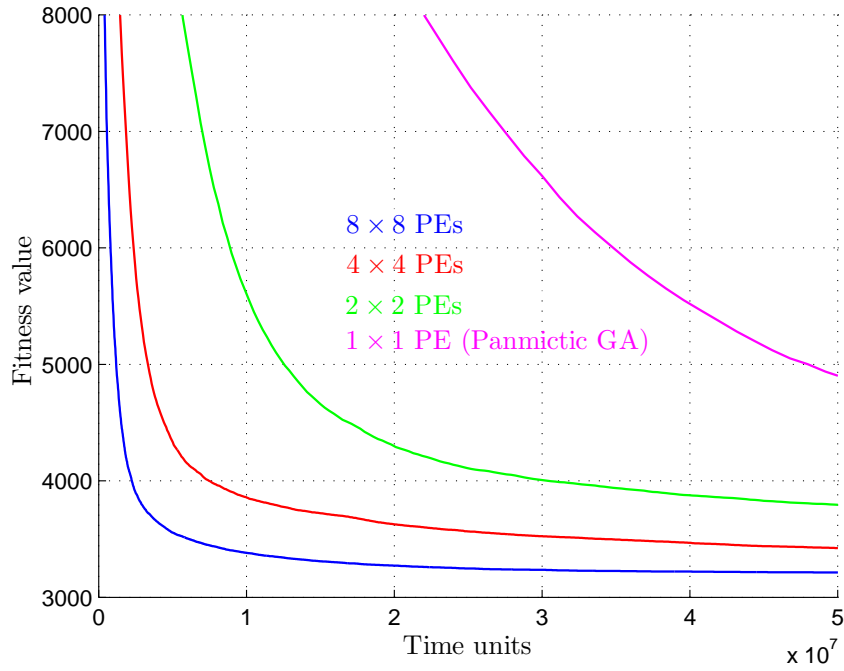


FIGURE 3.8: Fitness evolution with number of clock cycles, obtained with the SystemC model for the cGA supported by toroidal and square processor arrays to solve a TSP.

TABLE 3.3: Configurations of the toroidal and non-square arrays used in the SystemC model simulations.

	Processor array			
	1 \times 64	2 \times 32	4 \times 16	8 \times 8
PEs	64	64	64	64
Memories	128	128	128	128
Solutions/memory	1	1	1	1
Population size	128	128	128	128

in all situations. Therefore, we investigate how the algorithm behaves by changing the aspect ratio of the array of processors, while the remaining conditions are kept equal.

Figure 3.9 illustrates the fitness value evolution for the different arrays for 30×10^6 new generated solutions in all PEs. Additionally, in the graph we also include the panmictic GA of the previous experiments so that we can compare it with the cGA. Note however, that although the x-axis of the graph represents the number of generated solutions, it can also represent the abstract time units since all the array configurations have the same number of PEs. Therefore, the graph also represents the acceleration obtained for the different non-square arrays (exception is the panmictic GA that is 64 times slower, which is not shown in the graph). As it can be seen, the narrower arrays (e.g. 1×64) lead to a

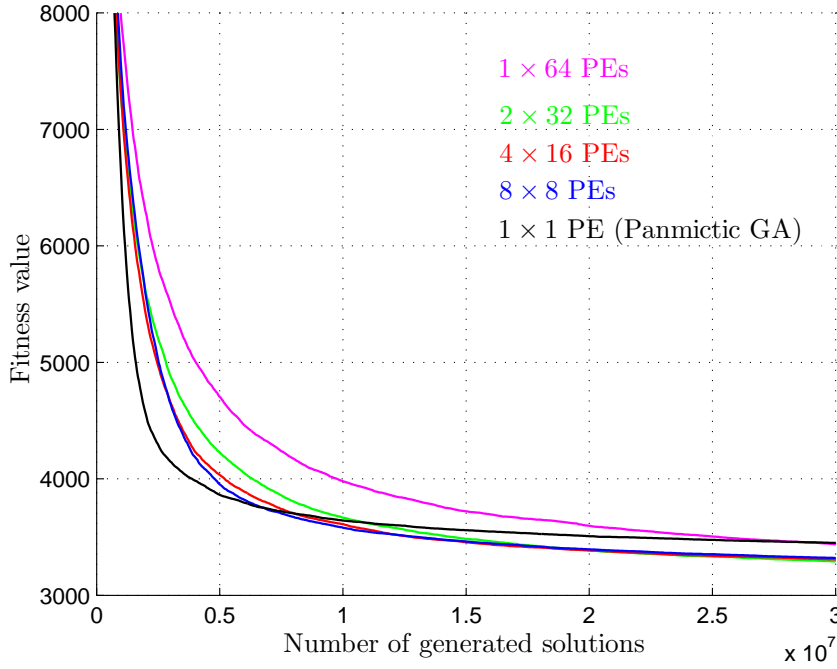


FIGURE 3.9: Fitness evolution with the total number of generated solutions, obtained with the SystemC model for the cGA supported by toroidal and non-square processor arrays to solve a TSP.

slower convergence rate of the algorithm when compared to more square. This behaviour is identical to the one previously seen for square arrays with more PEs that require a smaller neighbourhood size. It should be noted however, that it is the same reason in both situations that lead to this behaviour in the convergence rate of the algorithm: the ‘time’ that solutions take to spread their information throughout the array. Although we have used the same neighbourhood structure, the narrower the configuration is for the same number of PEs, the more difficult is to spread the solutions’ information.

Table 3.4 presents the fitness values observed in the previous figure for the different arrays, till the maximum number of generated solution in these experiments. At an early stage of the algorithm, the arrays where it is more difficult to spread the solutions’ information show a superior performance. After that, the less narrow arrays possess the best results (around 5×10^6 generated solutions), and continuing to evolve the algorithm the narrower configurations tend to show again the best results. Once again, these observations are consistent with the ones observed for square structures. Although the data after 200×10^6 does not show that the best array is the one with 1×64 PEs, this array, at this point, possess the higher convergence rate when compared to the remaining. Therefore, letting the algorithm evolve for more generations, will eventually show that the 1×64 array leads to the best results.

TABLE 3.4: Fitness values obtained with the SystemC model for the cGA supported by toroidal and non-square processor arrays to solve a TSP.

Generations	Best fitness value			
	1×64	2×32	4×16	8×8
5×10^3	28701	28208	28272	28235
10×10^3	26907	26265	26302	26416
20×10^3	24180	23615	23855	24077
50×10^3	19742	19150	19603	19864
100×10^3	16239	15775	16053	16394
200×10^3	13173	12761	12998	13227
500×10^3	9949	9424	9562	9797
1×10^6	7942	7269	7273	7524
2×10^6	6285	5608	5409	5562
3×10^6	5510	4881	4656	4646
4×10^6	5010	4477	4232	4185
5×10^6	4704	4225	4034	3953
10×10^6	3980	3666	3610	3582
12×10^6	3857	3582	3524	3524
15×10^6	3721	3485	3456	3462
20×10^6	3598	3386	3385	3395
30×10^6	3438	3291	3308	3320
40×10^6	3353	3234	3253	3279
50×10^6	3302	3198	3221	3255
100×10^6	3172	3136	3147	3215
200×10^6	3122	3109	3120	3200

It is evident that the change of the aspect ratio of the processor array configuration, while keeping the same amount of PEs, does not improve the number of generated solutions per time unit in a real hardware implementation. Therefore, it is important to keep in mind the different behaviours that these structures produce in the convergence of the algorithm. Narrower structures, although they may produce better quality results, take a longer time to converge to the final solution, while a square structure produces a good balance between convergence time and quality results. Nevertheless, the time or the number of generated solutions that a cGA takes to be executed are ambiguous parameters, and in a real implementation of a cGA we do not now *a priori* which configuration may be the best.

3.3.2 Non-toroidal arrays configuration

In the following subsections we will simulate the array with non-toroidal configurations, both for square and non-square arrays. A non-toroidal configuration may be interesting in a hardware implementation of the processor array since it can potentially reduce the

TABLE 3.5: Configurations of the non-toroidal and square arrays used in the SystemC model simulations.

	Processor array			
	1×1	2×2	4×4	8×8
PEs	1	4	16	64
Memories	4	12	40	144
Solutions/memory	32	11	3	1
Population size	128	132	120	144

routing complexity required to build a toroidal array, thus leading to reduced routing costs.

3.3.2.1 Square arrays

For non-toroidal and square arrays we have used the parameters presented in Table 3.5. The configuration sizes have been chosen to be equal to the ones presented previously for toroidal arrays. However, in order to ensure that all PEs work with the same number of solutions per memory, it is not possible to have in all the configurations a population size of 128 solutions since additional memories (and solutions) need to be introduced in the architecture. Therefore, the number of solutions per memory has been chosen so that Equation (3.5) is satisfied and the populations size is as close as possible to 128.

Figure 3.10 presents the fitness evolution for the different arrays analysed as well as for the 8×8 PEs with a toroidal configuration for a total of 20×10^6 new generated solutions. As it can be observed, the convergence behaviour is similar to the ones found previously in the toroidal case, where more parallelism (smaller neighbourhoods) leads to a slower convergence rate and a better quality results. Additionally, the convergence of the algorithm with time is similar to the toroidal for square arrays (c.f. Figure 3.8) where the acceleration of the cGA is proportional to the number of PEs. Therefore, the increase of the number of PEs is beneficial since it produces a much faster algorithm implementation, and capable of producing solutions with superior quality for the same number of generations.

Comparing the two 8×8 configurations (non-toroidal and toroidal) it can be seen that the non-toroidal shows a slower convergence rate, which is consistent to the previous reasoning that the convergence rate is related to the potential of interaction among the solutions in the population. However, since these two configurations have different populations sizes (144 for non-toroidal and 128 for toroidal) we cannot make a fair comparison between them, and the faster convergence of the toroidal array may also include effects of a smaller population which makes the algorithm to converge faster and

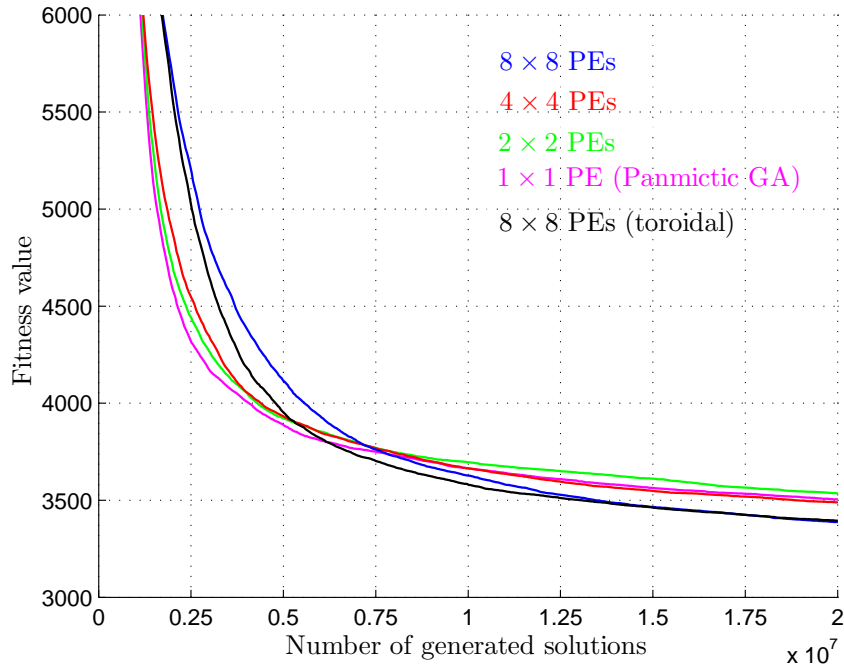


FIGURE 3.10: Fitness evolution with the total number of generated solutions, obtained with the SystemC model for the cGA supported by non-toroidal and square processor arrays to solve a TSP.

for a solution with less quality (after 200×10^6 generations, the toroidal array achieves 3200, while the non-toroidal array achieves 3114).

3.3.2.2 Non-square arrays

In this last experiment, we have analysed non-square and non-toroidal arrays configurations. As done previously, we target the population size to 128 solutions while changing the aspect ratio of the processor array, thus maintaining the same degree of parallelism. Table 3.6 presents the details of these configurations. As it can be seen, the narrower arrays present a considerable number of memories which are required to build the non-toroidal configuration. Therefore, the number of solutions increases far above 128 (193 for the 1×64).

Figure 3.11 illustrates the fitness evolution for the different configurations analysed together with the 8×8 PEs toroidal for a maximum of 30×10^6 new generated solutions. The results obtained are as expected, with the narrower configurations showing a smaller convergence rate that, eventually, will end in a better quality solution found by the algorithm. However, the configurations show a quite different population size among them (ranging from 193 to 144 solutions) which influences the overall performance of the cGAs and, therefore, making them less comparable.

TABLE 3.6: Configurations of the non-toroidal and non-square arrays used in the SystemC model simulations.

	Processor array			
	1×64	2×32	4×16	8×8
PEs	64	64	64	64
Memories	193	162	148	144
Solutions/memory	1	1	1	1
Population size	193	162	148	144

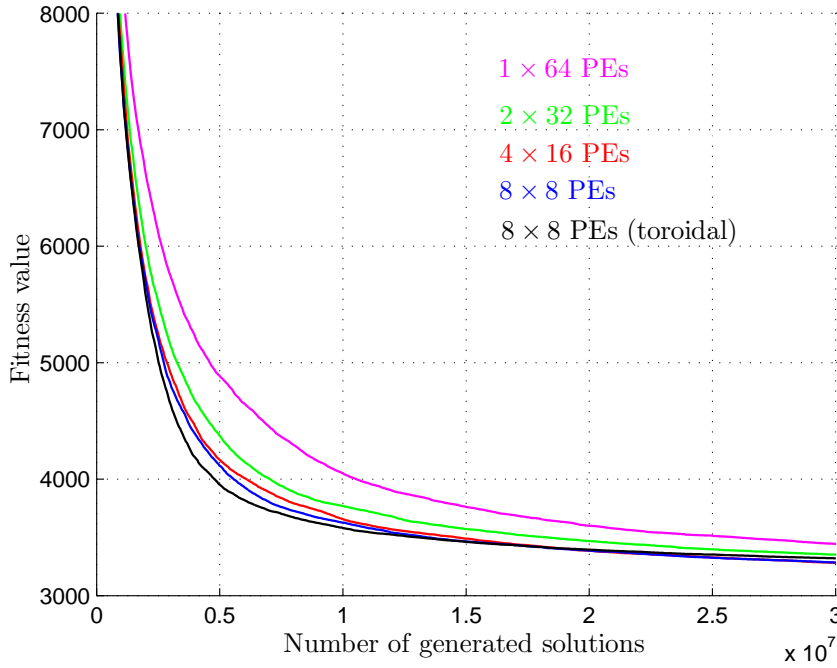


FIGURE 3.11: Fitness evolution with the total number of generated solutions, obtained with the SystemC model for the cGA supported by non-toroidal and non-square processor arrays to solve a TSP.

It is clear that for non-toroidal configurations with narrow arrays, the architecture requires a large number of memories which is less attractive in a hardware implementation than for a square structure. However, it is possible to build, for example, a 1×48 non-toroidal array with 145 solutions (1 per memory), which is comparable to a 8×8 non-toroidal array (with 144 solutions). Nevertheless, the 1×48 configuration has significantly less number of PEs and therefore does not produce the same number of solutions per unit of time as the square structure.

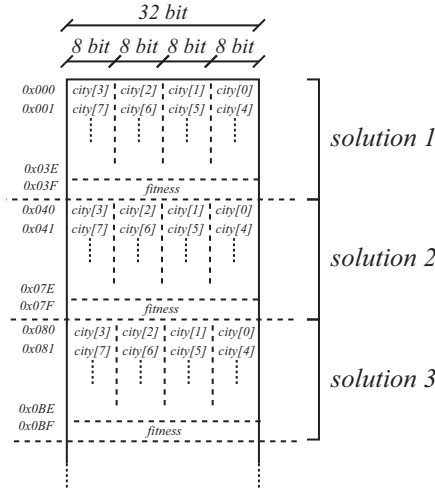


FIGURE 3.12: Organization of a shared memory in the processor array for solving the TSP with a maximum of 252 cities.

3.4 Hardware implementation: the TSP

In this section we will present a first implementation of the proposed processor array architecture in hardware that solves, like in the previous experiments with SystemC, the TSP. Therefore, we have adopted the same operations for the algorithm: a random selection of two solutions, the maximal preservative crossover (MPX), and a replacement of the worst previously selected solutions only if the new solution has a better fitness.

The logic system has been developed in synthesizable Verilog HDL to target a Xilinx Virtex-6 FPGA, and represents a clock cycle accurate hardware model. The next sub-sections present the details of the implementation of a PE to solve the TSP.

3.4.1 Processing element

The processing elements (PEs) of the array architecture are responsible to execute the operations of the algorithm. Therefore, by changing their functionality, we can adapt the architecture to solve different optimization problems, like the TSP.

Figure 3.12 shows how the solutions are organized and coded in the shared memories of the processor array, where a TSP solution is coded by a list of cities (path representation as described in Section 2.2.1.1). As it can be seen, the memory has a width of 32 bits that are equally divided to keep 4 cities of the problem. The sequence of the cities that define a solution are contiguously kept in the memory and the last memory position of a solution is used to store the solution's fitness value. Therefore and by design, we have constrained the maximum size of a TSP instance to 252 cities.

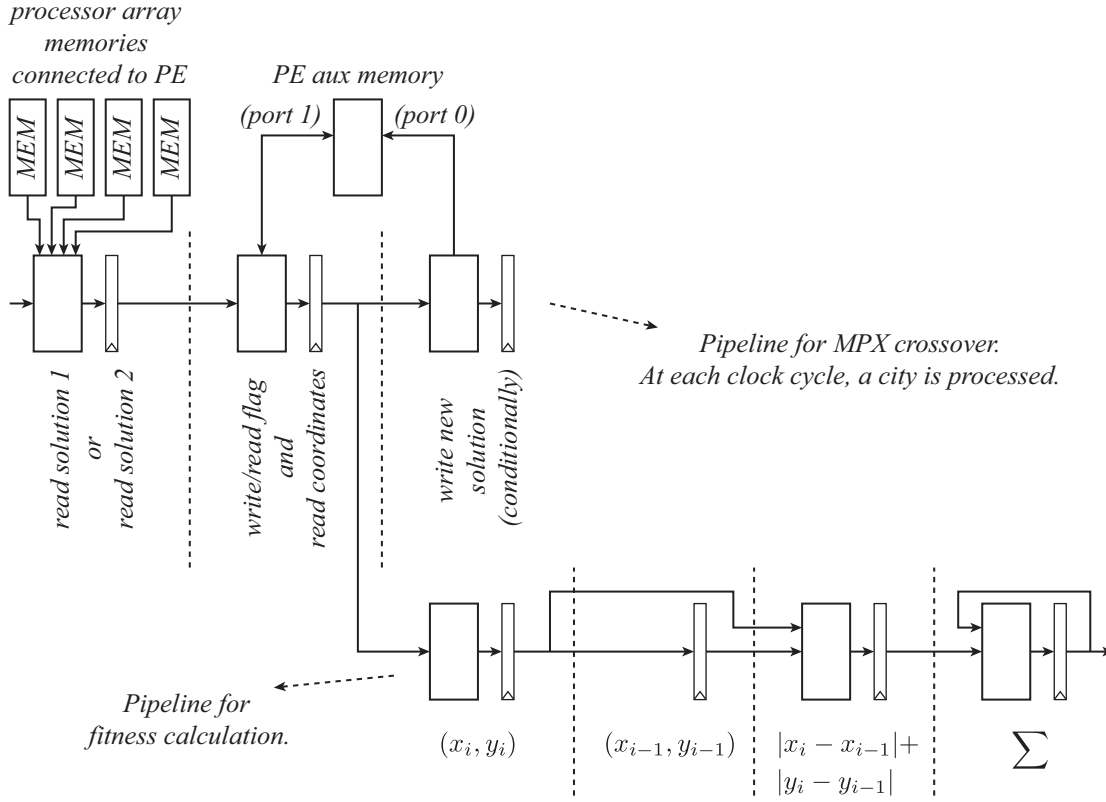


FIGURE 3.13: Overview of the hardware pipeline implemented in the PE to compute the MPX and fitness of a TSP solution.

As we will see, the most time consuming operations in a generation of a new solution are the crossover (MPX) and fitness evaluation. Therefore, we have developed a hardware pipelined architecture to process these operations which is illustrated in Figure 3.13. This circuit has a 3-stage pipeline to compute the MPX operator and a 4-stage pipeline for the fitness evaluation.

The MPX algorithm builds a new solution (see Figure 3.6) starting by copying a random consecutive sequence of cities of the first previously selected solution and, afterwards, it fills in the missing cities of the new solution in the same order as they appear in the second selected solution. Therefore, the MPX algorithm used has two main phases: in the first it copies the part of the first solution to an auxiliary memory while it marks the cities that have been copied; in the second phase all the cities are read from the second solution and written to the auxiliary memory if they have not been previously marked. The first stage of the pipeline corresponds to the memory read of the solutions; the second stage to the access to the *mark* memory; and the third stage writes the new solution into the auxiliary memory.

Since the hardware target of this implementation is a Virtex-6 FPGA, we have decided to use their memory blocks (BRAMs), which have true dual-port capabilities, to implement

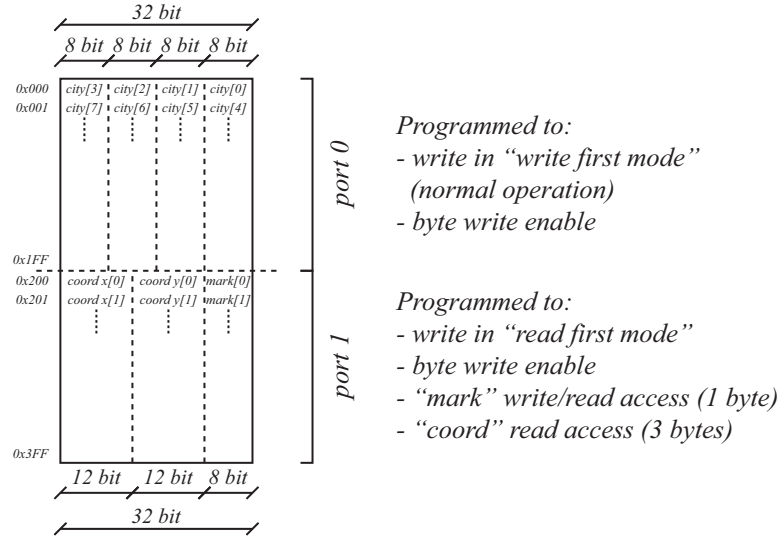


FIGURE 3.14: Configuration and organization of the auxiliary memory (a BRAM from a Virtex-6 FPGA) in a PE to solve the TSP.

the auxiliary memory to keep both the new solution and the *mark* memory. Figure 3.14 shows the organization of this memory. As it can be seen, port 0 keeps the new solution where the cities are organized in the same way as they are in the memories used to keep the population. In turn, port 1 ensures the access to the *mark* memory where each address corresponds to a city identification.

For the fitness evaluation, we have decided to calculate explicitly the distance of a solution based on the coordinates of the cities. Therefore, each PE needs to keep the coordinates values so that it can compute the fitness of a new solution. Although we could use the distance between any two cities, this approach would require much more memory to store all the distance between any two cities of a TSP instance. For example, considering 252 cities, 16 BRAMs of a Virtex-6 FPGA are required to keep all the information (using 16 bits for each distance value). Instead, using the coordinates' values, a single BRAM can easily accommodate the required values. Since each PE has its own TSP data, the use of the distance between any two points is thus impractical for a high level of parallelism in the array.

We also have used the auxiliary memory of a PE to keep the coordinates of the TSP instance as depicted in Figure 3.14. When a new city is read from the shared memories, the port 1 of the auxiliary memory simultaneously accesses to the *mark* part of the memory and reads the coordinates of the city that is being processed. It must be stressed that the port 1 of this BRAM is programmed to work in ‘read first mode’ so that it is possible to write data to the *mark* and simultaneously the previous data is also read. With these techniques, we use a single BRAM per PE to keep the necessary data to generate a new solution, while ensuring a pipelined circuit without stalls.

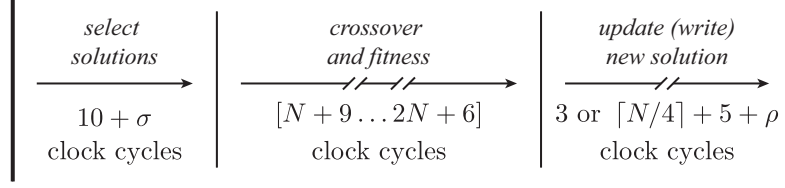


FIGURE 3.15: Sequence of operations executed by a PE during a generation of a new solution for solving a TSP with N cities ($N \leq 252$).

The first two stages of the pipeline of the fitness evaluation circuit keep the coordinates of the actual and previous processed city so that, in the third stage, the distance between the two cities is calculated. The last stage accumulates the distances so that the final value of the fitness is calculated. For simplicity, we have used the sum of the absolute differences of the coordinates as a metric to calculate the fitness value, which is given by

$$fitness = \left\{ \sum_{i=1}^{N-1} |x_i - x_{i-1}| + |y_i - y_{i-1}| \right\} + |x_{N-1} - x_0| + |y_{N-1} - y_0| \quad (3.8)$$

where N is the number of cities of the TSP instance.

Figure 3.15 illustrates the sequence of the different operations that a PE uses to compute a complete generation of the algorithm, together with the number of clock cycles required for each of them. The selection of solutions and update of a new solution clearly require less time than the crossover and fitness operations. As the update of the new solutions is conditionally (only happens if the new solution improves the fitness of the worst selected solution) it may take only 3 clock cycles. The variables σ and ρ account for the memory access collisions that may happen when two PEs try to get access to the same solution. Since these access requests are random, it is not possible to quantify them. In the following sections we will explain how these accesses are implemented and how they impact on the performance of the algorithm implementation.

Additionally, we have implemented in each PE a random number generator (RNG) to feed their selection and crossover modules with the necessary random numbers. In this work, a 64-bit RNG based on cellular automata (CA) techniques has been adopted. Section 4.6.1 provides details of this RNG.

3.4.2 Memory access control

The shared memories of the processor array architecture have been implemented with BRAMs of the FPGA, and configured to work in dual-port mode. Therefore, each of the two PEs connect to one of these ports so that they can access independently

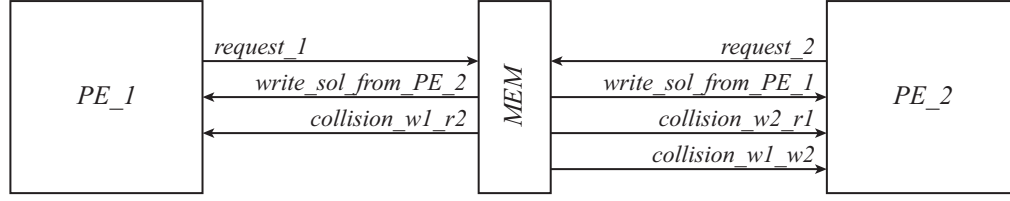


FIGURE 3.16: Signals involved in the memory access control among two PEs and a shared memory of the processor array architecture for solving the TSP.

to the memory's contents. Nevertheless, simultaneous access to the same contents (a solution) can occur, and thus they must be handled carefully. In general, a solution of the problem occupies several memory locations, and thus a simultaneous write into the same solution locations can make data incoherent. Consequently, if a PE is writing new data, the other cannot access it, as the contents of that solution will be temporarily incoherent. However, read operations can be performed simultaneously.

In our design, memories and PEs collaborate to avoid undesired access collisions. Each memory module keeps track of which of its solutions are being accessed, both read and written, by the two PEs connected to it. Figure 3.16 shows the signals involved for the memory access control among a memory and its PEs.

During the selection phase, a PE checks continuously the signal `write_sol_from_PE` that informs it about the solution (if any) being written by the other PE. Using this information, the PE must ensure that a different solution is chosen during the selection process. Then, the PE uses the `request` signal to inform the memory block which solution will be using, together with the information that if is a writing or reading access request. During this phase, a solution can be selected for writing even if it has been selected for reading by the other PE. In this case, the write operation must wait until the read finishes, as specified by the signal `collision_w_r`. If a memory receives two write requests to the same solution in the same clock cycle, the memory accepts one of the requests and uses the `collision_w_w` signal to inform the other PE that the access was denied. When the solutions previously selected for reading and writing are not needed anymore, a request command is issued to the memory to release the corresponding solutions.

3.4.3 Implementation and results

The proposed architecture was implemented as a parameterized Verilog HDL model to solve TSP instances up to 252 cities, and synthesized using Xilinx ISE 12.4 to run on a Virtex-6 FPGA (XC6VLX240T-1). We have considered a population size of 128 solutions that are distributed over the array to solve a benchmark named *ch150* with

TABLE 3.7: Characteristics of the toroidal processor arrays implementations on a Virtex-6 (XC6VLX240T-1) FPGA for solving the TSP.

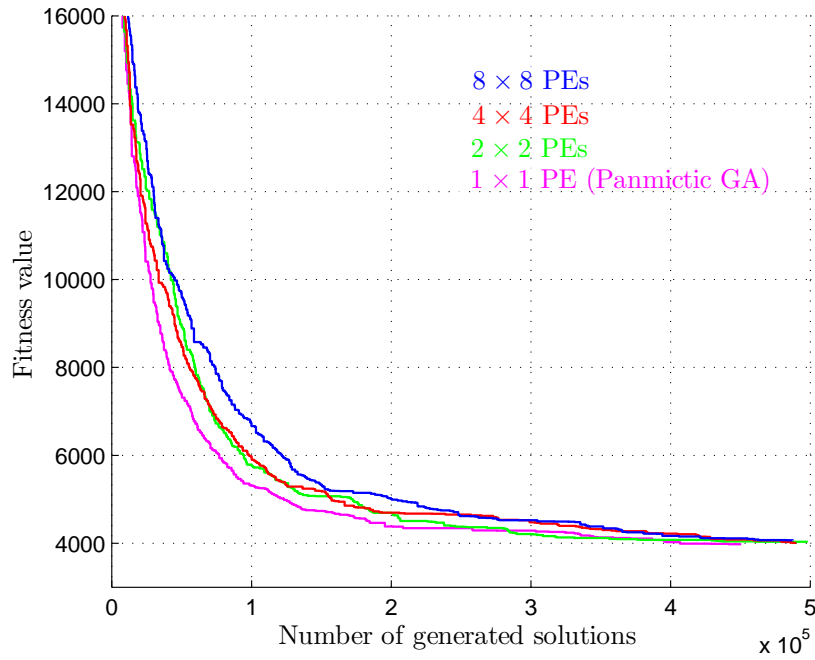
	Processor array configuration			
	1×1	2×2	4×4	8×8
Registers	543 (0.2 %)	2020 (0.7 %)	7485 (2.5 %)	27591 (9.2 %)
LUTs	663 (0.4 %)	2605 (1.7 %)	9306 (6.2 %)	34855 (23.1 %)
Slices	229 (0.6 %)	913 (2.4 %)	3727 (9.9 %)	13316 (35.3 %)
BRAMs	9 (2.2 %)	12 (2.9 %)	48 (11.5 %)	192 (46.2 %)
Frequency	186 MHz	179 MHz	152 MHz	122 MHz

150 cities from [TSP]. In these experiments, we have evaluated the scalable processor array for toroidal configurations with 1×1 , 2×2 , 4×4 and 8×8 PEs.

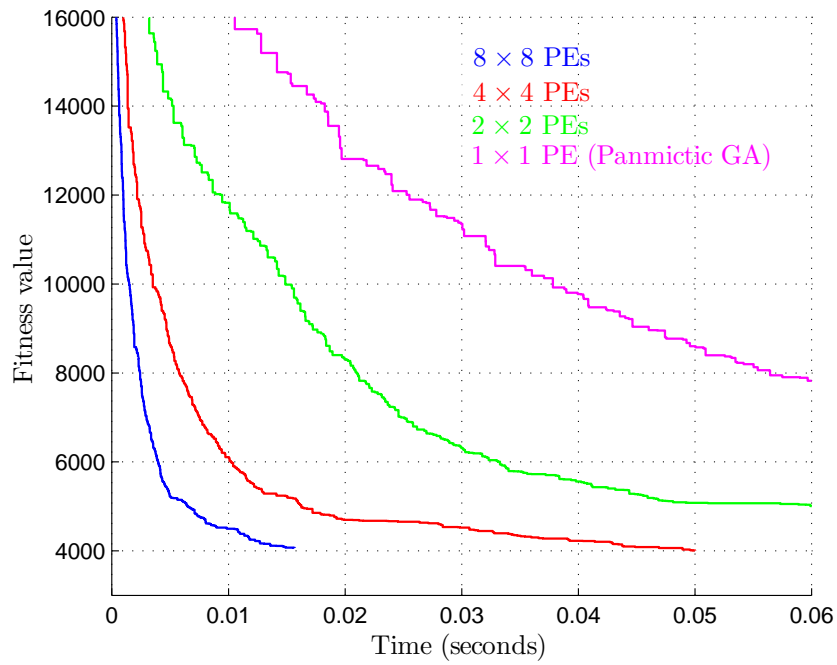
Table 3.7 presents the implementation results for the different array dimensions. As expected, the number of hardware resources required to implement the architecture increases almost linearly with the number of PEs. The maximum clock frequency reported by the timing analyser decreases from 186 MHz (1 PE) to 122 MHz (64 PEs). This happens due to the delays associated with the connections used to build the toroidal shape of the array, which become more critical as the number of PEs increases.

As stated in Equation (3.2), the number of shared memories needed to build a toroidal array is two times the number of PEs. In this implementation, each PE uses an additional memory to keep auxiliary data required to compute a new solution as explained previously. Therefore, the different configurations need 3 times more number of BRAMs of the FPGA as number of PEs, if a single BRAM is capable to accommodate all the necessary solutions. In the array with 1×1 PE, the two shared memories need each one 4 BRAMs (each with 36 Kbit) to keep all the solutions.

Figure 3.17(a) depicts the fitness evolution over 500×10^3 new generated solutions for the different configurations analysed during a single run of the algorithm. As it can be seen, the convergence rate observed is identical to the one observed with the SystemC model (cf. Figure 3.7(a)), where more parallelism level (a smaller neighbourhood in the solutions of the cGA) leads to a slower rate convergence rate at the beginning of the algorithm that, eventually, will translate to a better solution found by the algorithm



(a) Fitness evolution with the total number of generated solutions.



(b) Fitness evolution with time.

FIGURE 3.17: Fitness evolution obtained with the Verilog HDL model for a toroidal and square processor array to solve a TSP. Simulations stop at 500×10^3 new generated solutions.

as this evolve. As it can be seen, we obtain the same behaviour as the previous SystemC model, and that the cGA does not degrade the quality of the final solution when

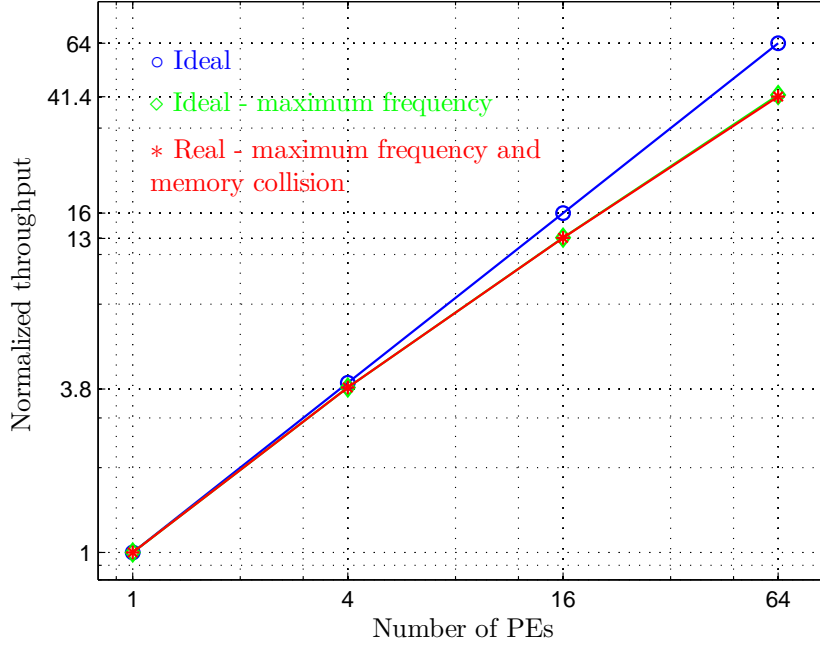


FIGURE 3.18: Throughput of the processor array for the configurations of 2×2 , 4×4 , and 8×8 , normalized to the throughput of a single PE. Results obtained with the Verilog HDL model for solving the TSP.

compared to the panmictic GA.

Figure 3.17(b) shows the same data as the previous figure, but now the fitness evolution is measured as a function of the actual execution time required for the different arrays, using thus the maximum clock frequency supported by each implementation. This graph presents the real overall time improvements obtained by exploiting the parallelism level of the architecture, as it includes the effects of a slower convergence rate for larger arrays, clock frequency degradation, and eventual memory collisions effects. The experimental results show that the 1×1 array produces an average of 0.75×10^6 solutions/second, and for the 8×8 array the throughput is 31.09×10^6 solutions/second, which represents a $41 \times$ throughput increase.

To measure the impact that the memory access collisions have in the performance of the architecture, we have taken the array with a single PE as a reference since in this configuration these effects are not present. Figure 3.18 shows the normalized throughput estimated by considering a proportional increase of the array size (ideal situation), accounting for the effect of maximum clock frequency degradation, and the actual measured throughput that includes additionally the effects of memory collisions. The data shows that it is mainly the degradation of the clock frequency for larger arrays that limits the speedup, and that there is no significant additional impact of the memory collisions. The maximum throughput decrease due to these collisions occurs for the 8×8

array and is less than 0.55 generations per unit of time, which is equivalent to an average increase of 3.26 clock cycles per new generated solution (represents the value of $\sigma + \rho$ in Figure 3.15). For the 4×4 and 2×2 configurations, these values are 0.361 and 0.0423 clock cycles, respectively. These figures mean a degradation on the execution time of the algorithm for the 8×8 , 4×4 and 2×2 arrays of respectively 1.31 %, 0.146 % and 0.0171 %.

An equivalent panmictic genetic algorithm was developed in the C language and run on a single processor of a personal computer (PC). This software implementation does not explore parallelism and is thus equivalent to the 1×1 cGA. The program was compiled with GCC -O3 and executed on an Intel T8100 processor running at 2.1 GHz, to solve the same TSP instance. The PC achieved a throughput of 0.69×10^6 solutions/second, which translates to a $45 \times$ speedup for the 8×8 PEs FPGA implementation.

The same procedure was also executed on a MicroBlaze soft-core processor running at 150 MHz in the same Virtex-6 FPGA. For embedded applications with limited computing capabilities, or situations where area is constrained and integrated solutions are required, our proposal for a custom implementation of cGAs is thus an application scenario and the performance comparison with such processor is realistic. Results have shown a throughput of 4.59×10^3 solutions/second. This means that our hardware architecture can achieve an impressive speedup of more than $6700 \times$ compared to a software version running on the same hardware platform.

3.5 Summary

In this chapter we have presented a scalable hardware architecture for cGAs and others population-based metaheuristics, that can be efficiently implemented in parallel computing platforms like FPGAs. The architecture is built by repeating a regular structure, where a PE connects to 4 memories and each memory is shared by 2 PEs; each PE computes the operations of the algorithm and each memory holds a subset of the population. The cGA supported by this architecture has been classified as asynchronous with a time-driven update policy, which reflects the way the solutions in the population are updated.

Simulation results have shown that the convergence rate of the algorithm is related with the difficulty that a solution has to spread its information throughout the population. Configurations where it is more difficult, lead to a slower convergence rate at the beginning of the algorithm and produce better quality results as the number of generations grows. In such situation are cGA architectures with increased number of

PEs, which lead to a smaller neighbourhood size. Nevertheless, results clearly show that the speedup obtained by increasing the parallelism is always beneficial as it produces a more efficient algorithm (runs faster), with increased effectiveness (better solutions for the same number of generated solutions) due to the smaller neighbourhood size.

Additionally, we have implemented the cGA hardware architecture in an FPGA to solve the TSP. Nevertheless, the effort needed to develop a custom PE for this optimization problem is high and time consuming using standard HDL to describe the hardware.

In the next two chapters we present all the details of an improved cGA hardware architecture with additional infrastructures to support the execution of the algorithm. A hardware design flow based on high-level synthesis techniques is proposed to ease the development of the architecture so that this is applied to different optimization problems.

Chapter 4

The cGAP architecture

4.1 Introduction

In the previous chapter a scalable processor array for accelerating cellular genetic algorithms (cGAs) was proposed. An architecture simulation was conducted to study the convergence of the algorithm, for solving the traveling salesman problem, for different levels of parallelism and configurations of the processor array. Additionally, a custom implementation in synthesisable Verilog hardware description language (HDL) was performed to solve the same problem. However, the effort needed to implement the genetic operations executed in the processing elements (PEs) is high using standard HDL, and thus adapting the array to solve other problems requires a similar implementation effort as these operations are very problem-specific.

This chapter presents all the blocks used to build the common infrastructure that supports the processor array, and are not dependent of the optimization problem, nor the metaheuristic used to solve it. Section 4.2 starts by introducing the complete hardware architecture, named cellular genetic algorithm processor (cGAP), and defines its main blocks. Then, Sections 4.3 and 4.4 describe all the array of PEs and shared memories, and detail the arbitration mechanism implemented so that the PEs access to these memories. All the hardware components required for ensuring the control of the execution of the algorithm are explained in Section 4.5. Finally, in Section 4.6 it is proposed a simple infrastructure using a single random number generator (RNG) to feed all the PEs.

All the blocks described during this chapter have been specified in Verilog HDL, and target a design flow where the problem-specific blocks are implemented with a high-level synthesis flow, thus easing the customization of the cGAP to solve new problems.

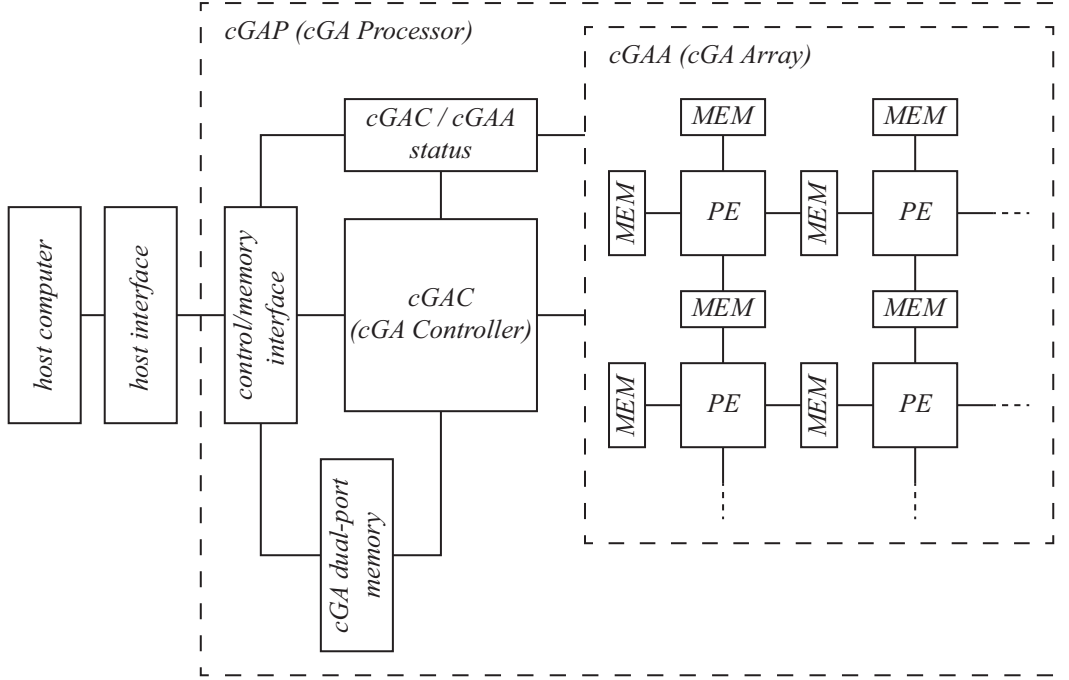


FIGURE 4.1: Overview of the cGA processor (cGAP). A controller (cGAC) communicates with the PEs so that the algorithm is globally controlled. The interface circuit ensure the control and communication with the cGAP from a host computer.

4.2 cGA processor (cGAP) overview

In the last chapter, we have presented a simulation model of a processor array for cGAs, by connecting conveniently processing elements (PEs) and shared memories. However, in a real hardware implementation it is required to endow the circuit with the necessary blocks to control the execution of the algorithm, as well as to monitor the status of the various hardware components. Figure 4.1 provides a simplified overview of the complete hardware architecture used in this work. A block named *cGA controller* (cGAC) interacts with the PEs so that the GA running in each of them can be individually controlled and monitored. In addition, the cGAC is responsible to receive and process commands sent by an external host computer. A control and memory interface circuit ensure the communication between the host computer and the cGAC, as well as it monitors the status of the main blocks, so that the complete hardware is controlled. Additionally, a dual-port memory block is placed between the cGAC and the host computer so that data can be transferred between the exterior and our architecture. In the context of this work, we call *cGA array* (cGAA) to the group of PEs and shared memories as it has been described in Chapter 3, and *cGA processor* (cGAP) to all the hardware blocks required to support the execution of the algorithm (cGAC, cGAA, and others that will be presented throughout this chapter).

By defining the functionality of the PEs and the cGAC, we can adjust the cGAP to solve different optimization problems. In this chapter, we present all the hardware blocks that are independent from the optimization problem, which have been described in Verilog HDL. The PE and the cGAC are specific of each problem and thus must be designed according to the problem's requirements to be solved by the cGA. Additionally, these two blocks are specified in the C++ language for a high-level synthesis design flow as it will be described in Chapter 5.

As we have already discussed, the shared memories of the cGAA are responsible to keep a subset of the pool of solutions. In this work, we call this subset a *subpopulation*¹. Therefore, a memory (and additional hardware to control it) is called *subpopulation memory*. We would like to emphasize that the target hardware device of our work is a Virtex-6 (XC6VLX240T-1) FPGA from Xilinx. This FPGA presents a total of 416 true dual-port block RAMs (BRAMs), each with 36 Kbits [Xile], which means that a maximum of 208 PEs can exist in the cGAP, which represents the maximum allowed level of parallelism. More recent FPGAs from the same vendor, like the Virtex UltraScale, present already an impressive maximum number of 3780 BRAMs each one with also 36 Kbits [Xild]. These numbers show that the cGAP, although requiring a considerable number of subpopulation memories, can be implemented with current FPGA devices.

4.3 cGA array (cGAA)

The scalable processor array proposed in this work presents a regular structure of processing elements connected to shared memories. To build this structure, a block named *cGA cell* formed by a single PE and two memories is replicated several times to form the desired rectangular shape of the cGAA.

Figure 4.2(a) depicts an example of a cGAA with 2×3 PEs (or cGA cells) with a toroidal shape. As it can be seen, the cGA cells are distributed over a bi-dimensional space so that a rectangular and regular structure is built. A cGA cell has four main ports: the north and west ports connect to memories, while the south and east ports connect to the PE of that cell. This way, by connecting the north and west ports of each cell to the corresponding south and east ports of the neighbour cGA cells, the complete cGAA is built. The toroidal shape of the architecture is achieved by connecting the top to the bottom sides of the array as well as the left to the right sides, which ensures a complete regular structure where all the PEs connect to four memories and each memory is shared by two PEs.

¹The concept of a subpopulation in the cGAP must not be confused with a subpopulation of a distributed GA, where the solutions of a subpopulation are evolved, till a certain point, separately.

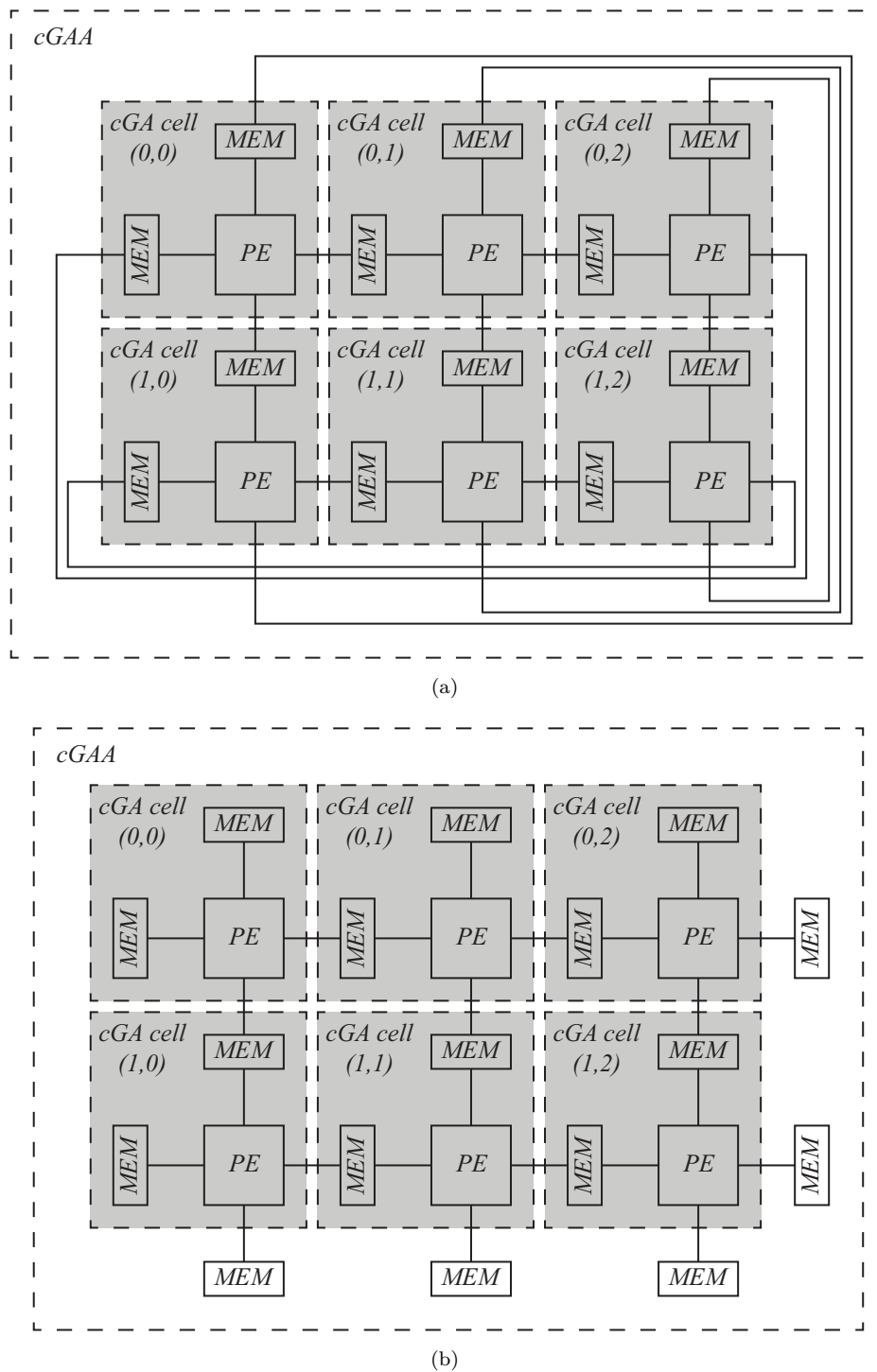


FIGURE 4.2: Hardware configuration of the cGAA with (a) toroidal and (b) non-toroidal shape.

Similarly, a non-toroidal architecture for the cGAA can be constructed with the same cGA cell block. In this case, the opposite sides of the array are not interconnected, and additional memories are placed both on the bottom and left sides of the cGAA as depicted in Figure 4.2(b). This ensures that all the PEs in the array have the same

interface, which ensures that a single PE can be built and replicated in all cGA cells. The memories placed on the borders of the cGAA (top, bottom, left and right) differ from the remaining ones placed inside the array, as they only have a single access port.

Regardless of a toroidal or non-toroidal architecture, the aspect ratio of the rectangular shape for the cGAA is parameterized by defining the number of PEs present in each row and column. This way, each PE (or cGA cell) is identified by $PE_{i,j}$, where i, j represents the row and column indexes of the cGAA. Both examples of Figure 4.2 present a cGAA with configuration of 2×3 PEs, thus with a total of 6 PEs.

4.4 cGA cell

In the previous section we have discussed how to build the cGAA with a basic block called cGA cell. We will now describe how a PE and the subpopulation memory blocks, the two main blocks that constitute a cGA cell and thus the cGAA, interact among each other. Additionally, we will present an arbitration mechanism to prevent data corruption in the shared memories due to possible simultaneous accesses to the same solution by two PEs.

4.4.1 Processing element (PE)

The processing element block of the cGAA is responsible for executing the basic operations of a genetic algorithm like selection, crossover, mutation and the fitness function. In our design, this block is connected to four memories that hold each one a subpopulation. This way, the PE can process the data (solutions) that are present on those memories, to generate new solutions that will replace existing ones, thus implementing the genetic-inspired evolutionary process.

Figure 4.3 presents an overview of the complete interface of a PE. This block connects to the shared memories that hold the subpopulations with four identical interfaces to each one of the north, south, east, and west subpopulation memories. The interface between a PE and a subpopulation memory comprises two main groups of ports: the *mem* port that connects directly to the memory block; and the *request* port that ensures the communication of a simple protocol that allows the safe access to the same memory contents by the two PEs connected to the memory. This mechanism will be explained in the following subsection.

Additionally, each PE has two connections that allow receiving and sending commands, respectively *PE_control_in* and *PE_control_out*. This way, and assuming that those

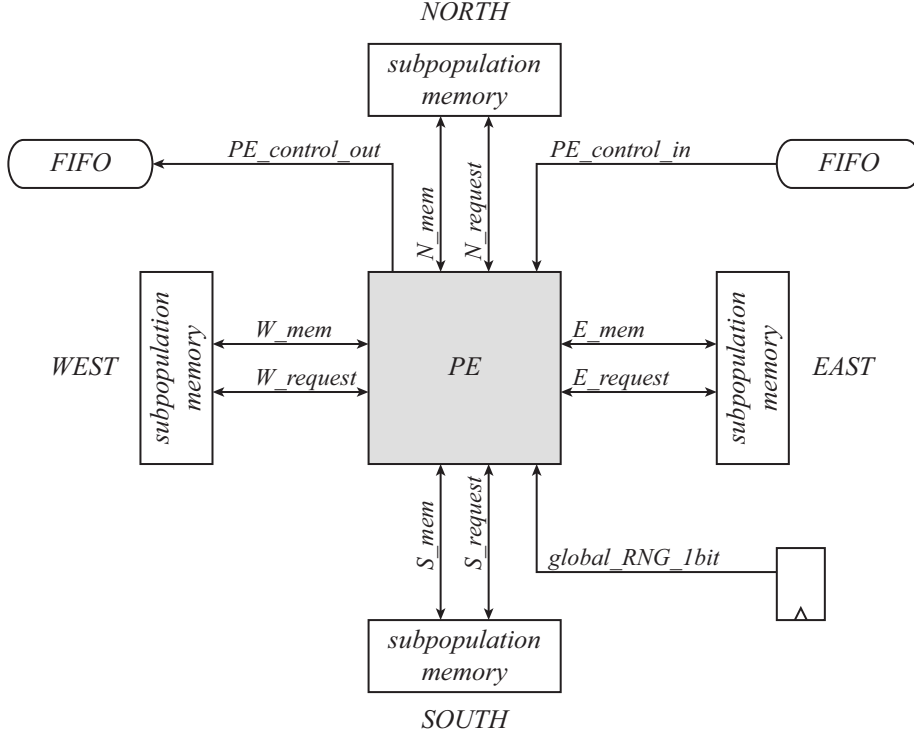


FIGURE 4.3: Processing element (PE) hardware interface.

commands are centrally processed by the cGAC, the execution of the algorithm (the cGA) distributed by the various PEs can be controlled by exchanging information (commands) among the controller and the PEs. These two interfaces are connected to FIFOs that collect the commands and send/receive them through a dedicated communication infrastructure. Details of the controller and the communication will be provided in Section 4.5.

An additional port in the PE receives a single 1-bit random number coming from a global random number generator (RNG) that feeds all the PEs. Therefore, a PE can generate the random numbers required during the execution of the algorithm by collecting data coming from this port (*global_RNG_1bit*). With this approach, it is possible to avoid the replication of RNGs in each PE by having a global RNG feeding sequentially all the PEs. In Section 4.6 the infrastructure of the RNG will be discussed.

4.4.2 Subpopulation memory

The subpopulation memory is responsible for keeping in memory a subpopulation and to provide a mechanism that allows the correct access to the solutions shared by the two PEs. In general, one solution occupies more than one memory location and thus the access to this information by the PEs must be performed so that the data is not corrupted. Figure 4.4 depicts the interface of the subpopulation memory with the two adjacent PEs.

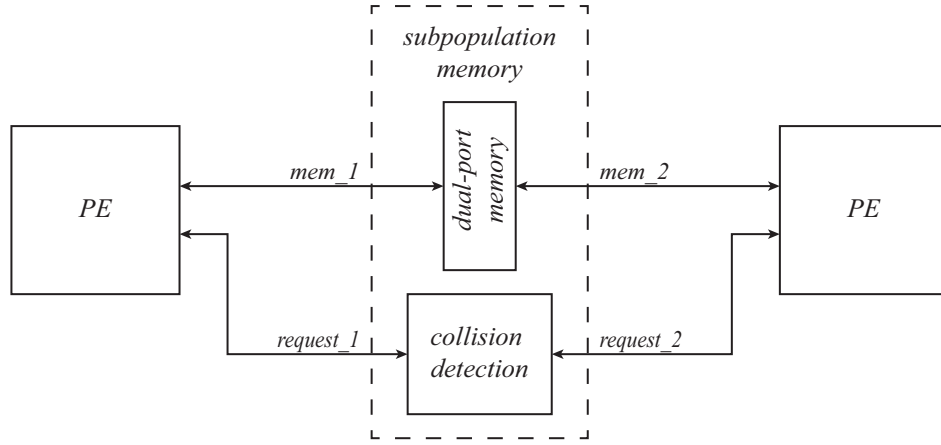


FIGURE 4.4: Subpopulation memory hardware interface.

From the configuration of the cGAA, each subpopulation memory either accesses to the north and south connections of a PE pair, or to the east and west connections of another PE pair (c.f. Figure 4.2). The memory contents in the subpopulation memory are directly and independently accessed by each PE connected to this block through the *mem_1* or *mem_2* ports. Thereby, this memory is implemented using dual-port memory blocks, which allow the simultaneous access to the memory contents by the PEs without any additional circuit. To avoid situations where the same memory contents are being accessed by the two PEs simultaneously, which may lead to the corruption of data, a circuit block called *collision detection* implements a set of verifications that in collaboration with the PEs ensure that the access to the memory is safe.

Figure 4.5(a) shows a possible organization of the subpopulation information kept in the dual-port memory. A solution of the optimization problem handled by the cGA comprises a set of memory words, and it is identified by a unique number in that subpopulation memory.

Before a PE accesses to a given solution, it must undergo a handshake mechanism with the collision detection block to ensure that the access to the memory contents is performed safely. Figure 4.5(b) shows a simplified overview of the collision detection circuit and its connections to the PEs.

When a PE starts the handshake, it issues a request command to the subpopulation memory asking read or write access to a given solution present in the memory. This request handshake mechanism is composed by the following signals: an enable to start the request (*req_en*); the command requesting for access to the memory (*req_cmd*); the number of the solution to which the command applies; and the acknowledge signal granting (or not) to the PE the safe access to that solution (*req_ack*). As a PE can perform either a read-only access or a write access to a given solution, it is possible

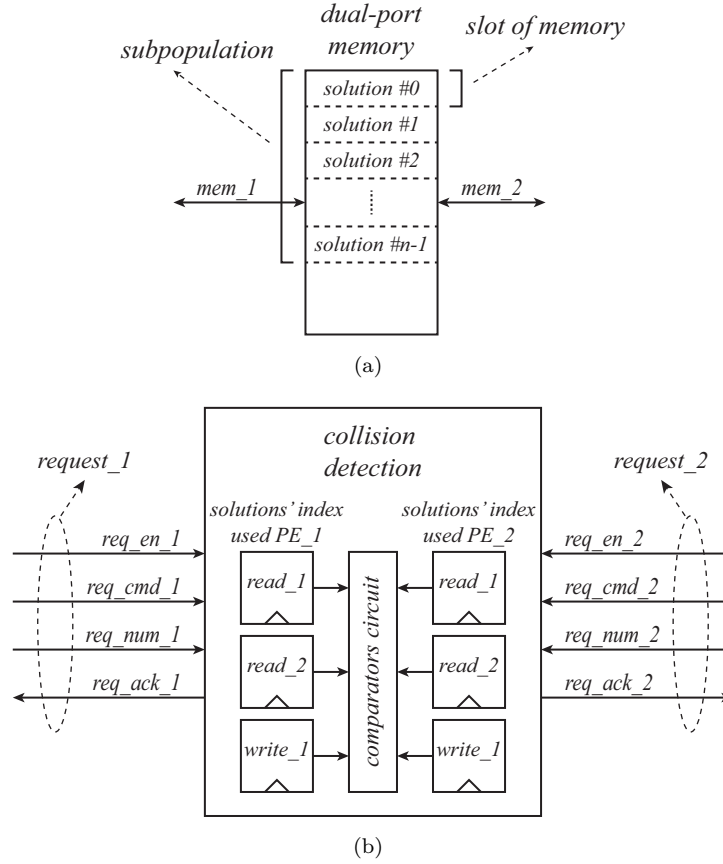


FIGURE 4.5: Details of the subpopulation memory hardware block: (a) possible memory organization in the dual-port memory, and (b) collision detection circuit used to avoid undesired accesses to the solutions' information.

to have a simultaneous access to the same solution only if both accesses are read-only. When a PE wants to write to a solution, the other PE must not access to the same information, either for reading or writing, as the data may not be coherent or may become corrupted.

In most of the cases, a genetic algorithm requires two solutions (due to the crossover operation) from the population to generate a new one. This means that during a generation in a PE two solutions will be read from the subpopulation memories and one will be overwritten with the new solution. Therefore, we have chosen to provide to the collision detection circuit the capability to keep track of a maximum of two solutions being read and one solution being written at the same time by each PE connected to the subpopulation memory. With this approach, in a generation of a new solution, a PE can choose freely any solution (subject to the constraints aforementioned per memory) from any one of the 4 subpopulation memories. In Figure 4.5(b) the three registers (*read_1*, *read_2* and *write_1*) per PE connection monitor the corresponding solutions (if any) being accessed by the two PEs. After a PE requests an access to the memory, the two

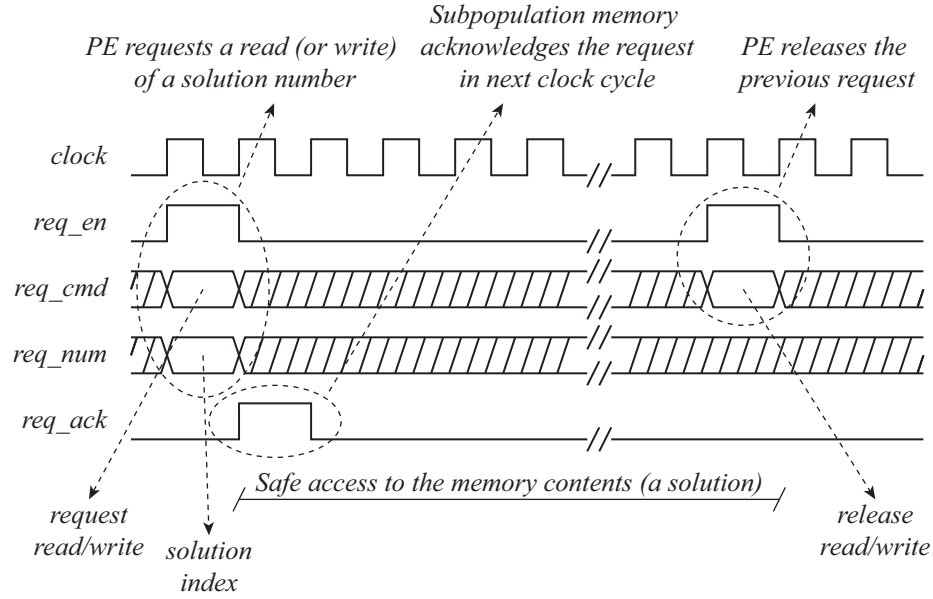


FIGURE 4.6: Example of the handshake protocol between a PE and a subpopulation memory.

groups of registers are properly compared to detect any undesired memory access request so that the acknowledge signal is properly sent to the PE.

An example of the handshake between a PE and the subpopulation memory is depicted in Figure 4.6. The PE starts by issuing the desired request command (read or write) to access a particular solution in that subpopulation providing the solution index. In the next clock cycle, the subpopulation memory acknowledges the access, granting the safe access to that particular slot of memory. In the case where the acknowledge signal is not asserted, the request fails and the PE must not access that solution. When the access previously granted to a solutions is not needed anymore, a new command is issued by the PE to release the request. In this case, it is not required to specify the number of the solution being released as this command only cancels the operation. Additionally, the subpopulation memory does not need to acknowledge a release command as this is always accepted.

Table 4.1 provides the results of a request by a PE to access a given solution in the subpopulation memory when the other PE is also accessing to the same information. Essentially, it is possible that a read operation occurs simultaneously in both PEs, but when a write operation is being performed by a PE, the other cannot access to the same solution.

We should emphasize that the collision control circuit does not provide mechanisms to avoid undesired accesses to the memory contents, which can happen if the handshake

TABLE 4.1: Request results of the handshake protocol among a subpopulation memory and the two PEs connected to it.

activity on a PE		result of request opposite PE	
read	write	read	write
0	0	accepted	accepted
1	0	accepted	rejected
0	1	rejected	rejected
1	1	rejected	rejected

mechanism between a PE and the subpopulation memory is ignored. The PE is responsible to trigger and analyse the results of the handshake.

4.5 Control infrastructure

In this section we will describe how the complete cGAP is controlled during the execution of the algorithm. We will explain how the cGAC block sends and receives commands through a dedicated communication infrastructure to each PE, allowing this way a centralized control of the entire flow of the algorithm. Additionally, we will explain how the cGAP communicates with the exterior.

4.5.1 cGA controller (cGAC)

The cGAC block is responsible to manage all the commands sent to the PEs and received from the PEs, and thus it can control the evolution of the algorithm. Additionally, the cGAC also processes the commands sent to the cGAP by an external host computer (e.g. by programming registers via software), acting this way as a bridge between the cGAA (the PEs) and the host.

Figure 4.7 shows an overview of the interface of the cGAC. On the right-hand side, the two connections *cmd_from_cGAA* and *cmd_to_cGAA* are responsible to receive and to send respectively any command to the cGAA. Both interfaces are directly connected to FIFO hardware blocks that collect the information transferred between the cGAC and a communication infrastructure, as it will be explained in the next subsection. On the left-hand side, the two ports *cmd_from_SW* and *cmd_end_ack* guarantee the correct communication of commands sent by the host to the cGAP. In this case, the cGAC receives the commands through the first port and asserts an acknowledge signal through the second port when that command has been processed.

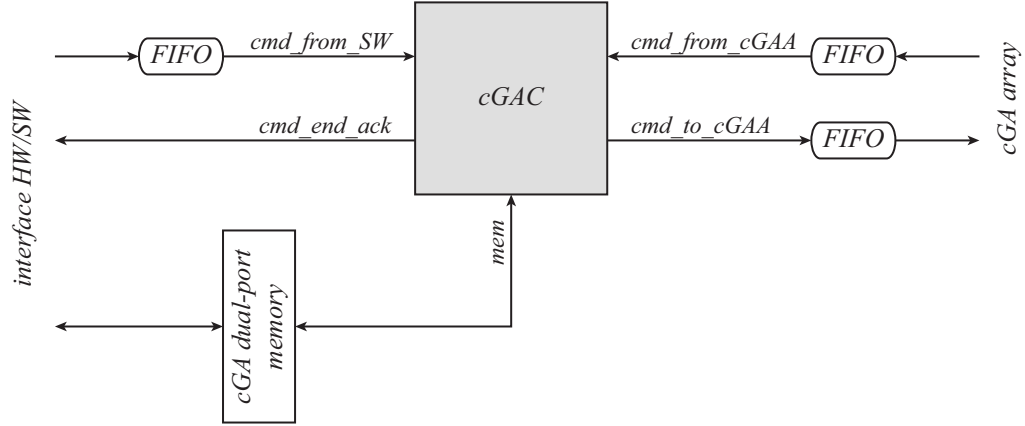


FIGURE 4.7: Cellular genetic algorithm controller (cGAC) hardware interface.

A dual-port memory (*cGA dual-port memory*) connects directly one of its port to the cGAC and the other port to the exterior of the cGAP as depicted in Figure 4.7. By doing so, it is possible to transfer data between the host and the cGAC without sending that information sequentially through commands as explained previously. In addition, if required this memory can be used by the cGAC during the execution of the algorithm, for instance, to keep track of the evolution of the fitness value in each PE.

4.5.2 Communication infrastructure

The communication infrastructure present in the cGAA, which is a simple form of a network on chip (NoC), allows the communication between the cGAC and all the PEs, as it is illustrated in Figure 4.8. When sending a command from the cGAC to a given PE, the information is guided through a sequence of *routing FIFOs* that route the information till the destination. In the figure, it is shown an example of sending a command to the PE positioned at coordinates (1,1). First, the command moves vertically through the cGAA configuration till it reaches the first coordinate number and then, in a similar way, it moves horizontally till it reaches the second coordinate number where the PE is found.

To reduce the complexity of the steering logic that implements the routing of commands, we have opted to build this as a parallel unidirectional bus. Other solutions would be possible, however for the target application it is only required a communication from the cGAC to the PEs and vice versa. The direct communication among PEs using the communication infrastructure is not implemented since it is not required by the cellular GA. This way, we keep the logic as simple as possible.

Figure 4.9 shows the composition of a command information that traverses in the communication infrastructure. The first three fields (looking from the left) constitute the

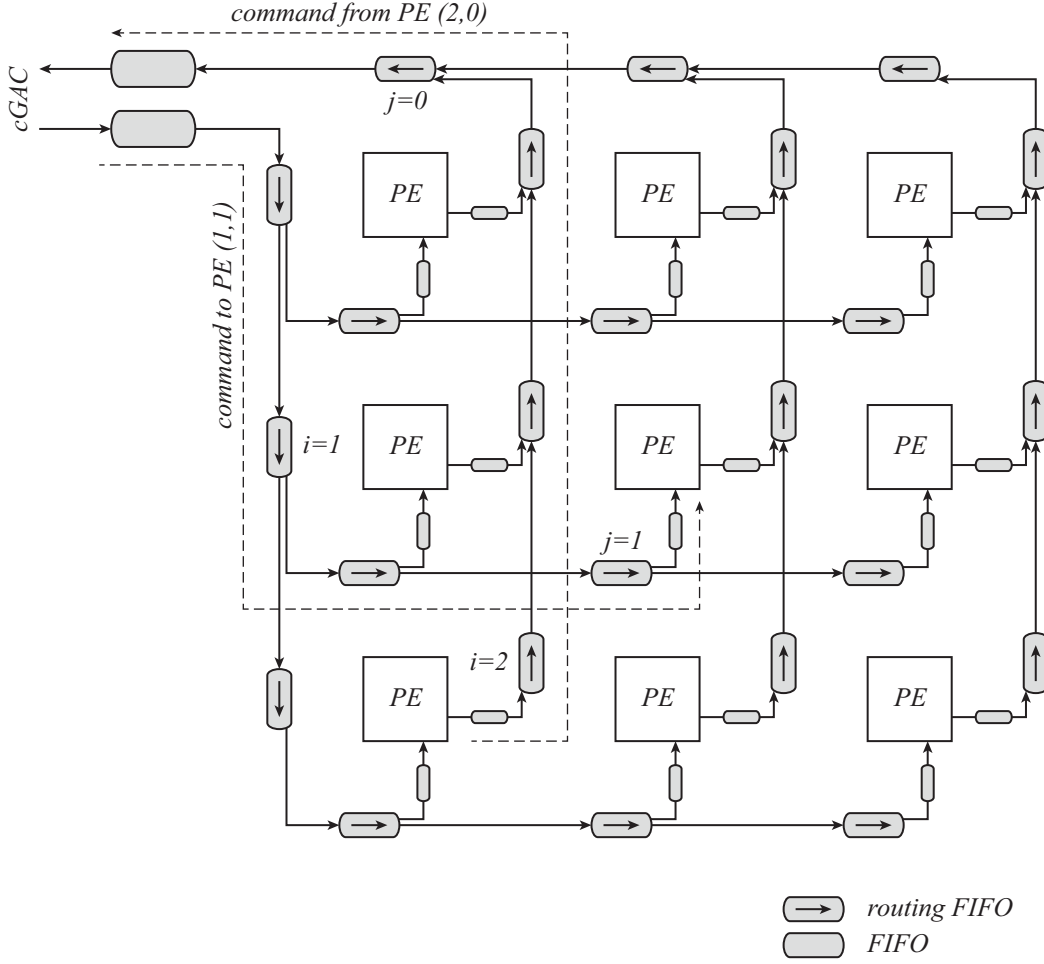


FIGURE 4.8: Overview of the communication infrastructure hardware used for the communication among the cGAC and the PEs.

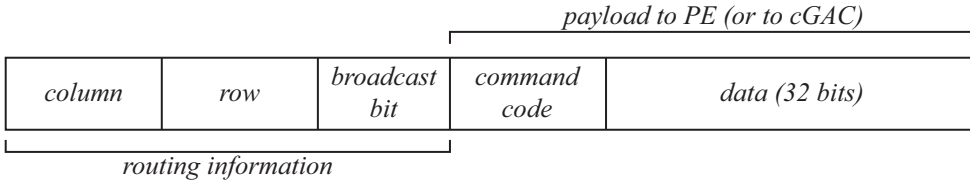


FIGURE 4.9: Command information used in the communication infrastructure.

information required to route the command till the PE of destination, where the *row* and *column* identify the coordinates of the PE, and the *broadcast bit* represents a command to be broadcasted to all the PEs. The fields *command code* and *data* carry the information (the payload) to be sent to the PE or to the cGAC. All these fields have the minimum number of bits necessary to represent them, which are parameters during the synthesis of the circuit; the *data* field is fixed to 32 bits which allows the communication of data (via 32-bit registers) from the host computer to the cGAC and then to the PEs without the need to split that information into several commands.

For sending a command from a given PE to the cGAC, a similar routing mechanism

has been adopted where the command is sent vertically and then horizontally through the cGAA as it can be seen in Figure 4.8. However, in this case only a single possible path exists between a command sent by the PE to the cGAC and thus the identification of the source PE is added by the routing FIFOs blocks as depicted in the figure for an example of a command from PE (2,0). This way, the cGAC knows from where the command comes from. The broadcast bit is not used with commands coming from the PEs.

The routing FIFO blocks guarantee the correct route of the command to the next routing FIFO according to the PE of destination and, additionally, have a register (acting thus as a FIFO) that preserves the data if the next block has data being processed. In turn, the blocks FIFO are positioned always at the end and beginning of any possible path in our communication infrastructure, that is, performing the interface between a PE or the cGAC and a routing FIFO. The two FIFOs have a set of control signals, as it is illustrated in Figure 4.10, that allow the passage of the information from one to another block. These signals check whether the next block is available for receiving data and, if this is the case, the data is passed to it.

This infrastructure can easily suffer from network congestion, where commands need to be retained in the FIFOs till the next block is available to consume the data. When sending commands from the cGAC, this situation arises when a PE receives the data and does not consume it immediately as it may be performing other operations. Meanwhile, if more commands are sent to the same PE, the FIFO connecting to that PE gets full and the commands start to be kept in the routing FIFOs throughout the path between the cGAC and that PE. In the limit, the cGAC will not be able to write to the communication infrastructure as the FIFO connecting to it also gets full. In such situation, the communication will only start to unlock as the PE starts to consume the commands. A similar situation occurs in the communication connecting the PEs to the cGAC. Additionally, network congestion may also arise when commands to (or from) different PEs need to traverse in the same routing FIFO at the same time.

The communication infrastructure ensures that no deadlocks occur in the commands sent from the cGAC and PEs as there is only a single path between the source and the destination. Moreover, despite all the network congestions that may appear, in any circumstance a command will be lost. All the FIFO blocks use the same control signals to check if they are receiving (or sending) to other blocks and, if congestion occurs, the FIFO halts the passage of information till the data is consumed. Moreover, the connections of the PEs or the cGAC to the FIFOs also have the same control functionality (see Figure 4.10), thus ensuring the same command transfer approach.

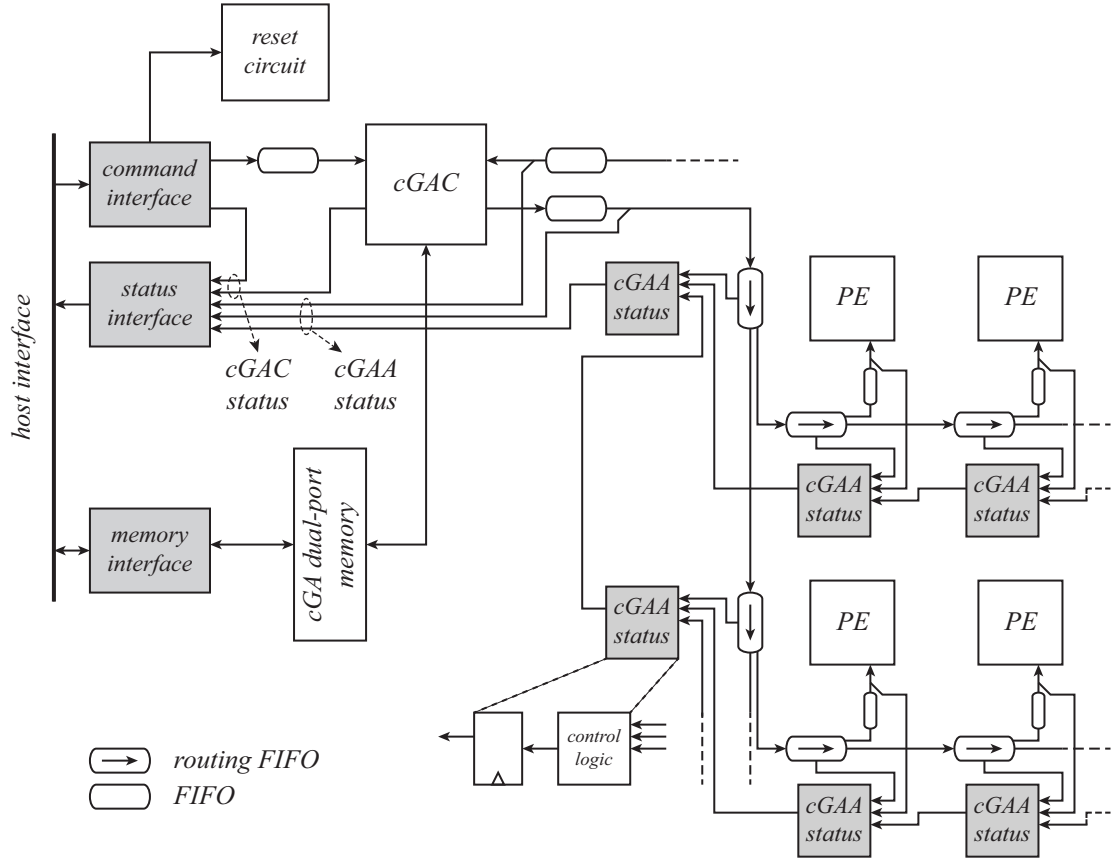


FIGURE 4.11: Overview of the interface circuits used to ensure the communication and control of the cGAP by a host computer.

4.5.3 cGAP interface

Figure 4.11 illustrates the interface circuits of the cGAP that ensure the communication and control of the cGAP by a host computer (or any other hardware block that controls the cGAP). In our design there exist three main interface blocks: the *command interface*, the *status interface*, and the *memory interface*.

The command interface ensures the passage of commands from the host to the cGAP. Therefore, this interface allows the command communication from the host to the cGAC that, in turn, can forward those commands to the PEs. All the commands received through this circuit are sent to the cGAC to be processed, with the exception of the command to reset the cGAP (defined by the macro `CMD_RESET_CGAP` in the description of the hardware) which activates the reset circuit. In our design, the host must send a new command only when the previous one has been processed by the cGAC, thus avoiding congestion problems in this interface.

The status interface informs the host computer the activity status of the cGAP. For that, this circuit keeps track, separately, of the activity in the cGAC and in the cGAA.

The cGAC is considered to be in a processing state if it is processing a command sent from the host. Therefore, the host knows when the cGAC is available to consume a new command, which is required to avoid congestion when the host is sending a new command to the cGAC.

The activity in the cGAA, which indicates if any PE is in a processing state and if any command is traversing in the communication infrastructure, indicates to the host when the cGAA has finished processing the cellular genetic algorithm. To accomplish this, the communication infrastructure is continuously monitored to check if any command is being transmitted from the cGAC to the PEs and vice versa. In addition, all the PEs are monitored to verify if they are in a processing state, which can be done by checking the control signals of the FIFOs that feed the PEs. When a PE is asking data to the FIFO and this one is empty, the PE enters in a blocking state where it continuously waits for a new command from the FIFO. In such situation, we know that the PE is not processing data anymore. The blocking state of the PE will be explained in Section 5.2.1.

To check if any command is being transmitted, we verify all the FIFOs and routing FIFOs only from the network that connects from the cGAC to the PEs. The circuit that detects activity in the PEs has the same clock cycle delay as the network that sends commands from the PEs to the cGAC. Therefore, in this case only the last FIFO that connects to the cGAC needs to be checked for the desired functionality and to account situations where network congestion happens.

Regarding the memory interface block, this ensures the communication with one of the ports of the cGA dual-port memory as depicted in Figure 4.11. The access to this memory can be performed simultaneously by the cGAC and by the host at any time during the execution of the algorithm. Hence it is the responsibility of the designer of the complete system (software running in the host and cGAP) to ensure that memory data corruption does not occur.

4.6 RNG infrastructure

A genetic algorithm relies heavily in random numbers during its execution. As the cGAA has several PEs that execute each one a GA, we propose an infrastructure with a single random number generator (RNG) capable of feeding all the PEs, avoiding this way the replication of local RNGs blocks in all the PEs. Figure 4.12 depicts the architecture of the proposed random number infrastructure for the cGAA. A single bit random number is generated in the RNG block that feeds a shift register where the output of each register connects to each PE as it can be seen in the figure. When a PE requires a new random

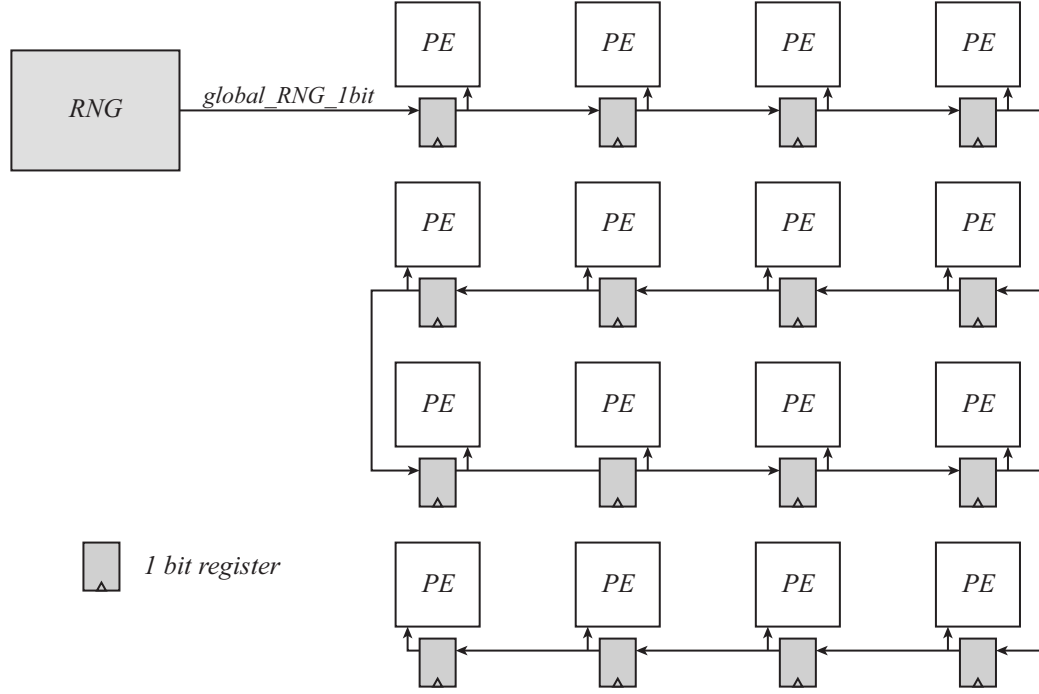


FIGURE 4.12: Random number generator (RNG) infrastructure hardware used in the cGAA.

number, it must read the corresponding port of the random number as many times as the number of bits required to form the desired random number.

In the proposed architecture there is not any control mechanism in the shift register that carries the random numbers and therefore the RNG block is continuously generating a new random bit at each clock cycle. By doing so, we simplify this hardware infrastructure to a single bit register per PE, while not compromising its functionality as the target is to feed the PEs with random generated bits.

With this infrastructure, it is clear that a correlation may exist in the random numbers obtained among all the PEs as the same random bits carried by the shift register can be sampled by any PE. During the evolution of the cGA all the PEs are executing each one an identical genetic algorithm and thus it is questionable the impact that such method of generating random numbers inside a PE may have in the quality of the final solution obtained by the algorithm.

Nevertheless, the PEs take different processing times for generating solutions, and thus they will not be synchronized among each other during the execution of the cGA. As described in Section 4.4.2, a PE must undergo an handshake mechanism before accessing a subpopulation memory to prevent undesired accesses to the same memory contents. If we consider that the PEs select the solutions from the memories with a random algorithm (which is the usual in a cGA), even if they sample from the shift register exactly the

same random numbers during the initial generations, eventually an unwanted access will occur that must be solved by one of the PEs. This results in an increase in the processing time during the handshake phase in that PE and, as a consequence, the next sample instants of the random numbers will change when compared to the other PEs. Therefore, as the handshake algorithm takes different times in the PEs, the generations will not be synchronized and a correlation among the random numbers obtained by the PEs is reduced.

Moreover, a generation in a genetic algorithm may have different processing times depending on the operators implemented and on the solutions being processed. For instance, the MPX crossover operator [MGSK88] copies a random part of a solution to generate a new one, as we have discussed in Section 3.4.1. As the extension of this copy is variable, different processing times will be required to process this operator. Indeed, depending on the optimization problem handled by the algorithm, potentially all the operations may have different processing times as the solutions handled by the PEs are different. This situation together with the handshake required to access the subpopulation memories, helps to ensure that during the execution of the cGA the PEs will be continuously out of synchronization regarding the instant when the generations of the algorithm start.

A few studies have been performed that evaluate the influence of the quality of a RNG in the performance of a GA. In [MF99] the authors analysed the influence of several RNGs in the performance of the GA and found that there is not a clear correlation between the quality of the RNG and the performance of the GA. Indeed, results have shown that good quality RNGs can deliver better or worse performance in the GA depending on the test suite function applied to the algorithm. Low quality RNGs also have led to similar results. However, in [CP02] the author has found that the quality of the RNG used to initialize the population has a critical impact on the performance of the algorithm, but for the other operations (crossover and mutation) also does not seem to affect the results. Another study has been performed in [TÄW11], this time for a differential evolution (DE) algorithm, a subclass of evolutionary algorithms (EAs) like the GA. In this work, the authors also found strong evidences that the algorithm performs identically for several RNGs.

Therefore, even though there is a possible correlation among the random numbers acquired by the PEs, this impact on the performance of the algorithm is expected to be low (or even negligible). Furthermore, the proposed RNG infrastructure aims mainly to reduce the hardware resources used by the cGAP and thus, if desired, it is possible to build a dedicated RNG in each PE as part of its functionality.

Nevertheless, this infrastructure must be used mainly during the evolution of the cGA, namely for generating the new solutions using the known genetic operators like selection, crossover, mutation and replacement. Regarding the generation of the initial population, the use of this RNG architecture in the PEs can be harmful to the quality of the cGA. As an example, if a command is issued by the cGAC to all the PEs to start calculating their initial subpopulations, they will start to work in parallel and acquire the same random bits from the shift register that carries the random numbers. As a result, a strong correlation will exist in the solutions generated by each PE (eventually, some may generate the same solutions). This situation is unacceptable for a good quality cGA as it is important to have diversity in the initial solutions to perform a better exploration of the search space. Therefore, the initial population can be generated previously (e.g. by software) and sent to the cGAP at the beginning of the algorithm. Another possible implementation, this time using the infrastructure, is to calculate the initial population sequentially in each PE, thus avoiding unwanted correlations among the random numbers of all the PEs.

4.6.1 RNG block

The RNG block in Figure 4.12 is responsible to feed all the PEs with the required random numbers as explained in the previous section. There are several known RNGs like the Combined Tausworthe [L'E99], the Mersenne Twister [MN98] or Twisted Generalized Feedback Shift Registers (TGFSRs) [MK94]. These generators have been developed to be implemented in software and thus they are not optimized for a hardware implementation. For the hardware RNG block of our cGAP we decided to choose a RNG based on cellular automata (CA) techniques that is adjusted for a hardware implementation as described in [STCS02].

Figure 4.13 depicts an overview of the implemented RNG and explains how the CA is built to form the desired generator. This CA is formed by a 1-dimensional space network (a ring) with 64 cells. The state of each cell (0 or 1) relates with the previous states of the cells positioned at $\{-7,0,11,17\}$ relatively to it and is defined by the rule 50745 as explained in the figure. From the 64-bit random number and as described in [STCS02], we use the odd-numbered bits to generate a 32-bit random number. The bits that constitute this last random number feed sequentially the 1-bit shift-register of the RNG infrastructure of the cGAA.

According to [STCS02] this configuration of the CA passes all the DIEHARD tests which are a battery of statistical tests for measuring the quality of a RNG [Mar95]. Additionally, the cycle length of the RNG is approximately 2^{35} if the CA starts in a state where

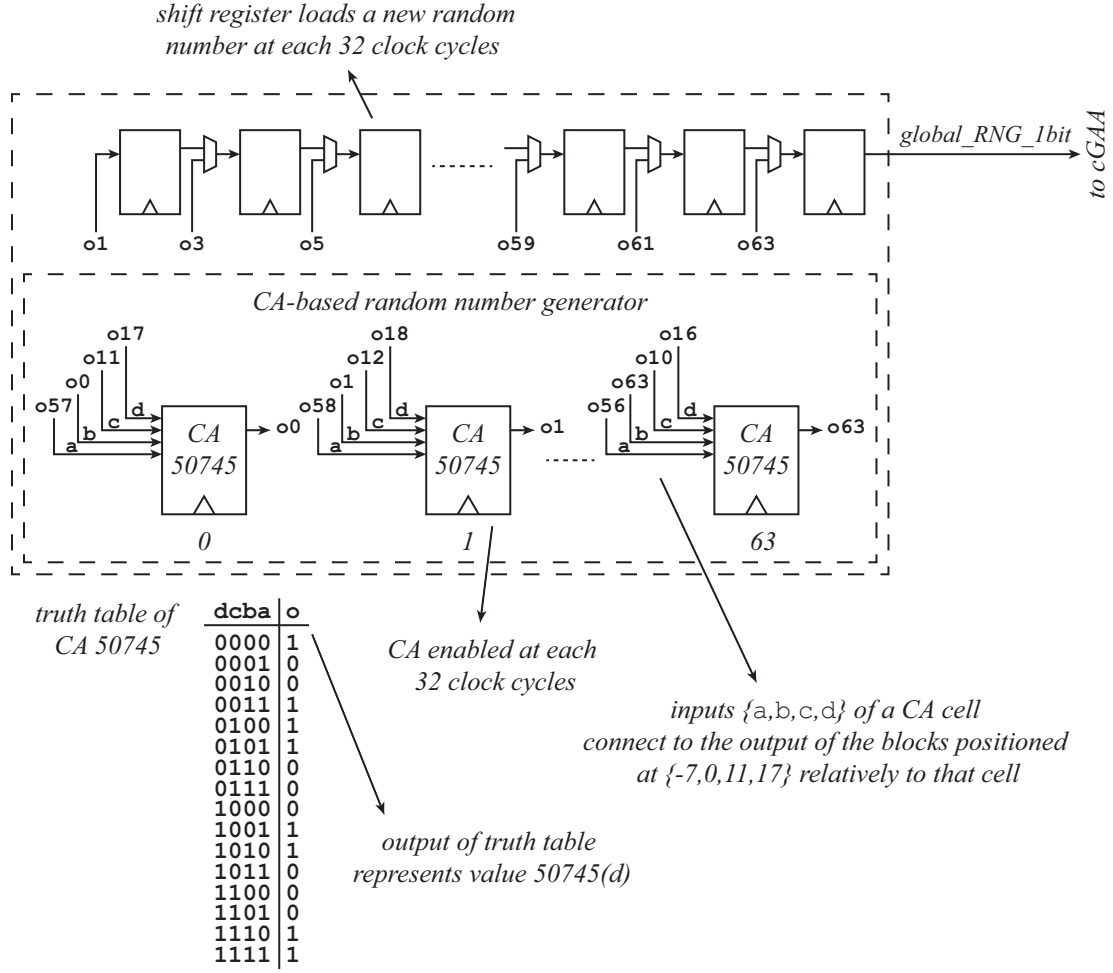


FIGURE 4.13: RNG based on a cellular automata, with a 1-dimensional ring topology with connectivity $\{-7,0,11,17\}$ and rule 50745, used to feed the RNG infrastructure of the cGAA.

only a single cell is 1. Different initial states may lead to different cycles and thus they must be avoided unless they are previously studied. For this reason, we have chosen a single seed (1) for the implemented RNG, which is loaded when the reset signal asserts in the cGAP.

4.7 Summary

In this chapter we have presented the details of the cGAP that are necessary to build a generic hardware template which is independent of the optimization problem solved by the metaheuristic. Therefore, we have specified all the hardware blocks, except the functionality of the PEs and the cGAC, which must be adapted to the operations required by each problem.

We have explained how to construct a configurable array of PEs and subpopulation memories, and how the contents of these memories (the solutions) can be accessed simultaneously by two PEs. For that, a handshake protocol between a PE and a subpopulation memory has been defined that allows that a PE accesses safely and at the same time to a maximum of two solutions for reading and one for writing per memory. Additionally, we have specified a communication infrastructure that allows that commands are transmitted between the cGAC and the PEs. Therefore, this infrastructure allows that the algorithm execution is centrally controlled by the cGAC. Finally, a global RNG infrastructure has been proposed capable to provide random numbers to all PEs.

In the next chapter, we will focus on the description of the PE and cGAC hardware blocks using high-level synthesis, which concludes with a design flow of the complete cGAP.

Chapter 5

The cGAP design methodology

5.1 Introduction

In the previous chapter all the problem independent support components that constitute the cellular genetic algorithm processor (cGAP) were detailed. These components form the underlying infrastructure to hold the processing elements (PEs) and the cellular genetic algorithm controller (cGAC), and are configured with a reduced set of parameters to define the organization of the array and the subpopulation memories.

To facilitate the design of custom and specialized PEs, a high-level synthesis (HLS) design flow has been adopted, allowing the customization of the core genetic algorithm implemented in a PE in the C++ programming language, accepted as input by commercial HLS tools. This chapter presents in Section 5.2 a general algorithm and interface structures that build a template model in HLS to describe a PE functionality. Besides the PEs, the cGAC that manages the entire processor array is also customized using the same methodology, thus implementing control specific commands required for each problem. Additionally, Section 5.3 presents the communication from a host processor (a MicroBlaze) to the cGAP as memory mapped device. The complete design flow that builds the cGAP connected to the host is presented in Section 5.4, with the discussion of the main configuration parameters to set up the cGAP framework for a new problem.

5.2 Specification for high-level synthesis

In this section, we detail how the PE and cGAC hardware blocks are described using high-level synthesis tools, to be rapidly adapted to different optimization problems. We focus on the interface and algorithms structures of these blocks, so that they interact

correctly with the remaining hardware blocks described in the last chapter. The target HLS tool is Catapult HLS¹ from Calypto Design Systems, which uses a subset of the standard C++ language to describe the hardware [Cal].

5.2.1 Processing element

A PE of the cGAP implements the genetic algorithm on its local subpopulations. To perform this, a PE must comply to the access rules established for the subpopulation memories (see Section 4.4.2) and implement a set of behaviours in response to commands issued by (or sent to) the cGAC.

5.2.1.1 Algorithm structure

The overall structure of the PE specification in Catapult HLS is represented by the flow chart in Figure 5.1. Besides an initialization phase, it consists in two main stages. First, all the parameters required for the execution of the PE are configured according to commands sent by the cGAC. After this, the genetic algorithm is executed by entering an iterative loop where new solutions are generated according to the specified genetic operators. It should be emphasized that the algorithm presented in the flow chart of the figure is the basic structure of a PE algorithm and thus it can be changed by adding more functionalities to target specific optimization problems or different optimization strategies.

The initialization phase happens immediately after the activation of the global reset action, where all the variables are loaded with known initial values. After that, the PE enters the stage where it reads and processes commands sent by the cGAC to define a set of mandatory configurations. During this phase, the reading of a new command is performed with *blocking reads*, stalling the hardware till it receives new data in the corresponding interface port. This process is repeated so that all the necessary commands are received to configure the data required to run the algorithm. These data can comprise, for instance, definition of parameters like the number of decision variables or the data necessary for computing the fitness function of a particular instance of the problem.

As Catapult HLS performs a blocking read during the configuration phase of the PE, the hardware is continuously entering into a stall state as it waits for receiving a new command from the cGAC. A command must be issued to conclude the configuration phase and move forward the PE for the execution of the algorithm.

¹Catapult HLS is commonly known as Catapult C.

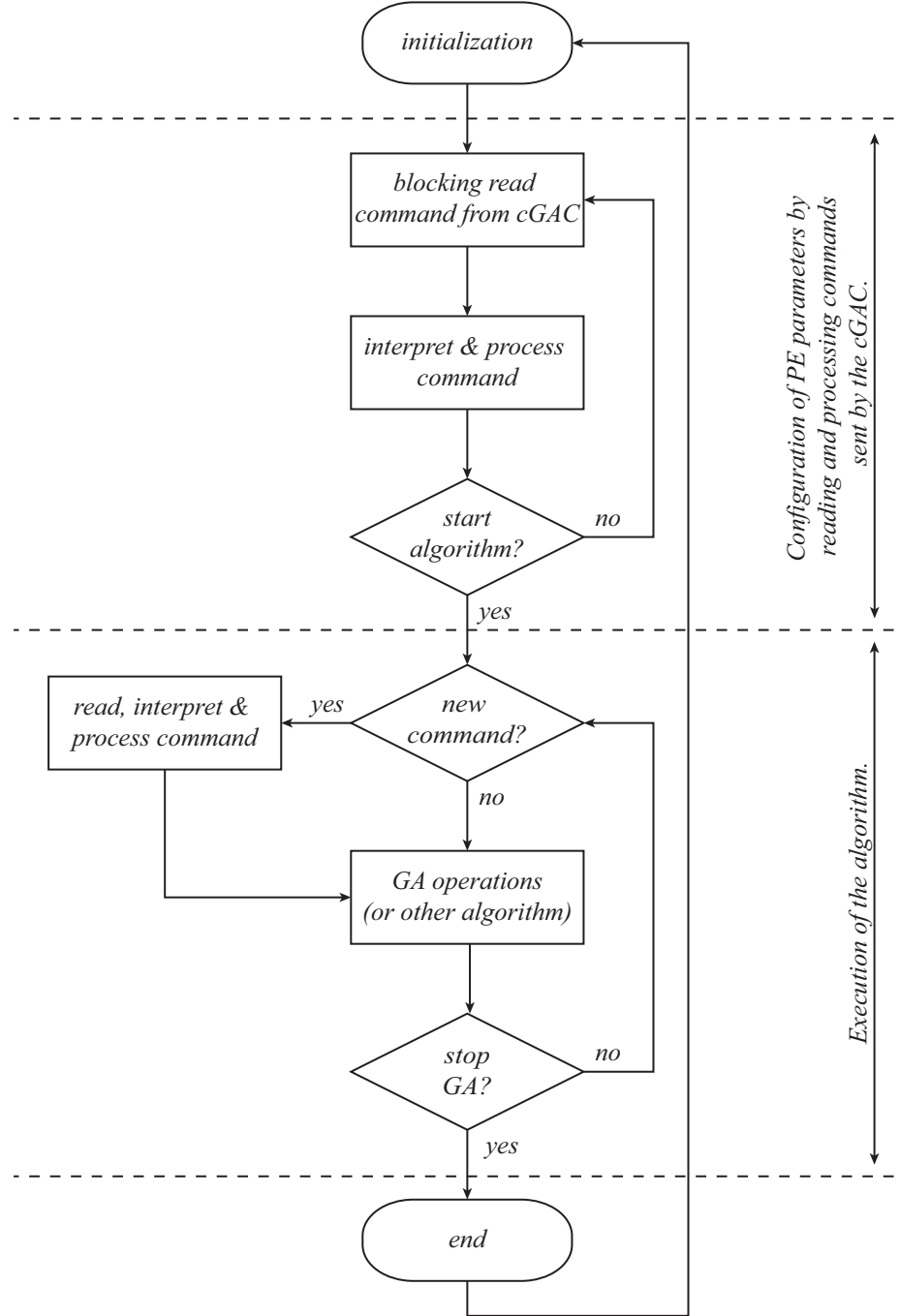


FIGURE 5.1: Algorithmic structure used by Catapult HLS to describe the PE.

In the execution phase of the genetic algorithm, the PE enters an iterative loop that makes the algorithm evolve. Usually, in each iteration the known operations of a genetic algorithm like selection, crossover, mutation, fitness evaluation, and replacement are applied to the solutions in the PE neighbourhood to generate a new one that may replace existing solutions. While this procedure is executed, the PE continuously pools the command port, performing *non-blocking reads* to handle commands eventually sent by the cGAC. Example of such commands are stopping the PE, request its best fitness,

or modify configuration parameters.

When the PE reaches the end of the complete algorithm, by receiving a stop command or exhausting the number of iterations, it automatically returns to the initialization phase as it is illustrated in Figure 5.1. This is the normal behaviour of a hardware block built by Catapult HLS, which continuously loops the sequence of C++ statements declared in the specification of the top level function. Then and as explained previously, the PE stalls waiting for new commands from the cGAC.

5.2.1.2 Interface

The interface of the processing element described in C++ used by Catapult HLS is shown in Listing 5.1. This interface matches, as expected, the one used to describe the cGA cell hardware block as discussed in Section 4.4.1. For the high-level description of this interface in C++, four main groups of ports are used to interact with the different types of hardware blocks: subpopulation memories (data and handshake ports), control, and random number. We will briefly discuss them during this section, and provide basic examples of how this ports can be accessed using the appropriate C++ methods of Catapult HLS.

```

1 void PE(
2     // subpopulation memories data interface
3     ac_int<MEM_WIDTH,false> N_mem[MEM_SIZE],
4     ac_int<MEM_WIDTH,false> S_mem[MEM_SIZE],
5     ac_int<MEM_WIDTH,false> E_mem[MEM_SIZE],
6     ac_int<MEM_WIDTH,false> W_mem[MEM_SIZE],
7
8     // subpopulation memories handshake interface
9     request_channel &N_request,
10    request_channel &S_request,
11    request_channel &E_request,
12    request_channel &W_request,
13
14    // control interface
15    ac_channel<command_type_cGA> &PE_control_in,
16    ac_channel<command_type_cGA> &PE_control_out,
17
18    // random number interface
19    ac_channel<bool> &global_RNG_1bit
20 )

```

LISTING 5.1: C++ description of the PE interface to be used by Catapult HLS.

Subpopulation memories data interface

This interface ensures the connections between the PE and the memories that keep the local subpopulation of the cGA (cf. Figure 4.4). Therefore, four different memory interfaces are required to access each memory (north, south, east and west).

In Listing 5.1 we can see that Catapult HLS performs the interface to arrays (or hardware memories) with `MEM_SIZE` words of a data type `ac_int<MEM_WIDTH, false>`. This data type is specific of Catapult HLS and, for the example presented, it represents an unsigned integer variable with width equal to `MEM_WIDTH` bits. In the description of the PE algorithm the access to these memories is straightforward since they can be manipulated using the C++ operators for arrays.

Subpopulation memory handshake interface

To implement the subpopulation memory handshake interface, we have developed a C++ class called `request_channel` which hides the details of the arbitration protocol between a PE and the subpopulation memory (see Section 4.4.2). The C++ code implementation of this class can be found in Appendix A.2. The class contains two data members, both `ac_channel` data type from Catapult HLS, which are responsible to manage the signal transactions sent from the PE to the subpopulation memory and vice versa. The methods of the class, with the correct design constraints in Catapult HLS, ensure the desired protocol between the two hardware blocks. The class template `ac_channel` used by Catapult HLS allows to synthesize an interface with a FIFO, which guarantees that the compiler will not reorder the instructions (read or write accesses) as a result of optimizations and, additionally, all the instructions are executed even if the data is not explicitly used in the algorithm.

A PE has four connections of data type `request_channel`, each one for the four subpopulation memories that the PE accesses to. The subpopulation memory handshake interface is generated by Catapult HLS by using exclusively the appropriate methods of this class. In Section 5.2.1.3 we will discuss all the details to use correctly this interface so that the subpopulation memories are correctly accessed while performing reading and writing operations on the solutions.

Control interface

The control interface is responsible for the communication between the PE and the cGAC. It comprises two distinct ports that allow the interface with the FIFOs of the communication interface circuit: one for sending and other for receiving commands from the cGAC (cf. Figure 4.8). Therefore, for this interface we have used the `ac_channel` data type of Catapult HLS to communicate with the FIFOs.

A dedicated C++ class, named `command_type_cGA`, has been developed to implement the access to these ports. This class essentially manipulates all the required fields (cf. Figure 4.9) that are used to ensure the communication of a command in the communication infrastructure. The implementation code of this class can be found in Appendix A.1, together with a description of all the methods developed to manipulate a `command_type_cGA` data type variable (see Table A.1).

Although in the `command_type_cGA` data type implementation we have included the routing information of a command (the coordinates of the PE: *row* and *column*, and the *broadcast bit*), these fields do not need to be specified by the PE. Indeed, all this specific information is used only by the communication infrastructure to guide the command to the destination. When a PE receives a command, only the payload (*command code* and *data*) is delivered to the PE. Similarly, when a PE sends a command to the cGAC no routing information is added by this to the command. The routing is handled by the communication infrastructure and the PE does not need to provide any routing information and also it will not receive any routing information. This results from the fact that the source/destination of data is always the unique cGAC.

The class `command_type_cGA` is also used for describing the cGAC functionality where, besides the payload information, the routing data needs to be defined and processed. Therefore, we use the same class for manipulating the commands for both PE and cGAC.

In the following example in Catapult HLS we show how to write a command to the corresponding control interface port:

```
command_type_cGA cmd_type;
cmd_type.set_command(1);           // sets command code value
cmd_type.set_data(53);             // sets data value
PE_control_out.write(cmd_type);    // writes command to port
```

First, a `command_type_cGA` variable is manipulated by using the appropriate class methods to set the command and data values. This data can then be written to the output port by using the method `write` which belongs to the class `ac_channel`. If the FIFO that receives the data is full, the process stalls till the FIFO has available space to receive the data.

To read from the control interface port, the method `read` of the class `ac_channel` is used as the following example shows:

```
cmd_type = PE_control_in.read();
```

In this case, a blocking read is implemented which means that the FIFO read process will be blocked until data is available.

The implementation of a non-blocking read in Catapult HLS can be achieved by only performing the read operation when the port has available data:

```
if (PE_control_in.size()>0) {
    cmd_type = PE_control_in.read();
}
```

Random number interface

The random number interface is responsible to read sequentially random bits from the RNG hardware infrastructure as explained in Section 4.6. Although this interface is a `ac_channel` data type, it does not connect with a hardware FIFO, but instead to a register that is providing a different single random bit at each clock cycle. Thereby, the Catapult HLS configuration of this interface is different from the one used in the control interface where the PE interacts directly with FIFOs and control signals are added to the circuit. The configuration of the random number interface, although of data type `ac_channel`, does not have any control signals and a reading operation is performed unrestricted when requested.

The following example shows how to read two different bits from the random number interface port so that a 2-bit random number is built:

```
ac_int<2, false> rn_2bit;
rn_2bit[0] = global_RNG_1bit.read();
rn_2bit[1] = global_RNG_1bit.read();
```

With the configuration of this interface port, the `read` method performs an unconditional reading operation and thus it never stalls the hardware. Also, and for the example provided, the `ac_channel` data type of this port ensures that both reading operations are executed in two different clock cycles.

5.2.1.3 Access arbitration to the subpopulation memory

In Section 4.4.2 we have discussed how a subpopulation memory manages the accesses to its contents (the solutions of the genetic algorithm) by the two PEs connected to this memory. The PE created by Catapult HLS must obey to the protocol that we developed to prevent corruption of data when conflicting accesses occur. This section provides all the details of how to describe the PE using C++ so that a subpopulation memory is correctly accessed.

As briefly described in the last section, the C++ class `request_channel` is responsible to generate all the required interface signals between the PE and the collision detection

TABLE 5.1: List of commands used during the selection procedure of a solution in a PE to access a subpopulation memory. Commands are used with the methods of a `request_channel` data type variable.

Command	Description
<code>CMD_READ_SOL_1</code>	Request read access to a solution (first solution).
<code>CMD_READ_SOL_2</code>	Request read access to a solution (second solution).
<code>CMD_WRITE</code>	Request write access to a solution.
<code>CMD_START_SELECTION</code>	Informs that the PE is starting a selection procedure.
<code>CMD_RELEASE_SOL_1</code>	Informs that read access to a solution (first solution) has finished.
<code>CMD_RELEASE_SOL_2</code>	Informs that read access to a solution (second solution) has finished.
<code>CMD_RELEASE_WRITE</code>	Informs that write access to a solution has finished.
<code>CMD_RELEASE_ALL</code>	Informs that all read and write accesses to solutions have finished.

hardware in the subpopulation memory. Therefore, this class generates all the required signals that ensure the correct handshake between a PE and a subpopulation memory.

Table 5.1 shows the commands that can be sent (using a `request_channel` data type variable) from the PE to a subpopulation memory. These commands can be divided in two main groups: the first three commands request access to a solution in the subpopulation memory and thus an acknowledge signal is expected; the last five commands inform the subpopulation memory to take actions (an acknowledge signal is not necessary). The class provides two methods to handle these two groups of commands, which fundamentally are the same and only differ from each other in the parameters that receive and in the return value.

Figure 5.2 depicts a flow chart of the algorithm structure with the sequence of commands that a PE must execute to choose a given solution in a subpopulation memory. The selection of any solution always start with the command `CMD_START_SELECTION` which indicates to the subpopulation memory that the PE is about to start a selection procedure. At this point, the PE can read the fitness value of the solutions in the subpopulation memory and take a decision, using known algorithms like roulette-wheel or binary tournament [SGK05], from which solutions are selected from the memory. Then, if the PE requests a read access to a given solution it must issue the corresponding command (`CMD_READ_SOL_1` or `CMD_READ_SOL_2`); for a write access the command `CMD_WRITE` is used. These three commands allow to have simultaneous read access to two different solutions, and one write access in the same memory, which is usually the required to generate a new solution in a genetic algorithm (read two solutions to generate a new one). These last commands are acknowledged by the subpopulation memory hardware that grants or not the access to the solution so that no conflicts occur between

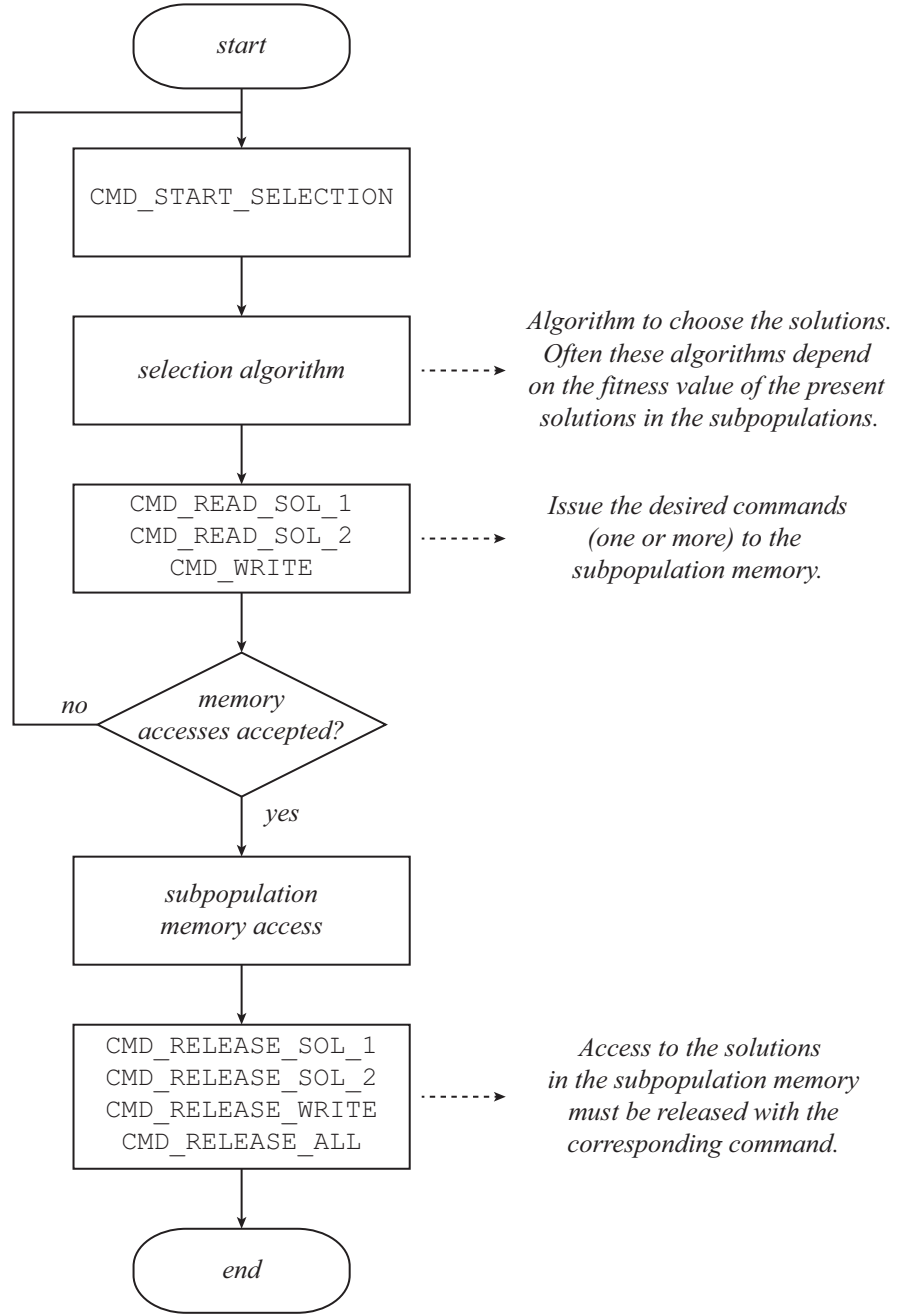


FIGURE 5.2: Algorithmic structure used by Catapult HLS to implement a selection procedure of solutions in a PE.

the two PEs while accessing to the same data (as specified in Table 4.1). When the access requested is not granted by the subpopulation memory, the PE must repeat the steps aforementioned to select again a solution (the same or other). After the solution has been accessed, the PE must issue a command to release the access block to that solution, so that the other PE can have a less restricted access to it.

All the commands that request and release a solution are straightforward to understand and use, and are enough to implement any correct access to the subpopulation memory.

Nevertheless, the command `CMD_START_SELECTION` plays an important role for a correct selection of a solution based on algorithms that depend on the fitness value of the solutions, which often happens. The problem is that when a request command is issued to the subpopulation memory, it requests an access to a particular solution which has been previously chosen based on its fitness value. However, from the instant when the fitness value is accessed till the actual request command is issued, that solution, which is not blocked by the PE, can be accessed and changed by the other PE. Therefore, the request will be based on a solution that does not exist anymore in memory, thus leading to an incorrect implementation of the selection procedure of the GA.

Figure 5.3(a) depicts an example of what could happen if we developed a solution in our hardware that did not use the command `CMD_START_SELECTION` when two PEs access to the same solution (number 4 for the example) in the same subpopulation memory. As it can be seen, during the selection algorithm in *PE#2*, *PE#1* acquires access to the solution and updates it. When *PE#2* requests the access this is granted as the solution is not being used anymore by the opposite PE. Nevertheless, the selection of the solution in *PE#2* is incorrect as it was based on a fitness value of a solution that is not the one that is in memory during the read access. Figure 5.3(b) shows the similar sequence of operations by the two PEs, but now using the command `CMD_START_SELECTION`. This command informs the subpopulation memory hardware that a selection algorithm is being processed in the PE and therefore solutions that meanwhile may be changed by the opposite PE cannot be selected by the algorithm.

It should be emphasized that the command `CMD_START_SELECTION` must always be issued before a read or write request command, even if the selection algorithm does not depend on the solutions' fitness values (e.g. selection of a random solution).

The following code example shows how a PE can grant a read access to a solution in a subpopulation memory, by performing several sequential requests to different solutions till the access is granted:

```
ac_int<2,false> n_sol = 3;
bool req_ack;
do{
    N_request.send_request(CMD_START_SELECTION);
    req_ack = N_request.send_request(CMD_READ_SOL_1, ++n_sol);
}while(!req_ack);
// safe read access to solution
// number 'n_sol' in memory 'N_mem'
N_request.send_request(CMD_RELEASE_SOL_1);
```

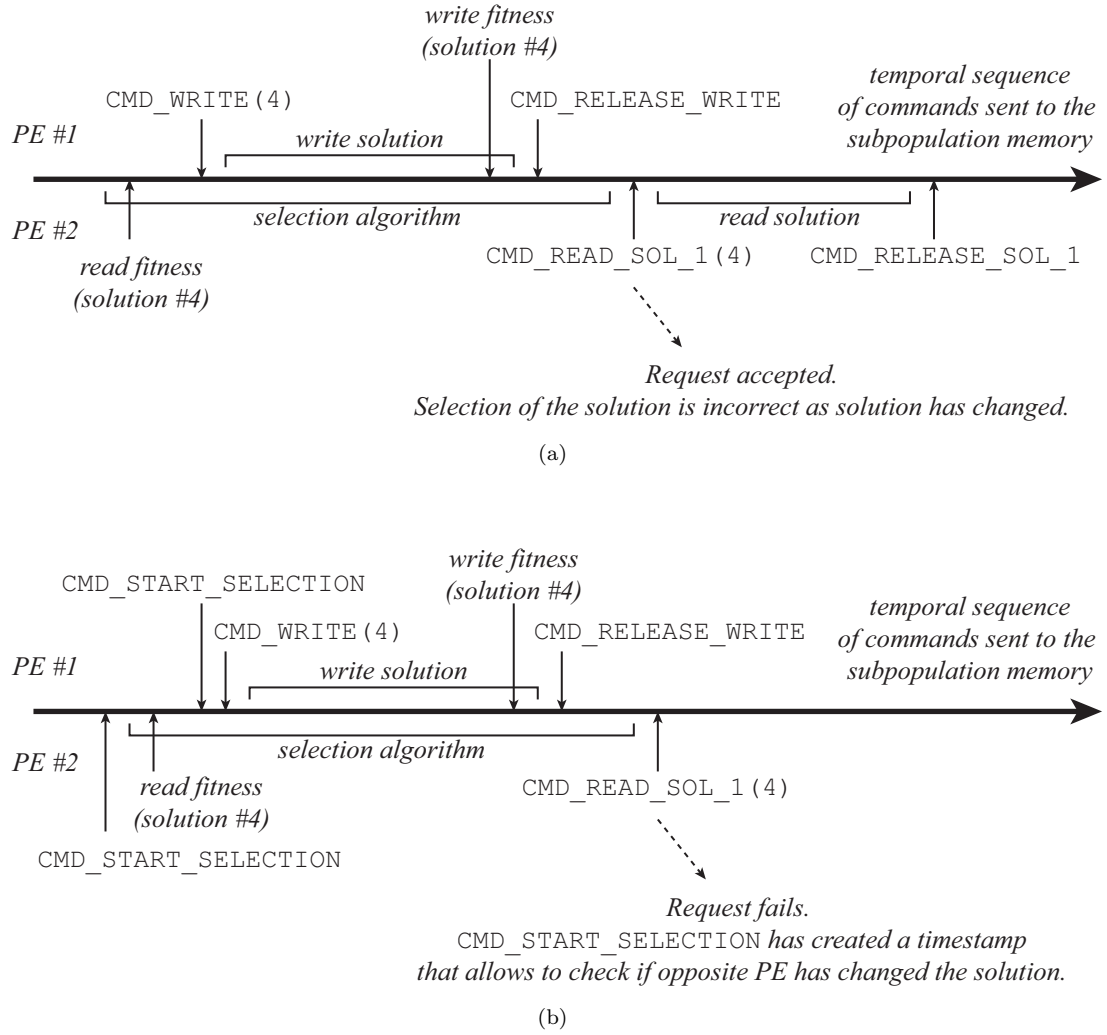


FIGURE 5.3: Sequence of commands issued by two PEs to a subpopulation memory to access the same solution: (a) not using `CMD_START_SELECTION` (not implemented); (b) using `CMD_START_SELECTION`.

As it can be seen, two methods `send_request` are used to send commands to the subpopulation memory. To request reading or writing access it is required to send the corresponding command (`CMD_READ_SOL_1`, `CMD_READ_SOL_2`, or `CMD_WRITE`) together with the solution identification that we want to access to. This method returns then a boolean value stating if the access was granted or not. All the other commands require only the command code as a parameter and no return value.

In the example above, we access to a solution in the north subpopulation memory, which is achieved by using the interface variable `N_request`. All the four subpopulation memories connected to the PE are independently managed by using the corresponding variables that must be conveniently manipulated by the designer while describing the PE.

5.2.2 cGA controller

The cGAC of the cGAP is responsible to control and monitor all the activities of the PEs. Using a dedicated communication infrastructure (see Section 4.5.2), this block communicates individually with each PE by sending and receiving commands. The cGAC also receives commands from the host computer to configure itself and the PEs with appropriate parameters.

5.2.2.1 Algorithm structure

Figure 5.4 illustrates the flow chart of the overall structure that specifies in Catapult HLS the cGAC. This algorithm implements an infinite loop, continuously checking for commands issued by the host or by any PE in the array. When data is detected in one of these ports, the cGAC reads and processes the command.

To allow a correct communication between the host and the cGAC, we have introduced an acknowledge mechanism that is issued by the cGAC when the previously sent command by the host has been processed. From the host side, a new command to the cGAC can only be sent after the acknowledge signal from the previous command has been issued. This technique simplifies the communication between the host and the cGAC, as it avoids a dedicated FIFO between these two elements capable of acquiring and controlling several consecutive commands sent by the host.

During the execution of the cGAP, it is expected that in a first phase the cGAC receives a set of parameters and data from the host to configure itself and the PEs. After that and during the evolution of the algorithm, each PE can send to the cGAC, for instance, the best fitness found as it evolves so that the cGAC knows which PE has the best solution when the process finishes. At the end, the host requests the best solution by sending the appropriate command to the cGAC which, in turn, requests the same to the PE where the best solution is. The desired information is then retrieved to the cGAC and then to the host. The communication with the host is performed via the cGA dual-port memory, where the cGAC loads all the necessary data.

The previous description of the cGAC behaviour is just an example of a possible approach to implement the cellular genetic algorithm. However, the communication among the host, the cGAC, and the PEs must be customized to each particular optimization problem (to be solved by a genetic algorithm or other metaheuristic). With the proposed functionality for the cGAC, the designer is able to build potentially any communication interaction among all the PEs and the host, as well as a centralized control of all the operations and evolution of the algorithm. Nevertheless, it must be stressed that the

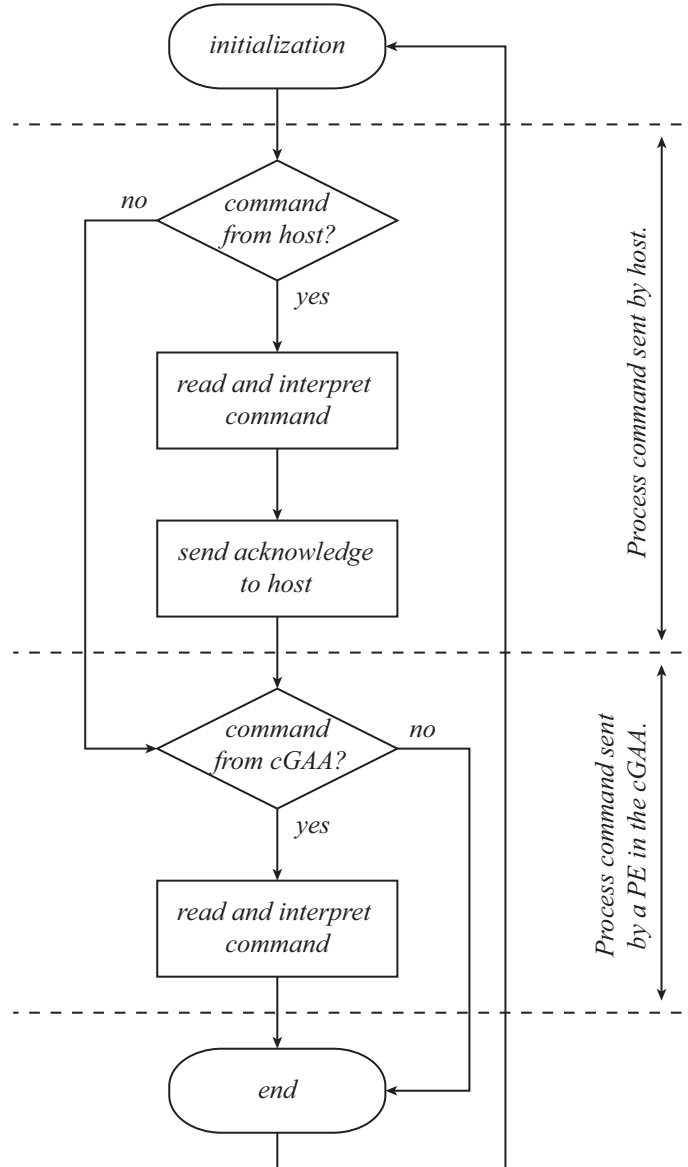


FIGURE 5.4: Algorithmic structure used by Catapult HLS to describe the cGAC.

communication among the cGAC and the PEs is carried out by an infrastructure that imposes throughput limitations, as discussed in Section 4.5.2.

5.2.2.2 Interface

The C++ interface used to describe the cGAC is shown in Listing 5.2. It contains three groups of interface ports that allow the communication with the principal blocks of the cGAP, mainly: the cGA dual-port memory; the host; and the cGAA where the PEs are placed. This section provides a brief description of how to use these groups of interfaces. Examples of how to write C++ code to be used by Catapult HLS tools can be found in

```

1 void cGA_controller(
2     // cGA dual-port memory interface (SW/HW shared memory)
3     ac_int<32,false> mem[1024],
4
5     // Host interface
6     ac_channel<command_type_cGA> &cmd_from_SW,
7     ac_channel<bool> &cmd_end_ack,
8
9     // cGAA interface (PEs)
10    ac_channel<command_type_cGA> &cmd_to_cGAA,
11    ac_channel<command_type_cGA> &cmd_from_cGAA
12 )

```

LISTING 5.2: C++ description of the cGAC interface to be used by Catapult HLS.

Section 5.2.1.2 as the techniques and the C++ classes used are the same for both the PE and the cGAC.

cGA dual-port memory interface

This memory can be accessed (reading or writing) by the cGAC to keep any data necessary to its execution, and to transfer data to the host computer. When describing the cGAC in Catapult HLS, this memory is accessed as a normal C++ array variable. In Listing 5.2 we can see an example of how Catapult HLS performs an interface to a memory with 1024 words of an unsigned integer with 32 bits data type (`ac_int<32,false>`).

Host interface

This interface guarantees the correct management of commands sent by the host to the cGAC. It comprises two ports: one for reading a command and the other that acknowledges when this command has been processed, as specified in the algorithm structure of the cGAC (cf. Figure 5.4). The first port reads a command from the host and therefore is a `command_type_cGA` data type variable.

cGAA interface

The interface to the cGAA (or the PEs) is performed by two distinct ports: one labelled `cmd_to_cGAA` to send a command to a PE (or all the PEs), and another labelled `cmd_from_cGAA` to receive a command from a specific PE. Both ports are a `ac_channel` class data type from Catapult HLS that uses the methods `write` and `read` to perform writing and reading operations according to the port. As expected, these ports carry a data type `command_type_cGA` that can be manipulated using the methods described in Appendix A.1.

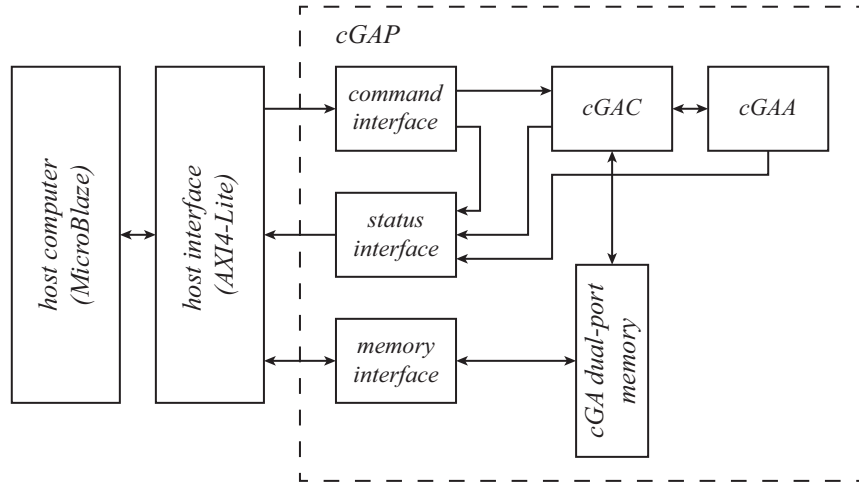


FIGURE 5.5: Overview of the host communication with the cGAP. A MicroBlaze soft-core processor accesses to the cGAP via AXI4-Lite protocol.

5.3 cGAP host communication

This section presents the host processor communication with the cGAP. Since the target FPGA is a Xilinx Virtex-6, a MicroBlaze soft-core processor, whose implementation is supported by Xilinx tools, is integrated in the same FPGA to access the cGAP [Xilb]. Additionally, we provide C functions examples of how this processor can access correctly to the cGAP as a memory mapped device.

5.3.1 The host interface

Figure 5.5 shows an overview of the communication between a host processor and the cGAP, which is performed by ensuring that both the host interface and the cGAP interface blocks communicate properly.

As our architecture will be implemented on a Xilinx Virtex-6 FPGA, which presently uses ARM AMBA4 protocol for connecting and managing different blocks in a system on chip (SoC), we have decided to use AXI4-Lite protocol for the host interface, which is a subset of AMBA4 that allows simple and small control register-style interfaces [ARM].

The interface blocks of the cGAP communicate properly with the AXI4-Lite protocol so that the MicroBlaze and the cGAP can communicate with each other. It should be emphasized that the functionality of the cGAP interface blocks belongs to the specification of the complete architecture, as it has been presented in Section 4.5.3. Nevertheless, these blocks may require adaptations so that they can communicate with different host interfaces.

In Appendix B we provide further details on how the MicroBlaze accesses to the cGAP, which is achieved by defining a set of registers in the interface blocks that can be accessed by a software program running in the processor.

5.3.2 Software access to cGAP

In this section we provide examples of C functions to be used by the operating system (OS) running in the MicroBlaze to interact with the cGAP.

Since the cGAP will be implemented in a Xilinx FPGA, together with a MicroBlaze soft-core processor, we have decided to run a Linux OS in the MicroBlaze capable of accessing the cGAP as a memory mapped device. This process involves two main tasks: a project must be built using the Xilinx tools with the appropriate configurations for the MicroBlaze to run Linux and all the required hardware peripherals (including the cGAP); a Linux kernel configuration and compilation with the device tree file previously generated by the Xilinx tools. These two steps create an FPGA bitstream and a kernel image file that are used to program the FPGA and to run the Linux in the MicroBlaze.

In Appendix B.1 it can be found the C source code of a set of basic functions that are used as an application programming interface (API) to build software applications that can run in the MicroBlaze to access the interface circuits of the cGAP. This API implements the mapping of the cGAP into memory, and the access to a set of registers that allow the access to the interface circuits of the cGAP.

By using the API, we can build new C functions that can properly send commands to the cGAC, and to monitor the status of the cGAP. The following code example shows an excerpt of C code running in the MicroBlaze to execute an optimization process in the cGAP.

```
// map cGAP hardware into memory
IP_cga_init();

// send commands to cGAP (set-up phase)
cga_reset();           // reset cGAP
cga_set_generations(1000); // define number of generations
...                   // other commands

// run the cGA
cga_start_all_PEs();   // command to start GA in PEs
```

```

// wait till cGAA (the PEs) stop
cga_PEs_stopped();           // check status interface circuit

// results from cGAP (best solution)
// data available in the cGA dual-port memory
best_fitness = read_BRAM(0);    // read fitness value
for (i=0; i<10; i++)
    best_solution[i] = read_BRAM(i+1); // read solution

// delete cGAP hardware memory mapping
IP_cga_clean();

```

In this example, several functions are used to send commands (e.g. `cga_reset` or `cga_start_all_PEs`) or to check the status of the cGAP (`cga_PEs_stopped`). Appendix B.2 presents the implementation of several examples of C functions that use the cGAP API so that the hardware is correctly accessed and controlled.

Different optimizations problems may require different commands to be sent to the cGAP. Therefore, it is expected that the software interface must be customized to each optimization problem. Moreover, the execution of the cGAP can be controlled by the software as desired. In the previous example, the best solution is only read after the PEs stop their execution which is defined by a parameter in the set-up phase. However, different methods can be adopted, like sending a specific command to stop the PEs at any given time, or check the evolution of the best fitness till it reaches a certain convergence criterion.

All the C functions presented in this section target a specific hardware platform where the cGAP will be integrated. Therefore, they must be viewed as a particular example of an application where a MicroBlaze running a Linux OS accesses the cGAP as a memory mapped peripheral. Other application scenarios, with the cGAP attached to different processing systems, may require other interface mechanisms.

5.4 Design Flow

The design flow proposed in this work aims to organize the development process of the cGAP, so that it can implement a cGA or other population-based metaheuristic for different optimization problems. Therefore, the main core of the flow is the specification of the PE and the cGAC hardware blocks using high-level synthesis methodologies to allow the specification of the problem-dependent blocks in a conventional programming

language instead of hardware description languages (HDL) at the register-transfer level (RTL). The rest of the architecture, which is not specific of the operations of neither the optimization algorithm nor the optimization problem, is described in Verilog HDL. The cGAP configuration, like the dimension of the cGAA, or the number of solutions and organization of the subpopulation memories, is defined by a set of parameters common to the different stages of the design flow.

Figure 5.6 shows an overview of the complete design flow used in this work which targets the implementation of a complete embedded system constituted by a MicroBlaze soft-core processor that uses the cGAP as a peripheral. Additionally, this flow includes the development of a software executable that can interact with the cGAP. The target hardware is a Virtex-6 FPGA (XC6VLX240T-1) embedded in a ML605 evaluation kit from Xilinx [Xilf].

We have used for the HLS tools Catapult HLS version 2010a (University Version) from Calypto Design Systems [Cal], combined with RTL synthesis by Precision RTL version 2010a from Mentor Graphics [Gra]. As the target hardware is an FPGA from Xilinx, we have used ISE Design Suite embedded edition (mainly ISE and EDK) version 13.4 from this company for generating the FPGA bitstream [Xila].

5.4.1 cGAP parameters configuration

A small set of parameters must be defined to configure the cGAP architecture as desired. This allows defining the number and aspect ratio of the PE array (cGAA) with their toroidal or non-toroidal shape, the configuration of the subpopulation memories, the maximum number of solutions that a subpopulation memory can handle, and the definition of all the control commands required by the cGAP.

In the following list all the parameters used in the design flow are described:

- **NONTOROIDAL**: Defines a toroidal or non-toroidal configuration (see Figure 4.2) for the cGAA.
- **ARRAY_DIM_I** and **ARRAY_DIM_J**: Both define the configuration of the PEs in the 2D regular structure of the cGAA. The parameters define, respectively, the number of PEs in the row and column of the array.
- **WIDTH_MEM_SUBPOP** and **DEPTH_MEM_SUBPOP**: These two parameters define the organization of the subpopulation memory (a dual-port RAM). The first parameter defines the width of the memory (the total number of bits), and the second parameter the \log_2 of the depth.

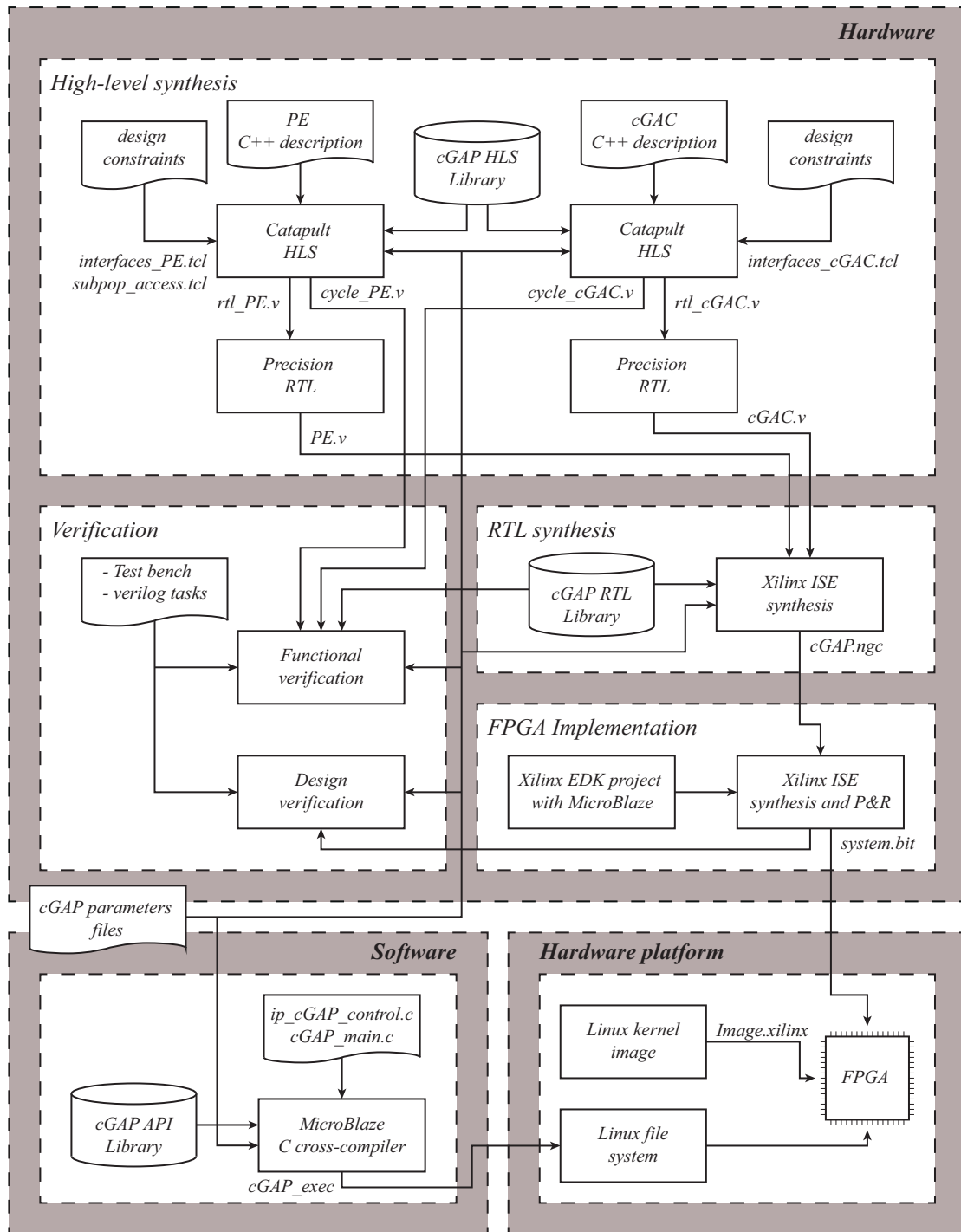


FIGURE 5.6: Overview of the hardware design flow used to build the complete cGAP, that connects to a MicroBlaze soft-core processor running a Linux OS, both embedded in a Xilinx Virtex-6 FPGA.

- `N_BIT_COMMAND_GA_PROC`: Defines the number of bits for encoding the command codes in the cGAP.
- `N_BIT_MAX_SOL_PER_SUBPOP`: Defines the number of bits for encoding the solutions in each subpopulation memory of the cGAA. This parameter ensures that

an access request by two PEs to the same solution are correctly handled by the subpopulation memory circuit (see Figure 4.5) as described in Sections 4.4.2 and 5.2.1.3. The way the solutions are organized in the memory is not defined by this parameter and it is the responsibility of the designer of the PE to define it.

- *Commands list*: A list of all command codes and names must be defined as required by the cGAP project. As an example, we can define a parameter named `CMD_START_PE` with code 0 that means a command to start a PE. The definition of the parameters names and meanings must be defined by the designer and adapted to each project. Nevertheless, a command named `CMD_RESET_CGAP` representing a reset to the cGAP hardware must exist in this list since it is hard-coded in the RTL description of the hardware.

5.4.2 Hardware

The hardware phase of the design flow targets the generation of a bitstream file to program the FPGA with the cGAP connected to a MicroBlaze processor. In the following subsections, we will explain the different phases of this part of the design flow.

5.4.2.1 High-level synthesis

The high-level synthesis phase allows easing the design of the PE and cGAC modules for a specific optimization problem. For that, two independent projects must be created using Catapult HLS, one for the PE and another for the cGAC, as depicted in Figure 5.6.

Both projects share the *cGAP HLS library* which includes the data types developed for manipulating the commands sent among the cGAC and the PEs (`command_type_cGA`), as well as for accessing to the subpopulation memories (`request_channel`). For reference, a description of all the files included in this library is presented in Appendix C.1.

Therefore, the designer must describe in C++ the algorithms for the PE and the cGAC with the intended operations for the genetic algorithm (or other) to target a given optimization problem. Although the algorithms are described in a current programming language which is inherently sequentially, a correct coding style must be followed to achieve good results, as for example, to produce the desired pipelining and parallel hardware structures. Additionally, the subpopulation memories must be organized to accommodate properly the solutions, or any other additional data required by the PEs. In Chapter 6 we provide several examples that exploit the organization of the data in the subpopulation memories so that efficient hardware structures are built. Both algorithms

must be described as C++ algorithms according to the templates explained in detail in Sections 5.2.1 and 5.2.2 for the PE and cGAC modules respectively.

In addition, each of the two Catapult HLS projects must be constrained so that all the ports are properly configured. For that, the PE and cGAC projects require the specification of design constraints for configuring all existing interfaces of the hardware modules, and for ensuring the correct synchronization used by the different ports for the subpopulation memory handshake protocol. Additional constraints to the projects (e.g. clock frequency, optimization efforts, loop pipelining, etc.) are also specific from each project and must be used as required.

5.4.2.2 RTL synthesis

The RTL synthesis phase of the design flow targets the synthesis of the complete cGAP architecture. Therefore, it requires the hardware description of the previously generated files by the HLS phase to describe the PE and cGAC. Additionally, it requires a *cGAP RTL library* with all the files describing the remaining cGAP. This library, which is presented in Appendix C.2 for reference, is entirely described in Verilog HDL and it includes all the hardware blocks that do not need to be adapted or changed so that the cGAP targets different optimization problems.

In this phase is used the XST tool from Xilinx to integrate the previous blocks created by HLS with the rest of the system to build the complete cGAP. However, we should emphasize that only a synthesis is implemented in this phase that generates a netlist file to describe the cGAP, and not an implementation of the circuit in the FPGA.

5.4.2.3 FPGA implementation

The FPGA implementation phase generates the bitstream file that will be used to program the target FPGA. For that, a project with an embedded design has been created with EDK for the Xilinx ML605 board. This design includes a MicroBlaze soft-core processor capable of running a Linux OS, and it contains the cGAP connected to the processor via AXI4-Lite interface as a memory mapped device. The ISE design flow is used to implement the complete project provided by EDK, including the netlist describing the cGAP block.

5.4.2.4 Verification

The verification phase of the design flow targets mainly to check if the cGAP is behaving as expected. Therefore, we perform a functional verification by using the functional models generated during the HLS phase for simulating the PE and cGAC hardware blocks, together with the cGAP RTL library files. A set of Verilog tasks have been developed to emulate the behaviour of the AXI4-Lite interface, thus emulating commands sent by the host computer to the cGAP. With this technique, we provide identical stimulus to the cGAP as the MicroBlaze does in a real hardware implementation.

After functional validation of the whole cGAP, the final verification of the complete embedded system was performed in real hardware while running various optimizations problems. A design verification of the complete system can also be accomplished by simulation, requiring all the simulation models of the system, including all the external devices of the ML605 board that the FPGA connects to.

5.4.3 Software

In the design flow, the software phase involves the creation of an executable program to be run in the MicroBlaze to control the execution of the cGAP. For that, the *cGAP API library* includes the basic functions that must be used to access the cGAP peripheral as a memory mapped device of the processor, thus allowing the mapping of the hardware to memory and to access the interface circuits of the cGAP. Appendix C.3 provides an overview of the files included in this library.

For each new cGAP implementation, it may be required to develop new software functions using the routines defined in the API, so that problem-specific functionalities (e.g. commands) are implemented. In Section 5.3.2, we have already presented examples of such functions. Additionally, Appendix B.2 provides detailed examples of how to implement them.

5.4.4 Hardware platform

As we have already mentioned, the design flow targets the implementation of the cGAP in a ML605 evaluation kit, with a Virtex-6 FPGA [Xilf]. A MicroBlaze soft-core processor is also implemented in the FPGA, to which the cGAP is connected via AXI4-Lite interface. Therefore, this processor controls the execution of the cGAP by using a software program that can be used as desired to evaluate the algorithm.



FIGURE 5.7: Set-up of the test bench platform used to evaluate the cGAP. A Virtex-6 FPGA, placed in a ML605 board, integrates a MicroBlaze soft-core processor connected to the cGAP. The processor runs a Linux OS with its file system hosted in a PC.

To facilitate the development of the software and have access to a file system, we use a Linux operating system to be run on the MicroBlaze. For that, we have utilized the Xilinx Open Source Linux [Xilh], which is an open source project, that provides a Linux OS and additional tools required for software development². This project can be obtained from a Git repository provided by Xilinx [Xilg]. A dedicated Linux kernel has been properly configured and compiled using a device tree file previously generated by the Xilinx tools during the FPGA implementation phase of the design flow, thus including the cGAP as a peripheral of the processor. Additionally, we have configured the kernel to boot over the network, using Network File System (NFS). This allows the file system of the Linux OS running in the MicroBlaze to be physically in a hard disc drive (or any other data storage element) of a PC. In a typical session to evaluate a cGAP, the MicroBlaze is accessed from the PC using a remote shell where the software that controls the cGAP is run.

²The Xilinx Open Source Linux has been chosen since it is an open source project. However, more recently, PetaLinux is also available under no-charge license and it is, at the present date, the Linux distribution more recommended and fully supported by Xilinx [Xilc].

Figure 5.7 shows a picture of the final hardware platform used in this work. As it can be seen, the ML605 board connects to a PC via an ethernet cable. In the PC, it is hosted the file system needed to boot the Linux OS of the MicroBlaze, together with the development tools (a MicroBlaze *gcc* cross-compiler) used to create a software application to communicate with the cGAP.

Since the Linux of the MicroBlaze has its file system in the PC, it is straightforward to develop a new software application and immediately run it on the MicroBlaze. Moreover, any additional files required to execute the cGA can be easily added to this file system to send or to collect data from the algorithm. This approach is ideal to evaluate the cGAP under different conditions since the PC can handle all of these files without the need to explicitly send or receive them from the ML605 board.

Although in this phase we can create any new software application to run in the MicroBlaze that accesses to the cGAP, if the cGAP needs to be changed the design flow must be started from the beginning with the HLS phase to rebuild the peripheral. However, it is not required to modify the Xilinx EDK project with the soft-core processor, nor to compile a new Linux kernel as the MicroBlaze configuration is not changed.

5.5 Summary

In this chapter we have introduced the description of the PE and cGAC hardware modules using HLS tools, so that the cGAP is customized to solve different optimization problems, or even using other population-based metaheuristics. We have defined three algorithm templates that must be followed to build these hardware blocks. The first algorithm template ensures that a PE acquires data from the cGAC so that it can be configured as desired, and it executes iteratively the operations of the metaheuristic. The second algorithm details the steps that a PE must follow to access safely to the solutions kept in a subpopulation memory. Finally, the third algorithm defines how the cGAC must handle both commands received from the host and from the PEs.

We also have provided examples of how the interface of the cGAC and PE blocks can be accessed with the HLS tool used. To do so, we have presented C++ data types specific from the HLS tool used, and defined new ones to manipulate specific data used to describe the algorithms.

The host communication used by the cGAP has been described so that the hardware can be accessed. We have explained how the cGAP can receive data or commands from the host, while providing C code examples that can be run on a Linux OS of a MicroBlaze soft-core processor to access the cGAP as a memory mapped device.

Finally, we have proposed a design flow to build the cGAP, including an experimental hardware platform where the cGAP can be evaluated, consisting in a ML605 board with a Virtex-6 FPGA from Xilinx that implements the cGAP together with a MicroBlaze processor running a Linux OS. This board accesses to a PC where the file system of the OS is placed.

In the next chapter, we will present two different implementations of the cGAP using the proposed design flow, and show results that demonstrate the speedup potential of the proposed architecture for cellular genetic algorithms.

Chapter 6

Experimental results

6.1 Introduction

In the previous chapters the architecture of the cellular genetic algorithm processor (cGAP) was presented, as a scalable and dedicated processor array to solve cellular genetic algorithms (cGAs). Moreover, a high-level synthesis (HLS) based design flow was defined that allows to specify in C++ the optimization problem-dependent blocks: the processing element (PE) where the genetic algorithm is run, and the cellular genetic algorithm controller (cGAC) where the operations of the array are coordinated. Therefore, by customizing these blocks we can build the cGAP to solve different optimization problems that may require specific operators to be used by the algorithm.

In this chapter, two distinct optimization problems will be addressed as case studies to demonstrate the effectiveness of the proposed computing array and design methodology: the spectrum allocation (SA) problem presented in Section 6.2, and the minimum energy broadcast (MEB) problem presented in Section 6.3. Both problems appear in the context of wireless ad hoc networks, where the first tries to improve the usage of the available spectrum by exploiting opportunistically unused spectrum, and the second consists in broadcasting a message with the minimum energy consumption. These problems pose several challenges when solved with GAs, like problem-specific constraints, special solution representations, or dedicated local search algorithms for better performance of the GA. The specification of the PE with a HLS flow is thus helpful for easing the design of these functionalities.

During this chapter, project details of how to build and optimize the problem-specific blocks with the HLS tools will be presented. Implementation results and acceleration figures are provided for the engines. General considerations on the cGAP implementation

are provided in Section 6.4. The cGAPs for the SA and MEB problems have been successfully run in a Virtex-6 FPGA.

6.2 Spectrum allocation in cognitive radios

With the increase of the wireless devices available in our world, it becomes more important an efficient usage of the spectrum available. This is usually poorly exploited by static spectrum assignments [TZFS13]. In this regard, cognitive radio has emerged as a promising technology that exploits unused spectrum in an opportunistic manner. Mobile wireless devices like smartphones, laptops, etc., can access to a portion of a free spectrum available (via a software defined radio) and not to a specific spectrum band like it happens in current radios like 4G and IEEE 802.11.

A cognitive radio network typically consists of a set of primary users with assigned channels, and a set of secondary users that are allowed to access the available spectrum in a coordinated way. It is expected that the secondary users, based on constraints imposed by the primary users, use unused licensed spectrum on a non-interfering basis. This way, the available spectrum can be reused resulting in an effective increase of the system usable bandwidth.

The spectrum allocation problem is thus a good example where the proposed architecture can be applied, since there is the need to continuously solve this problem in a real environment where mobile devices normally change their position. Moreover, the problem may impose time restrictions to be solved, and solving it as fast as possible means an increased efficiency of the spectrum used.

In this section, we present an implementation of a cGAP for solving a spectrum allocation problem. This problem, unlike the TSP used in Chapter 3, imposes several constraints to the solutions generated by a GA and thus additional operations must be used to correct them. Using the design flow presented in the last chapter, we develop the necessary hardware blocks customized to this particular problem.

6.2.1 Problem definition

The SA model adopted in this work is described in [PZZ06]. It consists of a set of N secondary users that try to access to M non-overlapping orthogonal channels. Each secondary user can utilize any channel, but limited to interference constraints that may appear among secondary and primary users.

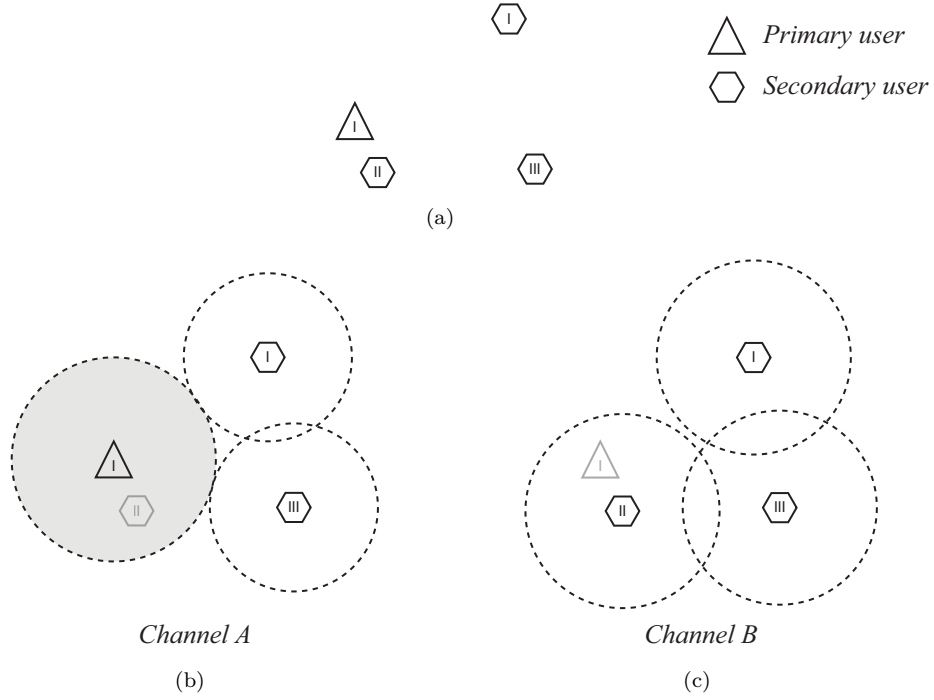


FIGURE 6.1: Example of a SA problem with (a) placement of 1 primary and 3 secondary users, (b) availability of channel A among secondary users and primary user, (c) availability of channel B among secondary users

Figure 6.1(a) illustrates an example of a placement of a primary user (a legacy spectrum device) and three secondary users (unlicensed devices). The primary user utilizes always channel A, whereas the secondary users can utilize any channel as long as no interference appears among all users. Figure 6.1(b) shows what happens when the secondary users utilize the same radio channel as the primary user (channel A). In our model, the secondary users are able to adjust their maximum transmission power so that they do not interfere with the primary user. Nevertheless, secondary users can interfere among them, as it is depicted in the figure, which leads to an infeasible solution of the SA problem. Since secondary user II is placed inside the range of the primary user, it cannot use the same channel. Figure 6.1(c) depicts the usage of a different radio channel (channel B) from the primary user in all the secondary users. In this case, and since there is not any interference with other primary user, the secondary users transmit at their maximum transmission power which may lead to undesired interferences. In the example, secondary user III interferes with secondary user II and I and therefore they cannot use the same radio channel at the same time.

A channel availability matrix $L = \{l_{n,m} | l_{n,m} \in \{0, 1\}\}_{N \times M}$, where N and M are respectively the number of secondary users and number of channels, defines when the channel m is available to user n by setting $l_{n,m} = 1$. This constraint ensures that a given secondary user does not interfere with a primary user when it is in its range of transmission.

A matrix $C = \{c_{n,k,m} | c_{n,k,m} \in \{0, 1\}\}_{N \times N \times M}$ informs when the user n and k cannot use simultaneously channel m by setting $c_{n,k,m} = 1$, which happens when the two users are in the vicinity of each other and thus interfere if using the same channel.

The spectrum allocation problem consists on finding the best channel assignment matrix $A = \{a_{n,m} | a_{n,m} \in \{0, 1\}\}_{N \times M}$, where $a_{n,m}$ means that channel m is assigned to user n . The constraints of the problem are defined by the following equations:

$$a_{n,m} \leq l_{n,m}, \quad \forall n < N, m < M \quad (6.1)$$

$$a_{n,m} + a_{k,m} \leq 1, \quad \text{if } c_{n,k,m} = 1, \forall n, k < N, m < M \quad (6.2)$$

A matrix $B = \{b_{n,m}\}_{N \times M}$ represents the channel rewards that can be acquired by user n when using channel m . These values are related with the maximum distance that a user can transmit when using a given channel.

The objective of this problem is to maximize a given utility function U . In this work, we consider the maximum sum reward defined by Equation 6.3, which maximizes the total spectrum utilization of the system, although others functions can be considered [PZZ06].

$$U_{sum} = \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} a_{n,m} \cdot b_{n,m} \quad (6.3)$$

It has been demonstrated that the previous formulation of the SA leads to a NP-hard problem [PZZ06], and therefore evolutionary algorithms are a good approach for solving them [ZPZS09].

During this section several instances of the SA problem are used to evaluate the cGAP, that have been generated by us according to the algorithm for modelling a network conflict graph provided by [PZZ06]. We could not use known instances for this SA problem because there is no knowledge of a database with such problem. However, with the aforementioned algorithm, we have distributed randomly a set of primary users and secondary users over a 10×10 square (of an abstract distance unit) and generated the required matrices for the SA formulation. As used by default in the instances of [PZZ06], the number of primary users has been fixed to 20, the maximum and minimum transmission distance of a secondary user is 4 and 1 respectively, and the transmission distance for the primary users is 2. The generated instances are labelled with the name N_M where N is the number of secondary users and M the number of available channels.

TABLE 6.1: Genetic operations adopted in the PEs for the spectrum allocation problem.

Parent selection	binary tournament
Crossover	uniform
Mutation	bit-flip (probability < 5 %)
Replacement	select random solution and replace if better

6.2.2 The cGA operations and control

In this subsection we present the genetic algorithm operations that each PE is going to compute, and the list of all the commands that the cGAP recognizes, that is, the commands used by the PEs and the cGAC.

Table 6.1 provides the list of the genetic operations used by each PE. Since a solution of the SA problem is defined by the binary matrix A , a solution is encoded with a binary representation that codes the complete matrix. Therefore, we have used the uniform and bit-flip operators that can be applied to binary representations of solutions. For the selection we have elected two binary tournaments to elect two solutions, and in the replacement a solution is randomly selected and replaced if the generated solution has a better fitness value. These operations are often used in cGAs [AD08].

To control the algorithm and to define new instances to be solved by the cGAP, we have defined a list of commands, as it can be seen in Table 6.2. As we have already explained, the cGAC can send a command to a specific PE or it can broadcast it to all the PEs. Therefore and as required, problem parameters that are common to all PEs, are broadcasted to configure the PEs simultaneously, whereas PE specific commands are sent only to one PE.

The commands implemented in the cGAC and in the PEs must be defined so that these blocks can communicate with each other while ensuring that all the information is correctly passed. For instance, the command to define an initial subpopulation in a PE requires several consecutive commands to send all the information (a subpopulation requires much more than 32 bits, which is the maximum for a single command). Therefore, when the cGAC sends this information, the PE is prepared to receive it in the same order as the cGAC sends it. Another example is the command to retrieve the best solution (CMD_GET_BEST_SOL), where the cGAC asks to a specific PE its best solution and waits till the requested information is sent back.

During the evolution of the algorithm, each PE notifies the cGAC when it finds a best solution, sending the newest fitness found. This way, the cGAC knows the best fitness value among all the PEs, which is used to track which PE has the best solution at the

TABLE 6.2: List of commands implemented in the PE and cGAC for solving the SA problem.

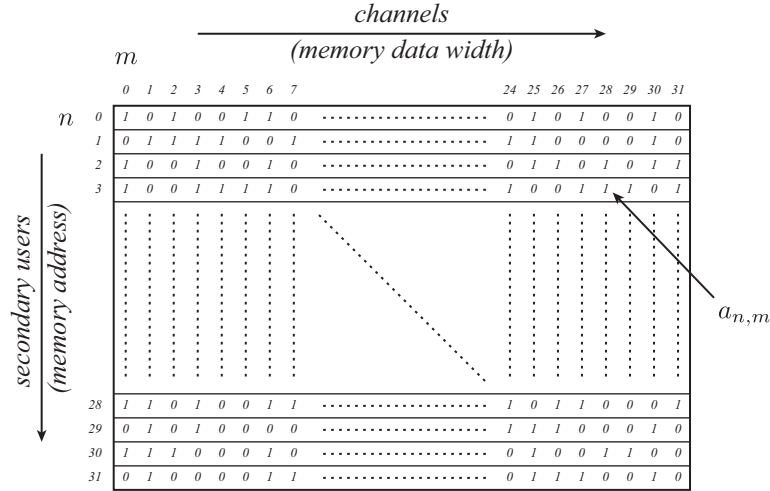
Command		Description
Name	Code	
CMD_START_PE	0	Starts execution of GA in a PE.
CMD_STOP_PE	1	Stops execution of GA in a PE.
CMD_N_SU	2	Defines number of secondary users (N).
CMD_N_CH	3	Defines number of channels (M).
CMD_MATRIX_L	4	Defines matrix L .
CMD_MATRIX_B	5	Defines matrix B .
CMD_MATRIX_C	6	Defines matrix C .
CMD_WRITE_SUBPOP	7	Writes subpopulation in a subpopulation memory.
CMD_RESET_FITNESS	8	Resets fitness value in all subpopulations.
CMD_N_SOL_SUBPOP	9	Defines number of solutions per subpopulation memory.
CMD_GET_BEST_SOL	10	Retrieves best solution in the cGAP.
CMD_SET_MAX_ITER	11	Defines maximum number of generated solutions in cGAA (only cGAC).
CMD_SET_FIT_STOP	12	Defines fitness value to stop cGAA (only cGAC).
CMD_RESET_CGAP	15	Command to reset the cGAP (defined in the HDL of the cGAP).

end of the algorithm. Additionally, this can also be used by the cGAC to stop the cGAA if the criterion is to evolve the algorithm till a certain fitness value is achieved.

Each PE also informs the cGAC when it generates a certain number of solutions, so that the cGAC can monitor the total number of solutions generated. Therefore, the cGAC can stop the cGAA till a given number of solutions are generated in all the PEs. This approach is followed since the PEs are not synchronized and thus different PEs can generate a different number of solutions for the same processing time. Although this does not guarantee an exact number of generations, it ensures that a requested number of solutions is reached.

6.2.3 The processing element

In this section we present a PE custom design for the SA problem. By distributing appropriately all the necessary data required by this problem (e.g. matrices L , C , and B) among the four subpopulation memories, we take advantage of this large memory bandwidth so that efficient pipeline and parallel hardware structures are built with HLS. We provide several examples of how a PE can be described in C++ for synthesis with Catapult HLS and how design constraints must be used to achieve good speed/area trade-offs.

FIGURE 6.2: Memory codification of a solution in the SA problem (matrix A).

6.2.3.1 Subpopulation memory organization

For the spectrum allocation problem presented the binary matrix A defines a solution of the problem. Figure 6.2 depicts the memory data organization for this matrix, for a maximum of 32 secondary users and channels. This way, a memory word keeps all the channels for a given user n in the matrix A . The memory addresses are thus assigned to secondary users (index n of the element $a_{n,m}$ in matrix A) and the data word to channels (index m of the element $a_{n,m}$ in matrix A).

Although the major role of the subpopulation memories is to hold the solutions evolved by the algorithm, they can also be used to keep additional data. This avoids the need to use extra block RAMs (BRAMs) of the FPGA, which may limit the maximum number of PEs (and subpopulation memories) in the cGAA. Additionally, by choosing a careful positioning of all the data in the 4 subpopulation memories, it is possible to exploit the memory bandwidth provided by these memories to build efficient hardware structures. The distribution of the data in matrices A , L , B and C of the SA problem among the subpopulation memories, has been chosen to explore this idea.

The 3 dimensional binary matrix C and the 2 dimensional matrix B are the ones that require more space. In total, and with a limit of 32 both for N and M (number of secondary users and channels), matrix C occupies 1024×32 bits and matrix B a total of 256×32 bits considering that each element $b_{n,m}$ (the cost of user n using channel m) occupies 8 bits. Therefore, we have divided the information of these matrices among the 4 subpopulation memories connected to each PE. Additionally and by realizing that $c_{n,k,m} = c_{k,n,m}$ and $c_{n,n,m} = 0$, only less than half of the elements of C need to be kept.

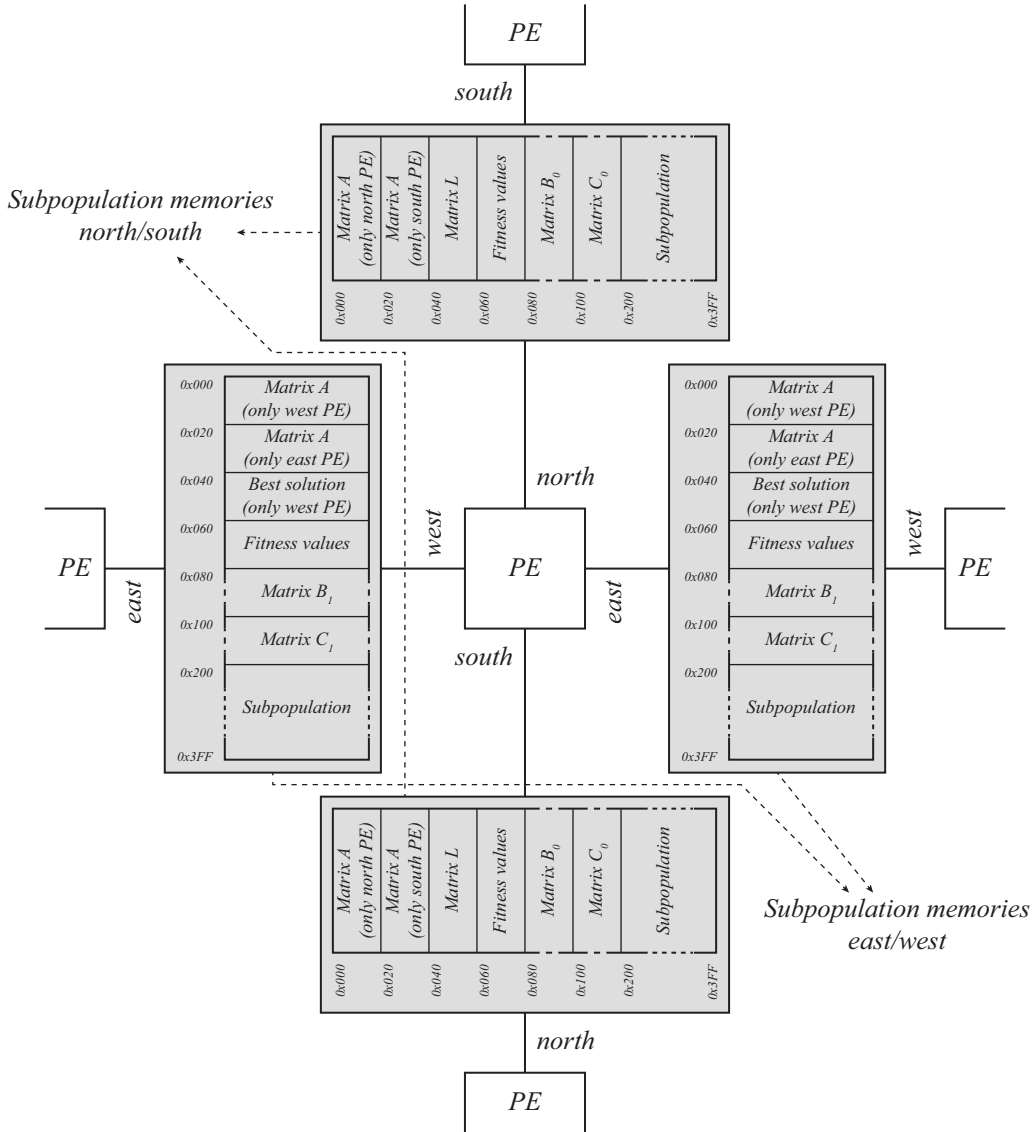


FIGURE 6.3: Subpopulation memory organization in the cGAP used for the SA problem.

Figure 6.3 depicts all the data organization in the 4 subpopulation memories: north, south, east and west. Both matrices B and C have their data divided between two slots: B_0 , B_1 and C_0 , C_1 . The first part of these matrices (B_0 and C_0) is replicated in the north and south memories, while the second part (B_1 and C_1) in the east and west memories. With this configuration every PE accesses to the same memory contents regardless of their position in the cGAA. This happens as the north memory of a given PE is the south memory of the PE above it. The same reasoning applies to the east and west memories. Although a division of the data among the 4 memories could be implemented, it would require different memory accesses for different PEs, depending on their position in the cGAA. Therefore, this approach would require more than one PE block to be built with HLS.

Matrix L is positioned and replicated in subpopulation memories north and south. As matrix A (the solution built at each generation) requires read and write accesses, each PE has its own memory space reserved to these data. Therefore, two different slots are allocated in each subpopulation memory, so that the PEs do not interfere with each other (see Figure 6.3). Additionally, the possible use of the matrix A in any one of the subpopulation memories, targets to increase the efficiency of pipeline structures for the different operations in the PE, since these data can be accessed from any of the memories.

With the proposed data organization in the subpopulation memories, it is possible to use a single BRAM of the Virtex-6 FPGA for each subpopulation memory to keep a maximum of 16 solutions per subpopulation and the auxiliary matrices required by the SA problem, as it is depicted in Figure 6.3. More solutions per subpopulation can be added, but this requires more than 1 BRAM per subpopulation memory. The fitness values of the corresponding solutions are kept in a different memory slot.

6.2.3.2 Coding in C++ for HLS

The subpopulation memory data organization described in the previous subsection has been exploited to improve the memory accesses as a PE can access to the same data from different memories. Table 6.3 shows the memory accesses used by the main operations computed by a PE to generate a solution.

For instance, the crossover operation requires a read of two solutions coming from any of the 4 subpopulation memories and a write to generate a new solution (matrix A). In this case, we have chosen the south memory to store the generated solution. In turn, the mutation operation reads the matrix A passed by the crossover and processes it to another matrix A location (west memory). This procedure is followed by all the operations so that the memory bandwidth provided by the subpopulation memories is exploited to generate more efficient hardware structures. An exception to this is the constraint of the generated solution according to matrix C , since the algorithm requires two nested loops accessing to the same data and thus it is more convenient to keep the data in the same memory.

Listing 6.1 provides some excerpts of the implemented code, mainly the read and process of two commands (start the GA, and define a variable value), the crossover operation, and the correction of a solution according to matrix L (as defined by Equation (6.1)).

TABLE 6.3: Subpopulation memories accesses adopted by a PE for the different operations of the GA for solving the SA problem.

Operations	Subpopulation memories							
	north		south		east		west	
	read	write	read	write	read	write	read	write
Crossover	Sub-pop.	-	Sub-pop.	A	Sub-pop.	-	Sub-pop.	-
Mutation	-	-	A	-	-	-	-	A
Constraint Matrix L	L	-	-	-	-	A	A	-
Constraint Matrix C	C_0	-	-	-	A	A	C_1	-
Fitness	B_0	-	-	-	A	-	B_1	-
Replacement	-	Sub-pop.	-	Sub-pop.	A	Sub-pop.	-	Sub-pop.
Best solution	-	-	-	-	A	-	-	Best sol.

```

1 // pointers to access the different memories sections
2 static ac_int<32,false> *mat_A_N = &N_mem[0x000];
3 static ac_int<32,false> *mat_A_W = &W_mem[0x000];
4 static ac_int<32,false> *mat_A_S = &S_mem[0x020];
5 static ac_int<32,false> *mat_A_E = &E_mem[0x020];
6 static ac_int<32,false> *mat_L_N = &N_mem[0x040];
7 ...
8
9 // read and process commands
10 while (read_cmd_flag){
11     command_type_cGA command_in = PE_control_in.read();
12     ac_int<4,false> cmd_code = command_in.get_command();
13     ac_int<32,false> cmd_data = command_in.get_data();
14     switch (cmd_code){
15         case CMD_START_PE: //start the GA
16             stop_ga = 0; read_cmd_flag = 0; break;
17         case CMD_N_SU: // define number secondary users
18             n_su = cmd_data; break;
19         ...
20     }
21
22 // process GA
23 while (!stop_ga){
24     // selection
25     ...
26
27     // uniform crossover
28     for (int n=0; n<n_su; n++){
29         // generates a 32-bit random number

```

```

30     ac_int<32,false> rng_num = RNG_32bit.exec(global_RNG_1bit.read());
31     // ptr_P1 and ptr_P2 are pointers to the solutions selected
32     ac_int<32,false> data32_1 = ptr_P1[n];
33     ac_int<32,false> data32_2 = ptr_P2[n];
34     ac_int<32,false> data32_3;
35     LOOP_CX_32:    // loop label
36     for (int m=0; m<32; m++){
37         if (rng_num[m] == 0)
38             data32_3[m] = data32_1[m];
39         else
40             data32_3[m] = data32_2[m];
41     }
42     mat_A_S[i] = data32_3;
43 }
44
45 // mutation
46 ...
47
48 // correction of solution (constraint matrix L)
49 for (int n=0; n<n_su; n++){
50     mat_A_E[n] = mat_A_W[n] & mat_L_N[n]; // ensures that:  $a(n,n) \leq l(n,m)$ 
51
52     // correction of solution (constraint matrix C)
53     // fitness evaluation
54     // replacement
55     // best solution
56     ...
57 }

```

LISTING 6.1: Excerpt of the C++ code to describe a PE for solving the SA problem.

The code is written having in mind that a data word coming from a subpopulation memory is processed in parallel, if possible. For this reason and in general, the innermost loops are indexed to the channels of the SA problem and are limited to 32 regardless of the real number of channels for a given problem instance. In turn, the outer loops are indexed to the secondary users (the address of the memory) and are limited by the maximum number of secondary users for a particular instance (variable n_{su} in Listing 6.1).

6.2.3.3 HLS optimizations

The C++ description of the PE algorithm has been synthesized using different optimization strategies, mainly those concerning with unrolling and pipelining of loops. The first strategy is the basic mechanism to add parallelism to a design by scheduling multiple iterations of a loop in parallel. The second strategy consists of starting a loop iteration at every predefined number of clock cycles, called *initiation interval* (II) of a pipelined circuit.

In this work, the unroll of the loops is generally performed in the inner loops which are associated with the processing of the channels of the SA problem (e.g. of label `LOOP_CX_32` in line 35 of Listing 6.1). The outer loops are pipelined with the minimum possible initiation interval of each iteration, so that the latency of the circuit is minimized. For instance, the crossover has a minimum $II=3$ since in each iteration 3 simultaneous access may occur in the same subpopulation memory. In the operation to correct the solution according to matrix L , an $II=1$ is possible since all the data required comes from different memories and can be accessed in parallel (see Table 6.3). If an initiation interval is set to a value less than the admissible, the HLS tool fails in the scheduling phase.

A total of 14 systems have been synthesized with the HLS tool by setting cumulatively the following design constraints (e.g. design #6 includes all the constraints from #1 to #6):

- #1: Initial project without constraints.
- #2: Random number of 32 bits implemented with registers instead of BRAMs (used by default by Catapult HLS).
- #3: Shift-registers associated with all random numbers unrolled.
- #4: Unroll of crossover operation (inner loop).
- #5: Pipeline with $II=3$ in crossover operation.
- #6: Mutation with $II=3$ in mutation operation. 3 accesses to the global 1-bit random number limit the pipeline II .
- #7: Pipeline with $II=1$ in constraint matrix L .
- #8: Unroll of constraint matrix C (inner loop).
- #9: Pipeline with $II=2$ in constraint matrix C .
- #10: Unroll of fitness operation (inner loop). Loop generates 4 adders in parallel.
- #11: Pipeline with $II=1$ in fitness operation.
- #12: Pipeline with $II=2$ in replacement (copy of a new solution).
- #13: Pipeline with $II=1$ in copy best solution operation.
- #14: Change in C++ code so that 8 adders are parallelized in the fitness operation by unrolling the corresponding loops. Possible since matrix B is divided between two memories allowing the read of 64 bits in 1 clock cycle (resulting in 8 adders of 8 bits each).

TABLE 6.4: Catapult HLS and Precision RTL results for implementing a PE for the SA problem. The different solutions represent added constraints in Catapult HLS. Target frequency of 100 MHz in all solutions.

Solution	Catapult HLS		Precision RTL			
	Area cost	Latency (cycles)	LUTs	FFs	Slices	Frequency (MHz)
#1	5865.03	196930	3439	2509	860	144
#2	6398.19	101506	3743	2680	936	149
#3	5502.42	43360	3618	2417	905	160
#4	5204.75	42271	3382	2289	846	162
#5	5114.76	42238	3595	2284	899	161
#6	5231.63	42238	3480	2329	870	166
#7	5313.44	42174	3538	2327	885	172
#8	4939.47	10461	3146	2188	787	151
#9	4944.93	9531	3465	2168	867	153
#10	4769.32	8475	3161	2097	791	138
#11	4994.45	7850	3289	2213	823	142
#12	4953.72	7818	3279	2209	820	149
#13	4944.17	7754	3361	2207	841	142
#14	4977.12	7622	3300	2189	825	150

Table 6.4 presents the results obtained by Catapult HLS and Precision RTL for the different experiments previously enumerated. Catapult HLS provides an area metric required to implement a solution and the number of clock cycles required to complete a call to the function (latency). The latency value provided by the tool, in the context of this project, must be interpreted qualitatively and not quantitatively. The reason for this, is that the complete function that implements the PE has several `while` cycles that are ended till a certain condition is met. Therefore, it is impossible for the tool to know how many iterations (and thus how many clock cycles), these loops will take¹. Additionally, several loops are only executed once during the execution of the function (e.g. the configuration of a subpopulation memory), and thus they do not have the same impact as the loops used to generate a new solution which are repeated for each new generation of the GA. Therefore, a lower value for the latency must be interpreted always as a better solution in the sense that it executes with less clock cycles. Quantifying this gain can be done, although the results must be carefully analysed taking in consideration the changes or constraints imposed to the circuit. Regarding Precision RTL synthesis results, the number of look-up tables (LUTs), flip-flops (FFs) and slices required by the target FPGA are presented as well as an estimation of the maximum clock frequency.

¹It is possible to constrain a loop in Catapult HLS so that the tool knows the maximum number of iterations of the loop. However, for the GA implemented we did not provide these constraints (to `while` loops) since we do not specify these values. When Catapult HLS cannot determine a maximum number of iterations for a loop, the tool considers that it executes only once.

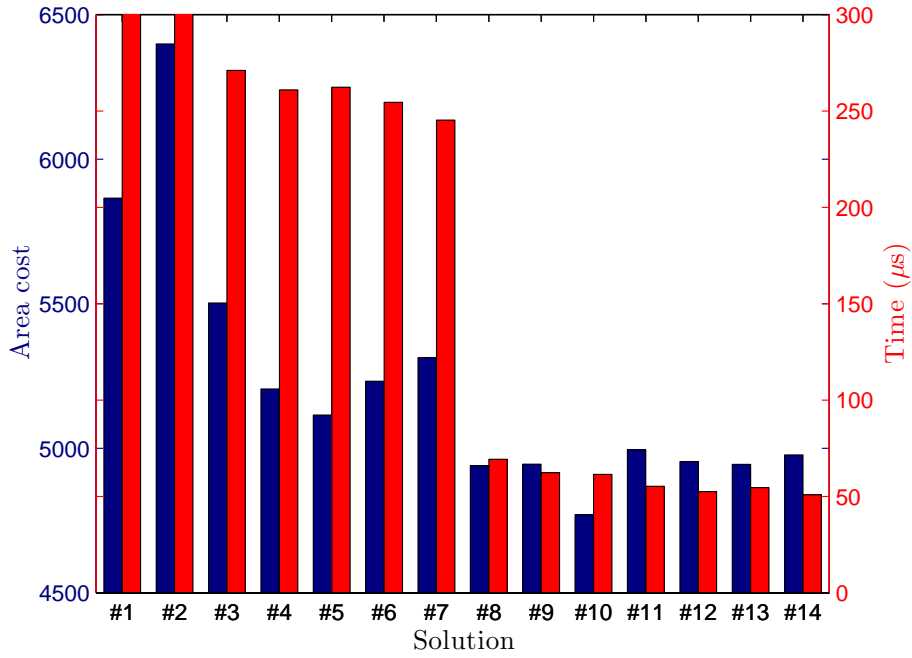


FIGURE 6.4: Area cost and computation time for the different PE implementation solutions with Catapult HLS and Precision RTL (in Table 6.4) to solve the SA problem.

Figure 6.4 presents a bar chart for the different solutions implemented with the area cost provided by Catapult HLS and the time that a solution takes to complete the execution of a PE (considering the latency and the frequency achieved by the circuit).

The target of all the constraints imposed by us to Catapult HLS has as objective a faster execution of the PE, although more hardware resources may be required to implement it. Therefore, we have set the design goal to latency in Catapult HLS. However, results clearly show the tool is not able to provide the desired results when compared to a design manually constrained by the user. Each new constraint adds an additional reduction in the latency value, which indicates that the tool does not optimize the loops by default.

As an example, solution #3 unrolls all the random numbers implemented as shift-registers, which represents a high decrease in the circuit latency. This happens since several random numbers exist in the circuit that were not implemented as shift-registers, thus affecting the circuit performance. Also, in solution #8 a large latency reduction exists when the loop unroll of the operations associated with matrix C correction is performed. This operation is the one that consumes more time in a GA generation.

An interesting result is that although the level of parallelism is augmented by unrolling loops, the area required to implement the final solution decreases. This result can be explained as the parallelized operations represent simpler circuits when compared to the

ones required for multiplexing the data for sequential processing. Most of the parallelized circuits are simple hardware functions (e.g. 32 parallel 2-input *and* operations for matrix L constraint).

Taking the constraints imposed in the final solution of Catapult HLS to implement the PE (solution #14 in Table 6.4), which provided the lowest latency circuit, we have performed a frequency sweep for several target frequency values as shown in Table 6.5. Additionally, each project was then synthesized by Precision RTL with three different optimization levels: no optimization, area optimization, and speed optimization.

From the results, we verify that there is not a meaningful change in the latency value for the different circuits generated by Catapult HLS. Also, the Precision RTL with speed optimization does not seem to provide improved results over the solutions with no optimizations. For instance, the best solution regarding the maximum frequency achieved is 155 MHz in both situations. When setting Precision RTL with area optimization, the generated circuits introduce additional digital signal processing (DSP) blocks to reduce the overall hardware resources used by the FPGA. However, in this situation the circuits generated achieve a lower frequency value than the other circuits with no optimizations or speed optimizations. In Table 6.5 we highlight 6 different projects (2 per each Precision RTL optimization level) that we elected as candidates to be synthesized with the Xilinx tools to build a complete cGAP with a single PE.

From the results in this section, it is clear that it is very important to describe properly the hardware block in C++ so that the tools can provide the best results. Moreover, by providing the appropriate constraints to Catapult HLS, the tool is guided to produce the best results, mainly in what concerns the unrolling and pipelining of loops.

6.2.4 cGAP implementation

In this section we present the results of the implementation in hardware of the cGAP, which includes the PE project described in the previous section together with the cGAC, both using HLS methodologies. This implementation corresponds to the RTL synthesis phase of the design flow (cf. Figure 5.6).

A cGAC project was also developed with Catapult HLS and Precision RTL, achieving a hardware utilization of 961 LUTs, 659 FFs, and 241 slices of the FPGA for a maximum frequency of 160 MHz. This project, by comparison with the PE to solve the SA problem, is much simpler to implement and require much less hardware resources. Therefore, during its development we have provided to Catapult HLS the necessary constraints so that the generated hardware unrolls and pipelines conveniently the loops, and no further

TABLE 6.6: Xilinx synthesis results taking 6 PE projects generated by the HLS tools (at bold in Table 6.5). Final implementation includes the complete cGAP with 1 PE and non-toroidal configuration of the cGAA.

Precision RTL solution	Xilinx synthesis				Frequency (MHz)
	Registers	LUTs	BRAMs	DSPs	
no opt. / 80 MHz	3635	4701	5	0	121.0
no opt. / 110 MHz	3687	4976	5	0	119.6
area opt. / 80 MHz	3625	4613	5	5	99.1
area opt. / 120 MHz	3716	4649	5	6	106.8
speed opt. / 70 MHz	3629	4794	5	0	105.5
speed opt. / 100 MHz	3694	4962	5	0	102.0

optimizations were performed. Moreover, the cGAC is a single hardware block in the cGAP, unlike the PE which is replicated several times, and thus it is not as critical as the PE.

In the last section we have elected 6 different projects developed with HLS tools for the PE. Since this block requires a considerable number of hardware resources and it is replicated several times, we have performed a synthesis of the cGAP with the Xilinx tools to determine which one is the best solution to be implemented. By doing so, we are also comparing the results provided by Precision RTL against the Xilinx tools.

Table 6.6 shows the results of the 6 mentioned PE circuits synthesized by the Xilinx tools. Moreover, we have included in the circuit synthesized the complete cGAP, which includes the 2 HLS projects (the PE and the cGAC) together with all the necessary blocks to support the cGAP. Since the only difference of these circuits is the PE, we can conclude which one is the best solution that has been previously generated by Precision RTL. We have chosen a cGAA with a single PE with non-toroidal configuration. As it can be seen, the solutions with optimization (both area and speed) provided by Precision RTL show a clear degradation in the maximum clock frequency when compared to the solutions with no optimizations. Additionally, the solutions with no optimizations require, in general, less hardware resources (including DSP blocks) than the others. Therefore, we have elected the final solution for the PE, the Precision RTL solution with no optimization and targeting 80 MHz, which provides the best maximum frequency and with reduced number of hardware resources. Also, the results show that the optimizations performed by Precision RTL (both in speed/area or target frequency) may not produce the expected results.

With the selected HLS projects, we have implemented the complete cGAP with the Xilinx tools. Tables 6.7 and 6.8 show the implementation results (after placement and

TABLE 6.7: Characteristics of the Xilinx implementation for cGAPs ranging from 1×1 to 6×6 PEs and toroidal configurations to solve the SA problem. Target FPGA is a Virtex-6 (XC6VLX240T-1).

	cGAP array (toroidal)					
	1×1	2×2	3×3	4×4	5×5	6×6
Registers	3738 (1.2 %)	11423 (3.8 %)	24182 (8.0 %)	41986 (13.9 %)	64899 (21.5 %)	92841 (30.8 %)
LUTs	4245 (2.8 %)	13579 (9.0 %)	29166 (19.4 %)	51133 (33.9 %)	78994 (52.4 %)	113656 (75.4 %)
Slices	1826 (4.8 %)	5971 (15.8 %)	11833 (31.4 %)	20438 (54.2 %)	30337 (80.5 %)	34435 (91.4 %)
BRAMs	3 (0.7 %)	9 (2.2 %)	19 (4.6 %)	33 (7.9 %)	51 (12.3 %)	73 (17.5 %)
Frequency (MHz)	115.2	95.6	74.0	66.9	78.1	75.7

TABLE 6.8: Characteristics of the Xilinx implementation for cGAPs ranging from 1×1 to 6×6 PEs and non-toroidal configurations to solve the SA problem. Target FPGA is a Virtex-6 (XC6VLX240T-1).

	cGAP array (non-toroidal)					
	1×1	2×2	3×3	4×4	5×5	6×6
Registers	3606 (1.2 %)	11159 (3.7 %)	23786 (7.9 %)	41458 (13.8 %)	64239 (21.3 %)	92049 (30.5 %)
LUTs	4097 (2.7 %)	13294 (8.8 %)	28666 (19.0 %)	50348 (33.4 %)	78021 (51.8 %)	112468 (74.6 %)
Slices	1740 (4.6 %)	4727 (12.5 %)	11665 (31.0 %)	21056 (55.9 %)	29218 (77.5 %)	35578 (94.4 %)
BRAMs	5 (1.2 %)	13 (3.1 %)	25 (6.0 %)	41 (9.9 %)	61 (14.7 %)	85 (20.4 %)
Frequency (MHz)	120.3	83.5	83.0	82.9	77.1	69.3

routing) for, respectively, toroidal and non-toroidal configurations of the cGAA. Also, we have focused on cGAA arrays with square configurations ranging from 1×1 to 6×6 PEs. Larger arrays have failed to implement in the target FPGA, since this device does not have enough hardware resources to implement them. Between the two groups of implementations, the toroidal configurations consume slightly more registers and LUTs than the non-toroidal. This happens since although a non-toroidal array presents more subpopulation memories for the same number of PEs, the memories that are placed in the borders of the array only connect to one PE. Therefore, the collision detection circuit

(cf. Figure 4.4) is simplified by the tools, resulting in less logic required by the cGAP.

The maximum frequency that a non-toroidal array can achieve decreases from 120 MHz to 69 MHz as the number of PEs increases from 1 to $6 \times 6 = 36$. These results are expected since as a project increases in size, it becomes more difficult for the tools to do the routing which results in a consequent decrease of the maximum frequency. Regarding the toroidal arrays, the configurations with 5×5 and 6×6 PEs achieve a higher frequency than smaller configurations. It is clear, that the tool should be able (by applying different constraints) to perform better in these situations. All these experiments have been performed by applying a constraint of 125 MHz to the tool, and by setting its effort level to maximum.

Since the design flow used to evaluate the cGAP uses a MicroBlaze soft-core processor connected to the cGAP, we have implemented this complete solution in the target FPGA. Due to this, we were able to accommodate together with the processor a cGAA with a maximum of 5×5 PEs. The maximum frequency target for this new solution is now 75 MHz for both toroidal and non-toroidal arrays at their maximum level of parallelism. It should be emphasized that this frequency is imposed by the global frequency of the complete system (the MicroBlaze project with its peripherals) and other values may not be admissible. Therefore, we have chosen to target all the projects presented in the following sections to 75 MHz, although smaller arrays could achieve a higher frequency.

6.2.5 cGAP results

In this section we analyse the results obtained by the algorithm running on the cGAP to solve the SA problem. The problem instances are labelled with the name N_M , where N represents the number of secondary users and M the number of channels. All the experiments have been averaged over 100 independent runs, with different initial populations, and different random numbers generated by the global RNG of the cGAP.

We first focus on the algorithm convergence for a 20_24 SA instance for different levels of parallelism, ranging from 1×1 to 5×5 PEs with both toroidal and non-toroidal configurations of the cGAA. Tables 6.9 and 6.10 show the parameters used to configure these cGAA so that the total population is as close as possible to 200 solutions. Since the projects described in the last section target a maximum of 16 solutions per subpopulation memory, we had to develop new cGAPs capable of accommodate the desired number of solutions for specific arrays that require more solutions.

Figure 6.5 illustrates the converge of the cGA for 10×10^6 new generated solutions in all the PEs. These graphs represent in the x-axis the total number of generated solutions,

TABLE 6.9: cGAP arrays with toroidal configurations used to solve the instance 20_24 of the SA problem.

	cGAP array (toroidal)				
	1×1	2×2	3×3	4×4	5×5
PEs	1	4	9	16	25
Subpopulation memories	2	8	18	32	50
Solutions/sub-population	100	25	11	6	4
Population size	200	200	198	192	200

TABLE 6.10: cGAP arrays with non-toroidal configurations used to solve the instance 20_24 of the SA problem.

	cGAP array (non-toroidal)				
	1×1	2×2	3×3	4×4	5×5
PEs	1	4	9	16	25
Subpopulation memories	4	12	24	40	60
Solutions/sub-population	50	16	8	5	3
Population size	200	192	192	200	180

and thus they do not reflect the acceleration obtained with cGAPs with more PEs. As it can be observed, all the cGAAs have an identical behaviour during the evolution of the algorithm, and there is not a clear difference among them. In addition, and by comparison with the TSP analysed during Chapter 3, there is not a clear pattern of the convergence rate for the different levels of parallelism. Also, both the toroidal and non-toroidal configurations lead to similar results.

An important observation that can be made with these results is the fact that the quality of the solutions obtained with larger arrays is not degraded when compared to the arrays with less PEs. As we have already discussed in Section 4.6, a global RNG feeds all the PEs which may result in correlated random numbers acquired by the different PEs. The array with 1×1 PE is not affected by this, and it does not show a superior performance over the remaining arrays. Therefore, we can conclude that for the SA instance analysed, the procedure used to generate random numbers does not affect negatively the rate of convergence in cGAAs with several PEs. This shows evidences that our approach of generating random numbers in the PEs can provide good results.

The results of the algorithm convergence have been obtained by sampling several points during the evolution of the algorithm, where the cGAP is stopped and the values are

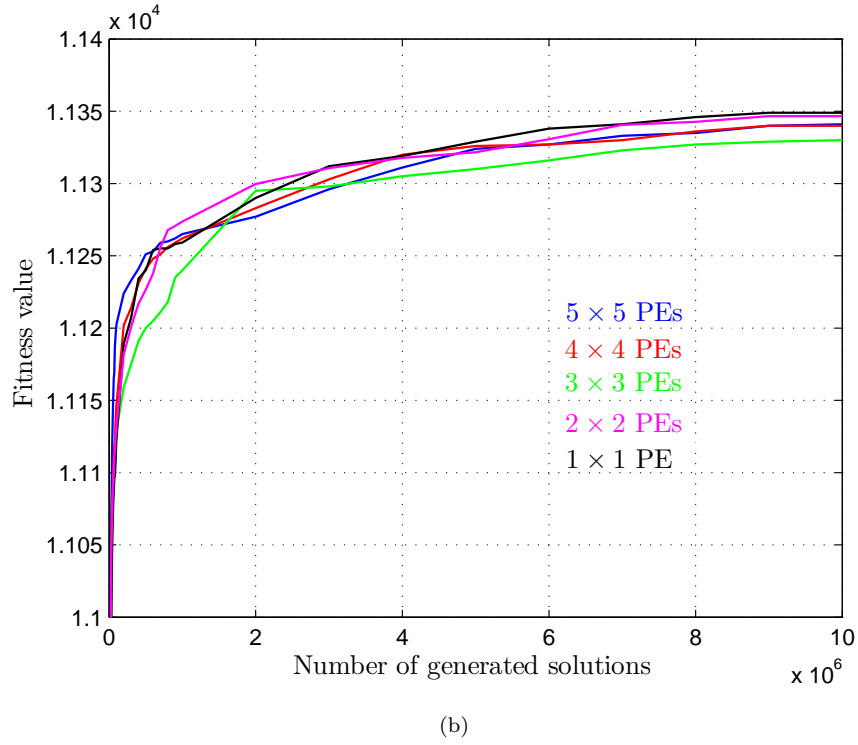
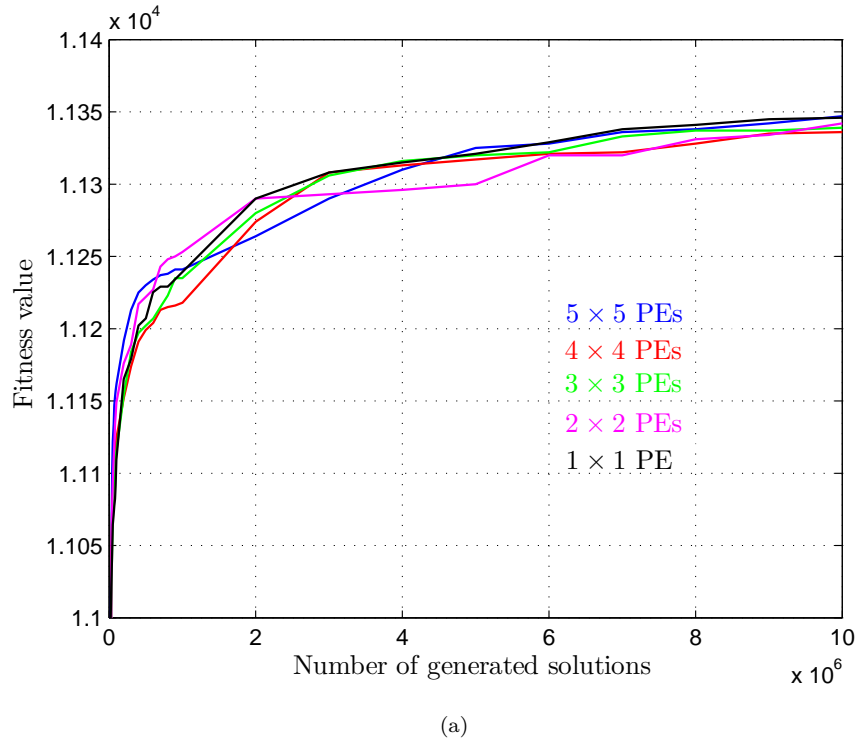


FIGURE 6.5: Fitness evolution with the total number of generated solutions, obtained for an instance 20_24 of the SA problem for (a) toroidal arrays and (b) non-toroidal arrays.

acquired by the MicroBlaze. Therefore, we do not see a smooth shape in the curves observed in Figure 6.5, as it was observed with the TSP in Chapter 3.

To evaluate the acceleration of the cGAP, we have focused on non-toroidal configurations of the cGAA since these require less hardware resources to be implemented when compared to toroidal configurations, and both toroidal and non-toroidal have led to similar results for solving the SA instance previously analysed. We have measured the speedup obtained with the different levels of parallelism when compared to a panmictic GA running on a processor of a PC, an Intel T8100 processor at 2.1 GHz, and the MicroBlaze soft-core processor implemented together with the cGAP in the same FPGA which runs at a clock frequency of 150 MHz. The implemented software was described in C and compiled with GCC -O3. Table 6.11 presents these results for 6 different instances of the SA problem, all run for 10^6 generated solutions. As it can be observed, for the maximum level of parallelism (5×5 PEs) the cGAP achieves a minimum acceleration of $2086 \times$ when compared to the MicroBlaze and $21.8 \times$ when compared to the PC.

For the different levels of parallelism, we also present in Table 6.11 the time and the throughput (measured in number of generated solutions in all PEs per μ s), and the normalized throughput in percentage. This last figure has been calculated considering the throughput per PE (and not in all the PEs) and considering the cGAA with 1×1 PE as reference for the different instances. As it can be seen, the throughput of a cGAP is almost directly proportional with the number of PEs. The results show a very small degradation ($< 0.4\%$) for the cGAP with the highest number of PEs, which represents the capacity of processing loss by these arrays when compared to a single PE. This slight loss happens due to the subpopulation memory access collisions that exist when the PEs access to these memories. Moreover, these values become more significant as the number of PEs increases, since smaller subpopulation sizes are used which lead to a higher probability of memory collisions. Additionally, the smaller instances are more affected by this, since the algorithms used to select the solutions (which are independent from the instance size) have more computational weight in a generation of a solution than with larger instances. Furthermore, as the 1×1 array is not affected by memory access collisions, we can conclude that these effects are almost negligible for the performance of the cGAP with larger arrays.

Figure 6.6 presents the speedup achieved for the different cGAPs compared to the 1×1 PE for the 32_32 instance. As it can be observed, an almost directly proportional acceleration is obtained with the level of parallelism.

We have also measured the quality of the final solution obtained by the cGAP at the end of 10^6 generations and compared it against a known heuristic based on a variant of the graph colouring problem, named colour-sensitive graph colouring (CSGC) [PZZ06]. Table 6.12 presents the results of the fitness values obtained by the different arrays together with the CSGC heuristic. As it can be seen, the cGAP in most of the cases surpasses the

TABLE 6.11: cGAP time results obtained for different SA problem instances. Array of PEs ranges from 1×1 to 5×5 with a non-toroidal configuration. A total of 10^6 solutions is generated for each trial.

Instance	cGAP				Speedup over SW	
	cGAA	Time (s)	Throughput (solutions/ μ s)	Normalized throughput	MicroBlaze	PC
5_6	5×5	0.082	12.129	99.62%	2086	21.8
	4×4	0.129	7.776	99.80%	1337	14.0
	3×3	0.228	4.379	99.92%	753	7.9
	2×2	0.513	1.948	99.98%	335	3.5
	1×1	2.054	0.487	100.00%	84	0.9
8_16	5×5	0.139	7.171	99.82%	3657	52.8
	4×4	0.218	4.592	99.89%	2342	33.8
	3×3	0.387	2.585	99.94%	1318	19.0
	2×2	0.870	1.149	99.98%	586	8.5
	1×1	3.480	0.287	100.00%	147	2.1
16_16	5×5	0.331	3.017	99.93%	3050	48.2
	4×4	0.518	1.931	99.96%	1953	30.8
	3×3	0.920	1.087	99.98%	1099	17.3
	2×2	2.070	0.483	99.99%	488	7.7
	1×1	8.281	0.121	100.00%	122	1.9
16_32	5×5	0.349	2.869	99.92%	5296	85.6
	4×4	0.545	1.836	99.95%	3390	54.8
	3×3	0.968	1.033	99.97%	1907	30.8
	2×2	2.177	0.459	99.98%	848	13.7
	1×1	8.708	0.115	100.00%	212	3.4
20_24	5×5	0.464	2.156	99.94%	3909	67.3
	4×4	0.725	1.380	99.96%	2502	43.1
	3×3	1.288	0.776	99.98%	1408	24.2
	2×2	2.898	0.345	99.98%	626	10.8
	1×1	11.588	0.086	100.00%	156	2.7
32_32	5×5	0.955	1.047	99.95%	4781	70.1
	4×4	1.492	0.670	99.96%	3060	44.9
	3×3	2.652	0.377	99.97%	1721	25.3
	2×2	5.966	0.168	99.97%	765	11.2
	1×1	23.856	0.042	100.00%	191	2.8

quality of this heuristic (a higher value means a better result). An exception to this is the largest instance (32_32) where the cGAP achieves systematically a lower value. This happens since the algorithm has stopped before it had time to converge to a satisfactory solution. From the graph in Figure 6.5(b) (for the 20_24 instance) it is evident that the algorithm continues to improve after 10^6 generations. Evolving the largest instance for a total of 2×10^6 generations shows already a superior fitness value than the CSGC heuristic.

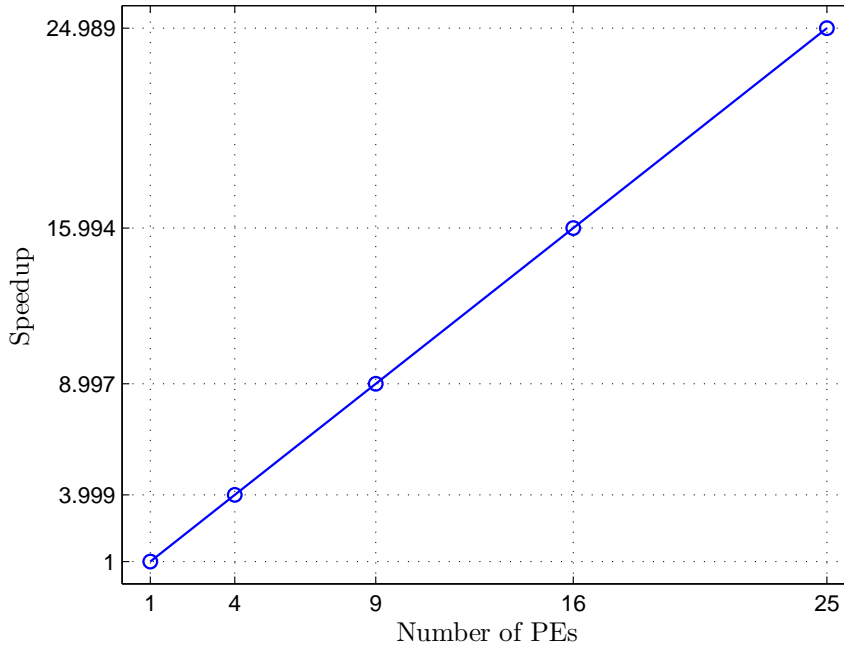


FIGURE 6.6: Speedup achieved by the cGAP with the level of parallelism for the 32_32 SA instance.

TABLE 6.12: Fitness results obtained for different SA problem instances with the cGAP and the CSGC heuristic. Array of PEs ranges from 1×1 to 5×5 with a non-toroidal configuration. A total of 10^6 solutions is generated for each trial.

	Instance					
	5_6	8_16	16_16	16_32	20_24	32_32
CSGC	1130	6090	6959	15197	11209	22882
1×1 cGAA	1130	6101	6965	15236	11268	22584
2×2 cGAA	1130	6117	6951	15234	11275	22645
3×3 cGAA	1130	6121	6939	15207	11254	22754
4×4 cGAA	1130	6112	6942	15192	11232	22746
5×5 cGAA	1130	6113	6946	15198	11265	22757

Although it is not our goal to develop and optimize the cGA to solve the SA problem by tuning several parameters of the algorithm (e.g. different crossover and mutation strategies), the results observed show already promising results as the algorithm provides good quality results. Moreover the speedup of the cGAP is scaled with the number of PEs and thus, with a larger FPGA, it could be possible to improve further the execution time of the algorithm.

Even though a fair comparison with other works is difficult to accomplish because there is not enough information on the details of the benchmarks used, the work presented in [ZPZS09] implements a GA for a 5_5 instance. The results reported show that a

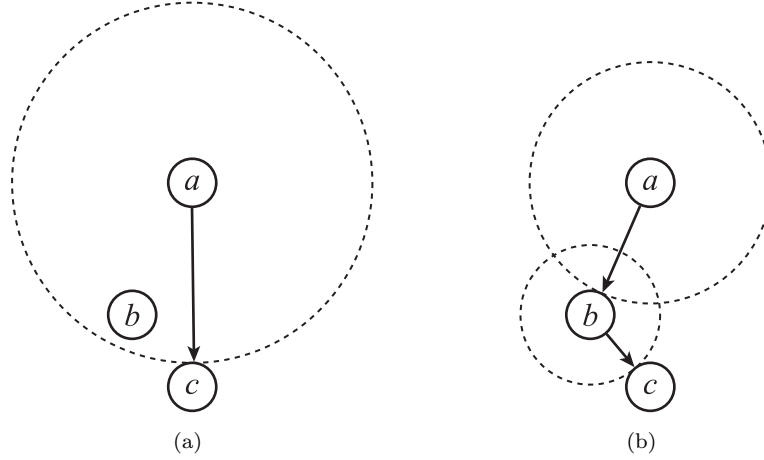


FIGURE 6.7: Example of a MEB problem with 3 nodes where (a) source node transmits to all the remaining nodes and (b) source node transmits to the closest node.

single solution takes an average of $547\mu\text{s}$ (0.093 s for running 10 iterations of the GA with 17 solutions each), running in a PC at 1.66 GHz with Matlab. With our 5×5 cGAP and a slightly larger instance (5_6), the average time for generating one solution is only 82 ns , which represents a speedup of $6670\times$ over the reported implementation for the throughput (number of generated solutions per time) of both algorithms. Once again, although this comparison is not fair since different instances are solved with different GAs and there are no details about the software implementation, it gives us an idea about the potential speedup that the cGAP can achieve.

6.3 Minimum energy broadcast

The minimum energy broadcast (MEB) problem is another optimization problem that arises with the use of the wireless networks. This problem consists in a set of nodes (wireless devices) where one of them has to broadcast a message to all the remaining nodes in that network with the minimum energy consumption.

Unlike in a wired network, in a wireless network a single transmission can be reached by several nodes, which means that when a node i transmits to node j , all the nodes placed near to i than j will also receive the transmission. Therefore, it is possible to increase/decrease a transmission range of a node so that more/less nodes are covered. For example and as shown in Figure 6.7(a), if a node a has to transmit to node b and c , it can transmit with enough energy so that both nodes are covered. Instead, a can transmit to the closest node, that in turn transmits to the last node, as depicted in

Figure 6.7(b). Although in the second case two consecutive transmissions are required, together they may consume less energy when compared to a single transmission.

The MEB problem is NP-hard [CCP⁺01] and thus metaheuristic procedures can be applied to solve this problem. For example in [MGD05] and [HCF12] the authors use respectively simulated annealing and particle swarm optimization (PSO) procedures to solve this problem. Additionally, different heuristics have also been developed like the algorithms called broadcast incremental power (BIP) [WNE00], embedded wireless multicast advantage (EWMA) [ČHE02], or largest expanding sweep search (LESS) [KP04].

In this section we present an adaptation of the work developed in [SB11], that proposes an hybrid GA, so that this is implemented in the cGAP. This algorithm consists of a combination of a GA where each solution is improved by applying a local search operator called *r-shrink* [DMES⁺03]. Although the authors name the algorithm as a hybrid GA, this combination of a GA with a local search is also known as memetic algorithm (MA). Therefore, we call it simply as MA. As in the original work, we will develop two solutions for different levels of the *r-shrink* heuristic: *1-shrink* and *2-shrink*, with increased levels in the intensity of the local search.

6.3.1 Problem definition

The MEB problem is specified as follows: given a direct graph $G = (V, E)$, where V denotes the set of nodes and E the set of edges of G , and a source node $s \in V$ that has to form a directed path (broadcast a message) to all other nodes in V . Therefore, this must be accomplished by creating an arborescence² of G rooted at s . For the MEB problem the nodes represent wireless devices, and the edges represent the transmission energy required for the communication between two devices. For a node in the arborescence, the transmission energy is determined by the longest edge among all the edges from which that node communicates to. The energy required by a leaf node in the arborescence is zero. The MEB problem consists in minimizing the total energy required to broadcast a message from s to all nodes in V , or to find an arborescence $T \subseteq E$ rooted at s that minimizes:

$$\sum_{i \in V} \max_{(i,j) \in T} d_{i,j}^\alpha \quad (6.4)$$

where $d_{i,j}$ is the Euclidean distance between nodes i and j , and α is the channel loss exponent that takes a value in the range of $2 \leq \alpha \leq 4$ depending on the characteristics of the communication medium.

²An arborescence in graph theory is a directed graph in which a vertex u , called the root, and any other vertex v there is exactly one directed path from u to v .

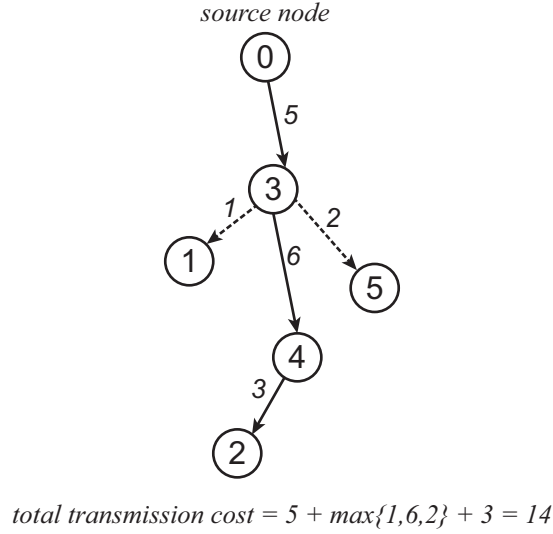


FIGURE 6.8: Example of a MEB solution with 6 nodes where solid edges represent the transmission costs and dashed edges represent implicit transmissions.

Considering $\alpha = 2$, as it is typically done, and knowing that the $d_{i,j}$ is given by:

$$d_{i,j} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \quad (6.5)$$

where (x_i, y_i) and (x_j, y_j) are the coordinates of nodes i and j respectively, the objective function of the MEB is now to minimize:

$$\sum_{i \in V} \max_{(i,j) \in T} \left\{ (x_i - x_j)^2 + (y_i - y_j)^2 \right\} \quad (6.6)$$

Figure 6.8 presents an example of a MEB solution with 6 nodes. As it can be seen, the source node (root) is node 0 and it transmits to node 3. In turn, node 3 transmits to node 4, and since nodes 1 and 5 are closer to 3 than node 4 ($3 \rightarrow 1 \leq 3 \rightarrow 5 \leq 3 \rightarrow 4$), they are implicitly covered by the transmission of node 3, as it is depicted by the dashed edges. Finally, node 4 transmits to node 2 to complete the arborescence of the MEB solution.

6.3.2 A memetic algorithm for the MEB problem

The algorithm used to solve the MEB problem has been adapted from [SB11], where a memetic algorithm (MA) is used. The metaheuristic consists in a GA with a local search procedure (or heuristic) that tries to improve the quality of all generated solutions during the evolution of the algorithm. Therefore, during a generation of a solution the MA applies the selection, crossover and mutation operations and, before the fitness evaluation and replacement, it improves the solution with the local search. Two

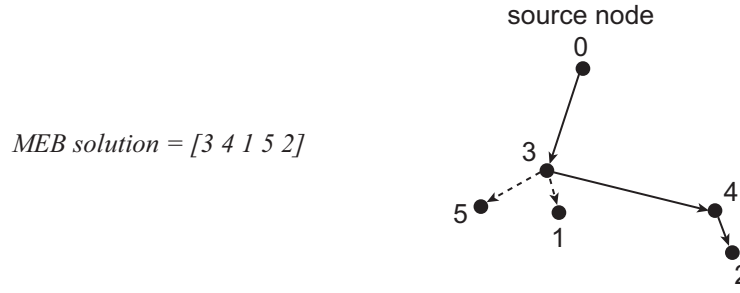


FIGURE 6.9: GA solution representation and codification used in the MEB problem.

different levels in the intensity of the local search (more solutions searched) have been implemented, namely 1-shrink and 2-shrink which are based on the improved procedure k -shrink proposed originally in [DMES⁺03].

In this section we will describe the algorithm used and how it has been adapted to the cGAP to solve the MEB problem.

6.3.2.1 Codification of solutions

As in any GA (or MA), a solution must be encoded to represent a tentative solution of the optimization problem, so that the genetics-inspired operators are applied. In [SB11] the path representation (or permutation encoding) is used to represent an MEB solution. Since this encoding scheme consists in a list of unique elements, it cannot be used directly to represent a solution (an arborescence), as it happens, for example, with the travelling salesman problem as discussed in Section 2.2.1.1. Therefore, a decoder is used to transform the solution representation to a valid MEB solution.

Figure 6.9 depicts an example of a MEB solution coded with the path representation and the corresponding arborescence. The algorithm used to decode the solution starts by selecting the first node on the list (node 3) and introduce it in the arborescence that, at the beginning, is formed only by a leaf node represented by the source node (node 0). Therefore, node 0 will transmit to node 3. Node 3 is then removed from the solution. After that, it is checked if there is any node from the remaining in the solution list that is implicitly covered by the previous transmission (in the example none is covered). The algorithm then proceeds in the same way: the first node on the list is chosen (node 4) and it is inserted in the arborescence so that a leaf node (node not transmitting) will transmit to the selected node with the minimum cost, which is node 3 (node 4 is closer to node 3 than node 0). The transmission from 3 to 4 is formed and implicit transmissions are checked that, for the example, are formed by $3 \rightarrow 5$ and $3 \rightarrow 1$. Nodes 4, 5 and 1

Algorithm 3 Pseudo-code of a MEB solution decoder and fitness evaluation.

```

1: solution                                ▷ GA's MEB solution to be decoded and evaluated
2: leaves_list  $\leftarrow$  [source_node]        ▷ list with all non-transmitting nodes
3: non_leaves_list  $\leftarrow$  []                ▷ list with all transmitting nodes
4: fitness  $\leftarrow$  0
5: i  $\leftarrow$  0
6: while not all nodes of solution processed do
7:   selected_node  $\leftarrow$  solution[i]
8:   selected_leaf  $\leftarrow$  select from leaves_list node that leads to less cost transmission
9:   update leaves_list and non_leaves_list
10:  fitness  $\leftarrow$  fitness + power transmission from selected_node to selected_leaf
11:  for all nodes in solution not processed do
12:    if node covered by transmission from selected_node to selected_leaf then
13:      update leaves_list
14:    end if
15:  end for
16:  increment i to next node in solution not processed
17: end while

```

are removed from the solution list. Finally the transmission from node 4 to 2 is formed with the same procedure.

Algorithm 3 presents the pseudo-code of the MEB solution decoder that has been implemented in the PEs of the cGAP. This algorithm is also used to evaluate the fitness value of the solution by adding the transmission costs of the nodes.

In a MEB solution representation list, we do not include the source node as it is done in [SB11], since this specific node has always the same position in the arborescence (the root). From the example in Figure 6.9, the nodes are labeled from 0 to 5, and any solution representation must include all the nodes except the source node (node 0). With this, we avoid the need to force the source node to be always the first element on the representation of the solution as it is done in the original work [SB11]. Moreover, any known crossover or mutation operator can be applied to our codification (path representation) and produce always a valid MEB solution.

6.3.2.2 Local search heuristic: *r*-shrink

The local search heuristic applied to the solutions generated by the crossover and mutation operations is the *r*-shrink proposed in [DMES⁺03]. This heuristic is based on *shrinking* the transmission energy of a node so that the arborescences disconnected from the original solution (nodes to which the broadcast is not performed due to the shrinking) is reassigned to other nodes by increasing their transmission energy. If the reduced energy is superior to the incremented energy for the resulting assignment, a

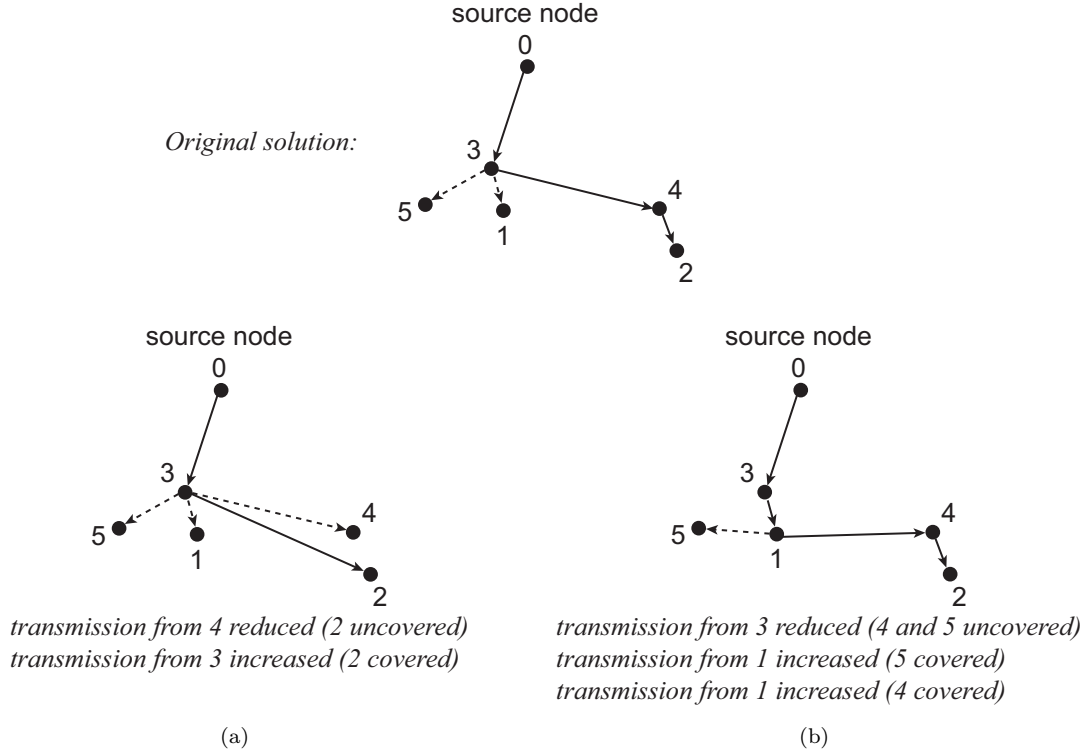


FIGURE 6.10: Possible local search moves by (a) 1-shrink and (b) 2-shrink in the MEB problem.

better solution is obtained. The r in the name r -shrink means the number of reduction steps performed by the algorithm. Therefore, in 1-shrink for each transmitting node a reduction on the energy is performed so that one node (with its corresponding arborescence) is left for reassigning to other node. With 2-shrink, a transmitting node (if transmitting to 2 or more nodes) reduces its energy so that two nodes are left for reassigning to other nodes. As in [SB11], we implement in the cGAP two solutions, both 1-shrink and 2-shrink.

Figure 6.10(a) illustrates an example of a possible 1-shrink move, where node 4 is elected to reduce its transmission. With this energy reduction, node 2 is left disconnected from the arborescence and thus it must be reassigned to other node by incrementing its transmission energy. The node elected to increase its transmission energy is the one that leads to the minimum energy increase among all possible nodes. For the example, all nodes (except nodes 2 and 4) are elected to transmit to node 2, but the one that leads to the minimum energy increase is node 3. Therefore, the transmission energy of 3 is increased so that it now transmits to node 2.

Algorithm 4 shows the pseudo-code of the 1-shrink procedure implemented in the cGAP. The algorithm starts by selecting one node that is transmitting and then it finds the best possible move that leads to the minimum energy increase. At the same time, it verifies

Algorithm 4 Pseudo-code of 1-shrink local search for the MEB problem.

```

1: non_leaves_list           ▷ list with all transmitting nodes (created by Alg. 3)
2: fitness                   ▷ MEB fitness value (created by Alg. 3)
3: number_nodes              ▷ Total number of nodes
4: total_fitness_gain  $\leftarrow 0$ 
5: while true do
6:   improvement_found  $\leftarrow 0$ 
7:   for all nodes in non_leaves_list (from last element to first) do
8:     node_reduce  $\leftarrow$  node from non_leaves_list
9:     node_sel  $\leftarrow$  node to which node_reduce transmits
10:    node_improvement_found  $\leftarrow 0$ 
11:    for  $i \leftarrow 0, \text{number\_nodes} - 1$  do
12:      if can node  $i$  transmit to node_sel then ▷ avoid cycles in arborescence
13:        calculate decremental_cost and incremental_cost
14:        if incremental_cost < decremental_cost then
15:          node_improvement_found  $\leftarrow 1$ 
16:          if best gain cost of all  $i$  nodes then
17:            node_fitness_gain  $\leftarrow$  decremental_cost – incremental_cost
18:          end if
19:        end if
20:      end if
21:    end for
22:    if node_improvement_found then
23:      improvement_found  $\leftarrow 1$ 
24:      total_fitness_gain  $\leftarrow$  total_fitness_gain + node_fitness_gain
25:    end if
26:  end for
27:  if not improvement_found then
28:    break
29:  end if
30:  update non_leaves_list
31: end while
32: fitness  $\leftarrow$  fitness – total_fitness_gain

```

if this move improves the solution fitness. If an improvement occurs, the algorithm starts again; if not, another transmitting node is selected for the same procedure. The algorithm stops when it cannot find any possible better solution.

According to the 1-shrink algorithm described in [SB11], the transmitting nodes are selected randomly. With our approach, we define the selection of these nodes by the inverse order as they appear in *non_leaves_list*, which is constructed by Algorithm 3. This was done since it simplifies the algorithm implementation and, at the same time, also ensures some randomness as the nodes order in this list is dependent on each MEB solution.

Figure 6.10(b) shows an example of a 2-shrink move. In this example, node 3 reduces its transmission energy so that 2 nodes (and the nodes to which they transmit to) are left

Algorithm 5 Pseudo-code of 2-shrink local search for the MEB problem.

```

1: non_leaves_list                                ▷ list with all transmitting nodes (created by Alg. 3)
2: fitness                                          ▷ MEB fitness value (created by Alg. 3)
3: number_nodes                                    ▷ Total number of nodes
4: while true do
5:   do 1-shrink as defined in Alg. 4
6:   total_fitness_gain  $\leftarrow$  0
7:   for all nodes in non_leaves_list (from last element to first) do
8:     node_reduce  $\leftarrow$  node from non_leaves_list
9:     node_sel_i  $\leftarrow$  node to which node_reduce transmits
10:    node_sel_j  $\leftarrow$  farthest node to which node_reduce implicitly transmits
11:    if node_reduce transmits to more than 1 node then
12:      for  $i \leftarrow 0, \text{number\_nodes} - 1$  do
13:        for  $j \leftarrow 0, \text{number\_nodes} - 1$  do
14:          if can node  $i$  transmit to node_sel_i and  $j$  to node_sel_j then
15:            if inc_cost < dec_cost then
16:              node_improvement_found  $\leftarrow$  1
17:              if best gain cost of all  $i$  and  $j$  nodes then
18:                node_fitness_gain  $\leftarrow$  dec_cost - inc_cost
19:              end if
20:            end if
21:          end if
22:        end for
23:      end for
24:      if node_improvement_found then
25:        improvement_found  $\leftarrow$  1
26:        total_fitness_gain  $\leftarrow$  total_fitness_gain + node_fitness_gain
27:      end if
28:    end if
29:  end for
30:  if not improvement_found then
31:    break
32:  end if
33:  update non_leaves_list
34: end while
35: fitness  $\leftarrow$  fitness - total_fitness_gain

```

disconnected from the original arborescence, in the case, nodes 4 and 5. Then, two nodes are elected to increase their transmission energy so that nodes 4 and 5 are reintroduced in the solution. Once again, these nodes must be chosen so that the incremental energy is the minimum possible. For the example, node 1 is the one that leads to the minimum incremental energy while ensuring the desired transmission coverage. It should be emphasized, that we can choose two different nodes or the same node for the incremental transmission, as it happens in this example. Therefore, node 1 now transmits to node 4 and implicitly covers node 5.

Algorithm 5 presents the pseudo-code of the 2-shrink procedure implemented in this

TABLE 6.13: Genetic operations adopted in the PEs for the MEB problem.

Parent selection	probabilistic binary tournament (75 % to accept best solution)
Crossover	maximal preservative crossover
Mutation	swap 2 nodes (maximum 3 swaps with probability 75 % each)
Local search	1-shrink or 2-shrink
Replacement	select random solution and replace if better

work. The core algorithm is identical to the 1-shrink, but instead of moving 1 node from one place to another in the MEB solution, now 2 nodes are moved. As proposed in [SB11], the 2-shrink starts first with the 1-shrink procedure to improve the solution.

6.3.2.3 The cGA operations

Even though our implementation is based on the algorithm implemented in [SB11], we adapted some operations of the original algorithm so that it could be executed in the cGAP. Additionally, we also used a different crossover operation that improved the quality of the final solution found.

Table 6.13 shows all the operations used by the cGA, and thus implemented in the PEs, that we have used to solve the MEB problem. Both the selection and mutations operators are identical to the original work. The local search operation, as described in the previous section, is also identical to the original.

Nevertheless, the GA in [SB11] is a steady-state GA, where a single solution replaces an existing one at each generation of the algorithm. In addition, it is ensured that no duplicate solutions appear in the population, and thus if the generated solution is already in the population, this is discarded. With our cGAP, this feature is not possible to implement since a PE only connects to its subpopulations and not to the entire population. Consequently, we have neglected this feature during the replacement operation. Moreover, it is not a common practice in a GA to verify if the generated solution is unique, unless the generation of solutions (crossover and mutation) is not capable of providing new solutions so that the entire search space is explored. Therefore, we have chosen for the replacement operation a known strategy used in cGAs, which consists in randomly pick up a solution (from all the subpopulations connected to the PE) and replace it with the new generated solution if this has a better fitness value.

With the described genetic operations, we have implemented in software the MA with two different crossovers operators: the maximal preservative crossover (MPX), and the cycle crossover which has been used originally in [SB11]. This implementation has targeted to perform a quick evaluation of these two crossovers using a panmictic MA to

solve the MEB problem. From the results, the MPX has consistently obtained better results. This is not surprising since the cycle crossover algorithm ensures that a generated solution keeps always the nodes position as they appear in the list of the two selected solution (the parent solutions). However, and from what we have discussed in Section 6.3.2.1, it is not the position of the nodes in the list of the MEB solution coded with the path representation that defines the solution, but instead the relative positions of the nodes placed in that list. On the other hand, the MPX operator ensures that the order of the nodes in the parents' solutions are preserved (till a certain limit) when the new solution is generated, thus preserving part of the original solutions. Therefore, we have elected the MPX to be implemented in the PEs of the cGAP.

6.3.3 cGAP implementation

This section presents the implementation details of the cGAP for solving the MEB problem. As described previously, the metaheuristic used is a MA and two distinct solutions are implemented for the local search heuristic: 1-shrink and 2-shrink. Although we could develop a single hardware solution with the 2-shrink (which includes the 1-shrink algorithm), the hardware resources required for a dedicated implementation of the 1-shrink are inferior, thus leading to an increase in the number of PEs implemented in the target FPGA.

As opposite to the SA problem (cf. Figure 6.3), we have developed a solution for the MEB where the variables that require hardware memory (arrays) are not kept in the subpopulation memories. The reason for this is that the algorithm implemented in the PE is not as simple as the one used in the SA problem, and the number of arrays required to manipulate an arborescence (especially during the 1- and 2-shrink algorithms) is large as we will see. Therefore, and since a PE is limited to four subpopulation memories, the access to them may limit the initiation interval of the pipelined circuits, which decreases the performance of the PE. In addition, by reducing the number of accesses in different parts of the PE algorithm to the subpopulation memories, the hardware resources required to access them (multiplexers) become simpler. Nevertheless, it is evident that with this approach a large number of block RAMs (BRAMs) will be required to implement a PE.

Figure 6.11 depicts the subpopulation memory organization used to solve the MEB problem, both for the 1-shrink and 2-shrink implementations, which is identical in all the four subpopulation memories (north, south, east and west) that connect to a PE. As it can be seen, we only have kept the nodes' coordinates of the MEB problem as additional data besides the solutions. With this, we can have a maximum of two parallel

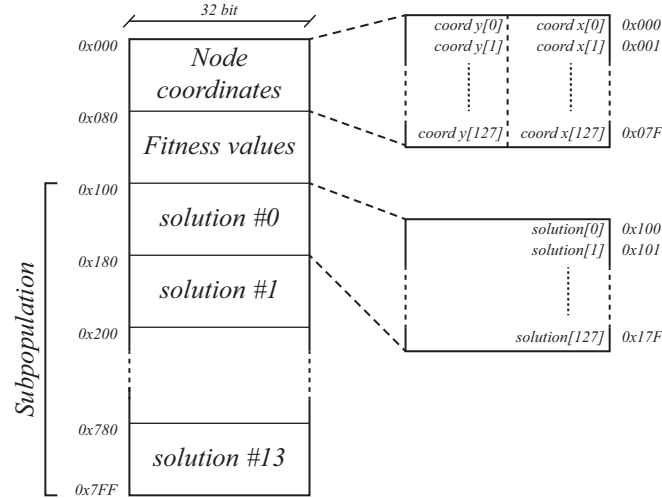


FIGURE 6.11: Subpopulation memory organization in the cGAP used for the MEB problem.

blocks to calculate each one the cost between two nodes, where the 4 memories are read simultaneously to provide the coordinates of the 4 nodes. The information kept in the subpopulation memories does not stall the pipelined circuits since the solutions and the nodes' coordinates are accessed in different parts of the PE algorithm, mainly during crossover and replacement for the solutions, and during fitness/decoder and local search for the coordinates.

We would like to emphasize that, as in the TSP discussed in Section 3.4.1, we could have kept the cost between two nodes in memory, instead of the coordinates and calculate their distance cost. However, this leads to a huge increase in memory space required to keep this information.

To keep the subpopulation memories with two BRAMs, we have limited our implementation to a maximum of 128 nodes and subpopulations with a maximum of 14 solutions. These memories have been configured with a width of 32 bits, which is required for the coordinates (a maximum of 16 bits per x or y coordinate as depicted in Figure 6.11). Additionally, we also keep a node of a solution per memory address (which requires 7 bits). Once again, with this approach we simplify the algorithm description and thus the algorithm hardware implementation, although memory is being wasted as it is not used.

As stated previously, we have used several arrays (memories) to describe the PE algorithm. All of these use the minimum width required to keep the information and a length of 128. The following list describes briefly these memories:

- `solution`: List with solution of the MEB problem generated by crossover and mutation to be decoded and evaluated.

- `leaves_list`: List with all nodes that do not transmit in a MEB solution (leaves in the arborescence).
- `nonleaves_list`: List with all nodes that transmit in a MEB solution (non-leaves in the arborescence).
- `mem_parent_nodes`: For all nodes, it provides which node is transmitting to it. For example, if node 1 transmits to node 5, then `mem_parent_nodes[5] = 1`. Used by 1-shrink and 2-shrink to avoid cycles in the arborescence when nodes are moved.
- `mem_child_nodes_1`: It provides the farthest node (if any) to which a node transmits to. Therefore, it presents a decoded solution. For example, if node 1 transmits to node 5, then `mem_child_nodes_1[1] = 5`. Used during the calculation of the decremental cost of the 1- and 2-shrink algorithms, and to avoid cycles in the arborescence when nodes are moved in these algorithms.
- `mem_child_nodes_2`: It provides the second farthest node (if any) to which a node transmits to. For example, if node 1 transmits to node 5, and it implicitly covers node 2 and this node is the farthest implicitly covered by 1, then `mem_child_nodes_2[1] = 2`. Used only by 2-shrink to avoid cycles in the arborescence when nodes are moved.
- `cost_1_trans_nonleaves`: It provides the transmission cost for all the nodes. The array is constructed during the decoder/fitness algorithm and it is used by the 1- and 2-shrink algorithms so that these transmissions costs are not explicitly recalculated.
- `cost_2_trans_nonleaves`: Same as `cost_1_trans_nonleaves`, but the array keeps the transmission costs required for the second farthest node reached by the transmitting node.
- `cost_3_trans_nonleaves`: Same as `cost_1_trans_nonleaves`, but the array keeps the transmission costs required for the third farthest node reached by the transmitting node. Used only by the 2-shrink algorithm.
- `best_sol`: It keeps the best solution found by the PE. The solution is saved after the decoder and local search operations are applied (format equal to the array `mem_child_nodes_1`).
- `flag_cx`: Auxiliary memory (1 bit width) used during the calculation of the MPX algorithm.

TABLE 6.14: Characteristics of the different projects used to implement the cGAP with 1-shrink local search for solving the MEB problem. Target FPGA is a Virtex-6 (XC6VLX240T-1).

	Registers	LUTs	Slices	BRAMs	DSPs	Frequency (MHz)
Precision RTL (PE)	3060 (1.0 %)	3996 (2.7 %)	999 (2.7 %)	5 (1.2 %)	17 (2.2 %)	87.6
ISE synthesis (cGAP - 5×5)	90214 (29.9 %)	116093 (77.0 %)	-	246 (59.1 %)	425 (55.3 %)	80.3
ISE (cGAP + MicroBlaze)	115759 (38.4 %)	130943 (86.9 %)	37212 (98.8 %)	207 (49.8 %)	431 (56.1 %)	> 75

- `flag_fitness`: Auxiliary memory (1 bit width) used during the decoder/fitness algorithm.

In the implementation we have used a total of 14 bits for each coordinate of a MEB node which is sufficient for the instances used in this work. Therefore, the cost to transmit from one node to other requires a total of 28 bits. The maximum fitness values have been limited to a 32-bit representation, with a saturated adder to avoid overflow errors during the fitness calculation.

We have developed the PE and cGAC hardware blocks with Catapult HLS and Precision RTL with an identical approach as the one presented in Sections 6.2.3.3 and 6.2.4. We have pipelined all the innermost loops of the algorithms and, in the majority of the cases, we could achieve pipelined circuits with an initial interval of 1. This was possible since we have used independent memories to describe the algorithm as stated previously. In this PE project we did not implement parallel structures (besides the ones required to ensure the $\Pi=1$ of the pipelines) and thus no unrolling of loops was performed. Moreover, this was not possible due to the memory organization followed which provides a single data value to be processed in a clock cycle.

At the end of the Catapult HLS project, we have reached a circuit with a latency value that reaches more than 2×10^6 clock cycles for the 1-shrink, and more than 200×10^6 clock cycles for the 2-shrink projects. As stated in the SA project, the latency value provided by Catapult HLS must not be interpreted as a real value for our projects to implement a GA in a PE. However, these figures show the increased temporal complexity of the algorithm when compared to the SA problem, which reached only more than 7×10^3 clock cycles.

Tables 6.14 and 6.15 provide the implementation details of the main projects used for implementing, respectively, the MEB with 1-shrink and 2-shrink local search heuristics.

TABLE 6.15: Characteristics of the different projects used to implement the cGAP with 2-shrink local search for solving the MEB problem. Target FPGA is a Virtex-6 (XC6VLX240T-1).

	Registers	LUTs	Slices	BRAMs	DSPs	Frequency (MHz)
Precision RTL (PE)	3738 (1.2 %)	5992 (4.0 %)	1498 (4.0 %)	7 (1.7 %)	4 (0.5 %)	104.8
ISE synthesis (cGAP - 4×4)	68974 (22.9 %)	106437 (70.6 %)	-	185 (44.5 %)	64 (8.3 %)	95.8
ISE (cGAP + MicroBlaze)	94695 (31.4 %)	114090 (75.7 %)	37374 (99.2 %)	165 (39.7 %)	70 (9.1 %)	> 75

The PE projects provided by Precision RTL have been optimized/constrained so that they could implement the cGAP together with the MicroBlaze in the target FPGA. As it can be seen, the cGAP for the 1-shrink MA achieves a maximum of 5×5 PEs, while the 2-shrink MA achieves 4×4 PEs. Both projects have an impressive level of slice utilization in the FPGA that is approximately 99 %.

The 1-shrink Precision RTL project has been optimized for area and therefore it has used a higher number of DSP blocks (and less slices than without this optimization) targeting a frequency of 140 MHz, whereas the 2-shrink project has been implemented with no optimizations and targeting a frequency of 90 MHz. It should be emphasized that several projects with similar constraints have been tried to implement the final circuit. Both cGAPs run at 75 MHz.

6.3.4 cGAP results

The cGAPs described in the last section have been used to solve the same instances of the MEB problem reported in [SB11]. Mainly, we have considered two sets of instances, one with 20 nodes and other with 50 nodes, each with a total of 30 different instances. As in the original work, the results are averaged over 30 independent runs for each instance.

The initial populations used in all the experiments have been built offline (in a PC) and have been loaded to the subpopulation memories during the initialization phase of the algorithm. With this, we ensure good quality initial populations, which is important in a GA, avoiding thus the use of the global RNG of the cGAP to calculate these data. It should be emphasized that we want to evaluate the cGA by averaging several runs of an instance. Therefore, it is essential to ensure that the initial populations are generated randomly and with no correlation. Moreover, in a real application scenario of the cGAP it is not expected to execute several times the algorithm for the same

instance, especially if time constraints are applied where a solution must be obtained as fast as possible. Nevertheless, a possible final implementation of the cGAP can use the subpopulation memories, for example, to keep the initial population of the algorithm previously generated in a PC.

The execution times reported in this work do not include the initialization phase. This way, we avoid to measure access times required by the MicroBlaze processor to access, via NFS, to a PC where the file system is hosted. This configuration was used in this work to evaluate the cGAP and thus it does not reflect a real scenario application. However, the initialization requires loading a few parameters to configure the PEs (e.g. number of nodes, number of solutions per subpopulation, stop criterion), and assuming this data is in the cGA dual-port memory (see Figure 4.11) its execution time is negligible when compared to the complete execution time of the algorithm. As stated previously, the initial subpopulations, which represent a large amount of data transfer during the initialization, can be kept in the subpopulation memories. When the algorithm starts for solving an instance of the problem it must load the initial population to a memory location where they will then be evolved. In a real scenario, the same initial population can be used for different instances. This further reduces the time of the initialization of the algorithm.

A total population size of 400 solutions has been used in [SB11]. For the cGAPs using the 1-shrink and 2-shrink (cGAP-1s and cGAP-2s respectively) we have kept, approximately, the same amount of solutions. For the cGAP-1s, which uses an array of 5×5 PEs with a non-toroidal configuration, a total of 7 solutions is used per subpopulation, which results in a population of 420 solutions. In turn, the cGAP-2s, which has 4×4 PEs also with a non-toroidal configuration, requires 10 solutions per subpopulation to achieve exactly a total of 400 solutions.

The stop criterion used to finish the algorithm is to have more than $1000 \cdot n$ generated solutions, where n is the number of nodes, without improving the best solution, as described in [SB11]. To implement this feature in the cGAP, the PEs notify the cGAC when they found a local best solution, and when a certain number of generations has been elapsed. With this information, the cGAC knows what is the best solution in the cGAA and, approximately, how many generations have elapsed in all the PEs since the last best solution was found. Therefore, the cGAC realizes when the stop criterion has been met and broadcasts a command to stop all the PEs, thus stopping completely the cGAP.

Tables 6.16 and 6.17 present the results obtained for the two groups of instances of the MEB problem with 20 and 50 nodes respectively. We provide for each instance the average excess which measures in percentage the distance that the solutions obtained are

TABLE 6.16: Results performance of the cGAP-1s and cGAP-2s on 20 node MEB problems. Data is compared against software versions of the algorithms developed by [SB11].

Instance	GA-1s	cGAP-1s		GA-2s	cGAP-2s	
	Time (s)	Time (s)	Acceleration	Time (s)	Time (s)	Acceleration
p20.00	0.45	0.022	20.0	0.60	0.082	7.3
p20.01	0.60	0.029	20.7	0.79	0.164	4.8
p20.02	0.56	0.024	23.7	0.80	0.118	6.8
p20.03	0.76	0.030	25.1	1.19	0.164	7.3
p20.04	0.54	0.028	19.6	0.81	0.143	5.7
p20.05	0.43	0.021	20.8	0.59	0.108	5.4
p20.06	0.48	0.022	22.2	0.70	0.102	6.9
p20.07	0.60	0.029	20.8	0.80	0.108	7.4
p20.08	0.71	0.036	19.6	1.35	0.298	4.5
p20.09	0.61	0.034	18.2	1.01	0.219	4.6
p20.10	0.61	0.022	27.8	1.04	0.141	7.4
p20.11	0.40	0.024	16.7	0.61	0.094	6.5
p20.12	0.50	0.033	15.1	0.62	0.122	5.1
p20.13	0.41	0.022	18.9	0.90	0.157	5.7
p20.14	0.61	0.026	23.0	1.18	0.192	6.1
p20.15	0.46	0.024	19.6	0.69	0.118	5.8
p20.16	0.74	0.031	23.6	1.20	0.153	7.8
p20.17	0.64	0.025	25.1	1.01	0.129	7.8
p20.18	0.49	0.025	19.5	0.66	0.095	6.9
p20.19	0.67	0.029	23.0	1.20	0.201	6.0
p20.20	0.61	0.028	21.8	0.81	0.120	6.8
p20.21	0.54	0.024	22.9	0.67	0.075	8.9
p20.22	0.46	0.026	17.4	0.57	0.091	6.3
p20.23	0.56	0.021	26.9	0.78	0.091	8.5
p20.24	0.65	0.028	22.9	1.07	0.157	6.8
p20.25	0.45	0.023	19.2	0.62	0.135	4.6
p20.26	0.55	0.027	20.0	0.96	0.155	6.2
p20.27	0.54	0.022	24.1	0.83	0.090	9.2
p20.28	0.56	0.024	23.4	0.89	0.100	8.9
p20.29	0.65	0.021	31.4	0.89	0.092	9.7
Averages	0.56	0.026	21.8	0.86	0.134	6.7

from the optimum, the number of times the optimum solution is found out of 30 trials, and the average execution time. Since for the 20 node instances the optimum solutions was always achieved by the cGAP and by the software, we only report on Table 6.16 the figures concerning the execution time. All the results are compared against the ones found in the original GAs (GA-1s and GA-2s, respectively for the 1- and 2-shrink), which were implemented in C and executed on a Pentium 4 system running at 3.0 GHz.

Additionally, we measure the acceleration of the cGAPs over the corresponding version of the software GAs.

As it can be seen from Table 6.16, for the 20 node problems the cGAP-1s executes in average $21.8\times$ faster than the GA-1s, whereas the cGAP-2s achieves an acceleration of $6.7\times$. For the instances with 50 nodes (Table 6.17), the cGAP-1s finds in average the optimum 17.37 times out of 30, while the original algorithm provides a slightly better result of 19.27 out of 30. However, the excess figure is better for the cGAP-1s, with 0.63%, against 0.81% of the GA-1s. Therefore, even though the cGAP-1s achieves less times the optimum, in average it achieves better quality solutions. For the cGAP-2s both the excess and the number of times the optimum is achieved, is in average better than the GA-2s, with 0.10% against 0.25%, and 25.87/30 against 22.9/30 for these figures.

However, the acceleration is degraded as the number of nodes increases, both for the cGAP-1s and cGAP-2s. Moreover, the cGAP-2s achieves a smaller acceleration than the cGAP-1s when compared to their software counterparts. Nevertheless, it should be emphasized that the cGAP-2s has a total of 16 PEs, whereas the cGAP-1s has 25 PEs. Therefore, the cGAP-1s can produce $25/16 = 1.56$ times more solutions than the cGAP-2s for the same time interval. Additionally, it should be emphasized that the algorithms implemented in both the cGAPs and in software [SB11] are slightly different. As described in Section 6.3.2.3, the original algorithm is a steady-state GA while our is a cGA. Moreover, we have used the maximal preservative crossover instead of the cycle crossover, since it has produced better results. Also, in the original algorithm it is checked if a generated solution is unique in the population, and if the algorithm does not find a new solution for 20 consecutive generations, it stops. In our cGAP, this stopping criterion is not implemented since we do not check for uniqueness in the population. This could help to explain why we achieve a slower acceleration figures for the 2-shrink version than for the 1-shrink. As the 2-shrink is a more intensive local search, it is likely to map more generated solutions by crossover and mutation to a single final and improved solution. Therefore, in the GA-2s the criterion of the 20 consecutive iterations without generating a new and unique solution in the population happens more frequently than in the GA-1s.

Additionally, the 2-shrink HLS project is more complex than the 1-shrink since it has a higher level of nested loops (cf. Algorithms 4 and 5). Since the constraints imposed by us to the HLS tool to the initiation interval of a loop were performed to the innermost loops (the others cannot be constrained), it is likely that the implementation of the 2-shrink is not as optimized as the 1-shrink and therefore results are worst in the cGAP-2s when compared to the cGAP-1s.

of the algorithms developed by [SB11].

Instance	GA-1s			cGAP-1s			GA-2s			cGAP-2s			
	Excess (%)	Found	Time (s)	Excess (%)	Found	Time (s)	Excess (%)	Found	Time (s)	Excess (%)	Found	Time (s)	Acceleration
p50.00	1.2	0/30	7.12	1.52	0/30	0.39	18.4	6/30	12.82	0.74	4/30	7.57	1.69
p50.01	0.2	26/30	7.88	0.02	28/30	0.59	13.2	9/30	20.18	0.18	18/30	8.08	2.50
p50.02	0.36	26/30	6.01	0.06	29/30	0.38	15.7	30/30	9.76	-	30/30	4.93	1.98
p50.03	0.7	20/30	10.47	1.14	2/30	0.58	18.1	30/30	15.28	0.01	23/30	7.08	2.16
p50.04	0.49	2/30	7.59	0.17	20/30	0.43	17.5	8/30	13.15	-	30/30	5.97	2.20
p50.05	0.06	29/30	5.1	-	30/30	0.26	19.4	30/30	9.61	-	30/30	3.46	2.78
p50.06	0.3	20/30	7.77	-	30/30	0.32	22.3	30/30	9.2	-	30/30	3.35	2.74
p50.07	1.21	6/30	7.12	0.19	27/30	0.55	14.2	1/30	13.24	0.10	28/30	5.70	2.32
p50.08	-	30/30	3.51	-	30/30	0.27	13.2	30/30	6.71	-	30/30	4.63	1.45
p50.09	2.45	10/30	10.25	1.71	5/30	0.74	13.9	17/30	21.84	0.32	22/30	15.15	1.44
p50.10	0.67	20/30	6.59	0.48	17/30	0.49	13.3	30/30	14.14	0.04	27/30	6.84	2.07
p50.11	3.31	1/30	5.39	2.70	1/30	0.40	13.6	25/30	15.05	0.10	26/30	6.81	2.21
p50.12	5.75	0/30	9.37	1.09	0/30	1.07	8.8	20/30	18.57	0.07	27/30	8.75	2.12
p50.13	0.05	27/30	12.42	0.16	5/30	1.29	9.6	11/30	25.55	-	30/30	9.52	2.69
p50.14	0.19	27/30	2.71	-	30/30	0.29	9.4	30/30	4.12	-	30/30	2.52	1.63
p50.15	0.31	18/30	7.22	0.09	23/30	0.52	14.0	30/30	13.09	-	30/30	4.62	2.83
p50.16	2.73	0/30	13.56	1.92	1/30	1.46	9.3	1/30	30.19	0.34	14/30	18.82	1.60
p50.17	0.03	28/30	7.72	0.16	17/30	0.50	15.4	28/30	14.37	-	30/30	4.31	3.33
p50.18	≤ 0.00	29/30	8.66	0.13	20/30	0.77	11.3	24/30	16.17	≤ 0.00	25/30	6.34	2.55
p50.19	-	30/30	6.3	-	30/30	0.28	22.8	30/30	10.57	-	30/30	3.70	2.85
p50.20	0.17	0/30	6.37	0.16	0/30	0.44	14.6	4/30	10.29	0.01	27/30	4.98	2.06
p50.21	0.28	27/30	6.18	-	30/30	0.33	19.0	30/30	10.61	-	30/30	4.02	2.64
p50.22	-	30/30	4.82	-	30/30	0.24	20.0	30/30	8.32	-	30/30	3.89	2.14
p50.23	1.7	17/30	10.69	2.02	1/30	0.78	13.7	30/30	12.49	0.18	25/30	8.54	1.46
p50.24	0.05	28/30	6.52	0.29	18/30	0.55	11.9	30/30	12.82	-	30/30	5.34	2.40
p50.25	-	30/30	4.15	-	30/30	0.27	15.3	30/30	6.37	-	30/30	3.32	1.92
p50.26	0.79	22/30	9.18	3.65	1/30	0.50	18.4	27/30	15.26	1.01	4/30	10.47	1.46
p50.27	-	30/30	6.87	0.03	26/30	0.38	18.0	29/30	9.85	-	30/30	4.36	2.26
p50.28	1.39	15/30	10.84	1.23	10/30	0.67	16.1	27/30	23.69	0.03	26/30	8.01	2.96
p50.29	-	30/30	6.29	-	30/30	0.28	22.6	30/30	8.1	-	30/30	3.47	2.34
Averages	0.81	19.27/30	7.49	0.63	17.37/30	0.53	15.4	22.9/30	13.71	0.10	25.87/30	6.49	2.23

To conclude, we achieve an average acceleration of, at least, $2.23\times$ for the cGAP-2s when compared to the GA-2s in software for the instances analysed. However, since we have a less constrained stopping criterion we obtain superior quality solutions.

More importantly, these results show that it is possible to port an existing GA to our cGAP using the proposed HLS design flow. The MEB problem is an optimization problem that to be solved with a GA requires more complex algorithms than the ones found in typical GA operators. Indeed, for this example, the difficulty is clearly in the local search procedure algorithms and in the decoder of a solution.

6.4 Considerations on the cGAP implementation

The HLS-based design flow has proven to successfully implement the PE and the cGAC blocks required for building the cGAP. However, it should be kept in mind that when describing a hardware block in C++ the code must be carefully structured so that good results are achieved. This way, it will be possible for the tools to infer the desired hardware structures, like pipelined circuits and parallelization of operations, and thus provide good results. Therefore, a straight coding style, like the one used to describe software in a sequential language, may not produce the best results in HLS. Moreover, the organization of the data arrays, and their consequent mapping to hardware memories, plays an important role to successfully implement the desired hardware structures. From our experience with Catapult HLS, it is crucial to structure the code, while applying the correct design constraints so that the tool implements the desired architecture. Although HLS raises the abstraction level when describing the hardware, design space exploration and architecture optimization, often require changes in the source code [MVBG⁺12].

The cGAP can achieve an acceleration that is proportional to the number of PEs. The impact in the performance of the architecture as the level of parallelism increases, for example due to the access to the subpopulation memories, is almost negligible. This happens since the majority of the computational effort in the GAs executed by the PEs (for the examples analysed in this chapter) do not involve accesses to the subpopulations. However, in algorithms requiring a high demand for accesses to the subpopulations (especially if the subpopulations have a small number of solutions) can reduce the expected acceleration of the cGAP.

The results have shown that the solution's quality found by the algorithm supported by the cGAP is not affected due to changes in the level of parallelism. Therefore, the architecture can be scaled to fulfil the desired requirements, both in the expected acceleration

and in the hardware resources required to implement it. Additionally, toroidal and non-toroidal configurations of the array of processors have also led to similar results in the solutions found by the algorithm. In this case, a non-toroidal array leads to a slight decrease in the logic required to implement it when compared to a toroidal array for the same number of PEs, although it requires more subpopulation memories. However, we must be aware that the changes in the number of PEs and in the configuration of the array, leads to changes in the neighbourhood of the solutions of the cellular genetic algorithm (while keeping the same population size), which alters the convergence rate of the algorithm as it has been discussed in Sections 3.2 and 3.3 of Chapter 3.

6.5 Summary

In this chapter we have used two real-world optimization problems to be implemented and solved by the cGAP, mainly the spectrum allocation problem and the minimum energy broadcast problem.

For the SA problem, we have exploited the subpopulation memories connected to the PEs so that they could be used to keep auxiliary data so that the performance of the PE is improved without using more hardware resources. Moreover, we have verified that it is essential to give to Catapult HLS tool the proper constraints so that the final hardware implementation is optimized. With the correct constraints of loop unrolling and pipelining, we have obtained not only a circuit that requires less hardware resources but, more importantly, a circuit with a substantially lower latency.

The results for the SA problem have shown that the use of the global RNG, which provides random numbers to all PEs, does not show to degrade the quality of the final solution found by the algorithm. Also, the cGAP achieves an acceleration of the algorithm that is directly proportional to the number of PEs. A degradation of the throughput due to the memory access collision is negligible for the arrays with more PEs. Acceleration figures for the problem instances analysed have shown a speedup of the cGAP with 5×5 PEs over $21 \times$ and $2086 \times$ when compared, respectively, to a PC and the MicroBlaze soft-core processor, both running a panmictic GA.

Regarding the MEB problem, we have ported to our cGAP an existing algorithm with two versions for a local search procedure in the GA: 1-shrink and 2-shrink. These algorithms, which manipulate tree data structures, have been successfully implemented in the PE using the HLS flow. We have focused in achieving the best quality algorithm, both in execution time and quality of the final solution obtained, by replicating, with a few changes, the original algorithms. Arrays with a maximum of 5×5 PEs and

4×4 PEs have been implemented in the target Virtex-6 FPGA for respectively the 1-shrink and 2-shrink versions (cGAP-1s and cGAP-2s). For the instances analysed, the cGAP-1s achieves an average acceleration over $15.4\times$ when compared to the original algorithm, while maintaining similar quality solutions. The cGAP-2s obtains an average acceleration over $2.2\times$, but it finds better quality solutions.

Chapter 7

Concluding remarks

The main goal of this work was to study and develop high performance custom computing machines that support the execution of genetic algorithms (GAs). We have conceived a scalable processor array, which implements a cellular genetic algorithm (cGA), that parallelizes the operations of the algorithm so that its execution time is improved. By distributing the population of the cGA over several independent memories, we do not compromise the memory accesses as the parallelism increases since more (and smaller) memories are added to the architecture. Moreover, a high-level synthesis (HLS) based design flow has been proposed to describe the problem-specific operations of the algorithm, thus easing the design of the proposed architecture to solve different optimization problems.

During the evolutionary process of a GA, the candidate solutions of a problem (the population) must interact among them by using genetics-inspired operators. In a dedicated hardware implementation of a GA, these operators can be parallelized so that several new solutions are generated at the same time. However, and as discussed in Chapter 2, this can lead to bottlenecks while accessing to the memory that keeps the population, with negative impact on the performance of the hardware. Nevertheless, it is possible to structure the population of a GA so that a given solution only interacts (till a certain point) with a given subset of solutions, as it happens with a cellular GA where the solutions are distributed over a regular grid and a solution only interacts with solutions placed in a given neighbourhood. By doing so, the population can be distributed over several independent memories that are accessed by processing nodes that implement the GA. Although a cGA has a great potential to be implemented in dedicated hardware, this has not been an active subject of research.

Additionally, GAs are known for using few and simple operators, where solutions are encoded by a binary sequence. However, this not always holds true and often different

encoding schemes are required to represent a valid solution (e.g. a graph). Moreover, if specific constraints are applied to the problem, operators must be added to deal with this. Also, the evaluation of a solution quality is always problem dependent. Therefore, dedicated hardware solutions that offer a set of the best known GA operators as templates to solve generally any optimization problem, only succeed in a restricted set of problems.

With this in mind, our work focused on building a dedicated hardware architecture that supports the execution of cGAs, while being able to specify the problem-dependent operations so that different problems could be solved. Moreover, the target hardware technology is field-programmable gate arrays (FPGAs) devices, which currently present a significant amount of distributed on-chip memory, which can be used to keep subsets of the population (a subpopulation) that are independently accessed by processing units, thus accelerating the execution of the algorithm.

The overall architecture proposed in this work is built by replicating processing elements (PEs) and subpopulation memories that are conveniently connected, as described in Chapter 3. Each subpopulation memory holds a subset of the cGA's solutions and is shared between two PEs, whereas a PE accesses to four subpopulation memories so that it evolves the algorithm with that solutions. Moreover, the architecture can be scaled by changing the number and configuration of the PEs in the array, and by adapting the number of solutions in each subpopulation memory to achieve a given population size.

This architecture supports the execution of cGAs since a solution can only interact with a predefined set of solutions, and the solutions' information is naturally diffused to the entire population without an explicit operator for doing so. This cGA is asynchronous since each PE generates and updates a solution to the subpopulation memories as soon as it is computed, and there is no synchronization in the update step among all the solutions. In addition, the order in which each solution is updated is given by the time that a PE takes to perform such operation. Therefore, we have classified our cGA as asynchronous with a time-driven update policy.

We have specified, in Chapter 4, a complete hardware infrastructure that supports the execution of the cGA in the proposed architecture. This complete solution, called cellular genetic algorithm processor (cGAP), is comprised, besides the array of processor and subpopulation memories, of a controller module named cellular genetic algorithm controller (cGAC), the interface of the cGAP, a communication infrastructure that supports the change of commands among the cGAC and the PEs, and a global random number generator (RNG) that feeds all the PEs with the necessary random numbers.

To develop the cGAP we have presented a HLS-based design flow where the problem-specific operations of the algorithm are specified in C++ and translated to hardware by using commercial HLS tools as described in Chapter 5. Therefore, the PE and the cGAC, which are the blocks that need to be changed according to the optimization problem, are rapidly customized with HLS to the problem's requirements so that they can be integrated with the remaining infrastructures to generate the complete cGAP with the desired level of parallelism. We have developed C++ classes and defined algorithms templates that can be used when building these two blocks and ensure a correct interface with the remaining hardware blocks.

Two distinct real-world problems that appear in the context of wireless ad hoc networks have been addressed in Chapter 6 as case studies to demonstrate the proposed architecture and design methodology. These problems, the spectrum allocation (SA) and the minimum energy broadcast (MEB), pose several implementation challenges when solved with GAs like problem-specific constraints, special representation for the solutions, or dedicated local search heuristics to improve the solutions.

We have seen that when describing the PE and cGAC in C++ for HLS, it is crucial to structure conveniently the code so that the tool is able to infer the desired hardware structures. Furthermore, the memory data organization of the arrays needed by the algorithm, which are mapped to hardware memories, must be organized in a way that efficient structures can be implemented. With this approach the HLS tool can then be guided by applying design constraints to implement the architecture. It is clear that although HLS raises the abstraction level when describing hardware, it does not provide automated mechanisms to build efficient hardware structures without a careful coding style.

Results have shown that the cGAP achieves an acceleration that is directly proportional to the level of parallelism, i.e., the number of PEs. Although memory access conflicts to the subpopulation memories happen, which becomes more critical as the number of solutions per subpopulation decreases, the performance degradation due to this is almost negligible. Therefore, even though cGAPs with more PEs require a smaller number of solutions per subpopulation (not to increase the population size), the performance of the engine is not degraded.

We have observed that the change in the level of parallelism in the cGAP does not compromise the quality of the solutions found by the algorithm. Indeed, by altering the number of PEs we change the number of solutions per subpopulation which results in changes in the neighbourhood of the solutions of the cGA, which affects the convergence rate of the algorithm as we have observed from the architecture simulation performed in Chapter 3. The convergence rate of a cGA is related with the difficulty (the distance)

that a solution has to spread its information throughout the array, and in configurations where this difficulty is increased (smaller subpopulations and higher number of PEs) lead to a slower convergence rate at the beginning of the algorithm while producing better quality results as the number of generations grows. Nevertheless, it is clear that a higher number of PEs, produces better results as it achieves an increased acceleration of the algorithm.

Besides the neighbourhood of the solutions in a cGA, the aspect ratio of the processor array or its toroidal or non-toroidal configuration, also change the way the solutions spread their information with the remaining solutions. For the same number of PEs, narrower arrays by comparison with square arrays, and non-toroidal compared with toroidal arrays, lead to an increased distance when a solution spreads its information throughout the population. Therefore, the same convergence rate behaviour is observed as the one found for arrays with more PEs. However, in these cases the cGAP does not have an acceleration gain as the same number of PEs exist for the different cases. Since we do not have a clear acceleration due to an increase in the number of PEs, we cannot say which of these configurations is the best, since it depends when the algorithm stops its evolutionary process.

Nonetheless, a non-toroidal implementation of the cGAP leads to a slightly smaller number of logic required to implement the architecture, although it requires more subpopulation memories. In addition, this configuration is interesting since no long data paths are required to build the toroid, which can be an advantage for a hardware implementation.

The architecture of the cGAC has proven to be effective for monitoring globally the processor array. It should be emphasized that this block is not necessary to ensure the implementation of the cellular genetic algorithm, since the architecture of the PEs and subpopulation memories are by themselves sufficient to implement the algorithm. However, with few commands sent among the cGAC to each PE and vice versa, it is possible to control the algorithm as required, which can be used, for instance, to stop the PEs based on a global stop criterion or to retrieve the best solution at the end of the algorithm. These functionalities are essential for the cGAP as a complete solution that supports the execution of a cGA while providing effective mechanisms to configure the PEs and monitor the algorithm evolution.

The proposed global RNG infrastructure aims to feed all the PEs with random numbers while using the minimum possible of hardware resources. Although a correlation among random numbers acquired by the different PEs may exist, for the examples analysed this has not shown to affect negatively the quality of the results obtained by the cGAP,

particularly when the parallelism is increased. Therefore, this infrastructure shows evidences that it can provide good results for generating random numbers that are required by the genetics-inspired operations of the algorithm.

The HLS-based design flow proposed in this work has demonstrated to be effective as a means to rapidly specify new functionalities so that the cGAP can be used to solve different optimization problems. Moreover, the two problems used in this work show that applying a GA to solve them requires customized operations so that the algorithm produces valid solutions. Additionally, in the MEB problem, a local search heuristic is introduced to improve the solution's quality. Therefore, our approach of customizing all the operations executed by the PE and cGAC so that it fulfils the problem's requirements allows addressing any optimization problem without being restricted by a set of generic operators supported by the architecture. Additionally, since we do not constrain the operations of the GA in the PE, the cGAP can be used to support other population-based metaheuristics that may take advantage of such architecture.

Moreover, the implementations of the cGAP in the Xilinx Virtex-6 FPGA have targeted cGAs with realist population sizes. The level of parallelism achieved by the implementations was constrained by the amount of hardware resources required to implement the PEs, and not by the memory. Therefore, there is space to increase further the dimension of the population, which can result in higher number of solutions in the population, or solving problems that require more memory space to represent a solution. Additionally and as the technology evolves, new families of FPGAs arrive to the market with increased hardware resources. Therefore, the cGAP can be implemented in such devices with an increased level of parallelism to accelerate the algorithm further. This means that our architecture can be used in the coming years while taking advantage of the level of integration that those devices will provide.

Nonetheless, we do not claim that the proposed architecture is the best in all circumstances and for all the problems. Parallelizing the operations of a GA to increase its performance can be simply achieved by having several parallel processing units that access a single memory where the solutions are kept. However, for a high level of parallelism these memory accesses are difficult to manage. In our architecture, that supports cellular genetic algorithms, this issue does not happen and the parallelism can be increased without bottlenecks while accessing to the solutions. Additionally, the cGAP requires that all the population of the algorithm is distributed in the internal memory of the FPGA. If these data or any other data required for the algorithm cannot be accommodated in the on-chip memory, the current cGAP architecture may not be employed.

To conclude, the cGAP and its design methodology provide an effective framework so that different optimization problems are solved by the metaheuristic, where different levels of parallelism can be adjusted to fulfil the requirements, mainly the trade-off between the acceleration of the algorithm and the hardware resources used to implement it. Moreover, the architecture can be exploited to support other population-based metaheuristics by specifying proper PEs, or even to study different approaches to population-based metaheuristics that can combine heterogeneous PEs, for example, PEs that execute GAs while others execute a local search procedure or simulated annealing.

7.1 Recommendations for future work

Based on the presented work, in the following items we highlight several research directions for future work.

- One feature of this architecture is the distribution of solutions in various memories. The implementation done in this work used an uniform distribution of the solutions among the memories local to a PE, thus causing an uniform propagation of the genetic information throughout the array. By distributing the generated solutions in a PE to the different memories according to their fitness value, it is possible to structure the population during the evolution of the algorithm so that the solutions are organized in the array according to their fitness values. For example, this can be accomplished by assigning different probabilities to the 4 subpopulation memories connected to a PE to receive a new solution that improves (or not) the average fitness value of all the solutions accessed by the PE. Therefore, the best solutions will be concentrated on a given set of PEs, whereas the worst ones on other (opposite) set of PEs. This type of cGA is known as hierarchical cGA and it has been proved to improve the quality of the metaheuristic [JADM06].
- Exploit the partial reconfiguration capability of the FPGAs to change the functionality of the PEs in run-time. Study the benefits of having changes in the operators of the PEs during the evolution of the algorithm so that the metaheuristic improves the solution found. For example, change a set of PEs near the end of the evolutionary process so that they execute an intensive local search heuristic. Therefore, those PEs first execute a GA, like the remaining PEs, and then a local search on solutions that have been evolved. The partial reconfiguration of FPGAs allows saving hardware resources since these are shared in time.
- Although the cGAP has been conceived to be implemented in a single FPGA device, it can be extended to implementations in multi FPGA devices, thus allowing

a higher level of parallelism. The cellular structure of the cellular genetic algorithm array (the cGAA) can be divided so that each part is assigned to an FPGA. Therefore, each FPGA supports the execution of a cGA, as it was presented in this work, while the different FPGAs are interconnected among them to form the complete array of processors. Each FPGA evolves independently its solutions, and occasionally exchanges solutions with neighbour FPGAs, resembling thus a distributed GA. This model of GA is known as cellular distributed GA [LA11].

- Combine a GA in a CPU and FPGA to solve problems that exceed the memory capacity of an FPGA. By doing so, it is possible for the cGAP to evolve a subset of the population, while the CPU keeps the remaining. Occasionally, the CPU sends new solutions to the cGAP and receives new ones.

References

- [AD08] Enrique Alba and Bernabé Dorronsoro. *Cellular genetic algorithms*, volume 42. Springer, 2008.
- [ADGT06] Enrique Alba, Bernabé Dorronsoro, Mario Giacobini, and Marco Tomassini. Decentralized cellular evolutionary algorithms. In J. Fagerberg, D.C. Mowery, and R.R. Nelson, editors, *Handbook of Bioinspired Algorithms and Applications*. CRC Press, 2006.
- [Ale05] S. Alexander. On the history of combinatorial optimization (till 1960). *Handbooks in Operations Research and Management Science: Discrete Optimization*, 12:1, 2005.
- [ARM] ARM. AMBA specifications. <http://www.arm.com/products/system-ip/amba-specifications.php>.
- [AT02] Enrique Alba and José M Troya. Improving flexibility and efficiency by adding parallelism to genetic algorithms. *Statistics and Computing*, 12(2):91–114, 2002.
- [ATAH11] Fariborz Ahmadi, Reza Tati, Soraia Ahmadi, and Veria Hossaini. New hardware engine for genetic algorithms. In *International Conference on Genetic and Evolutionary Computing.*, pages 122–126. IEEE, 2011.
- [ATDA05] Enrique Alba Torres, Bernabé Dorronsoro, and Hugo Alfonso. Cellular memetic algorithms evaluated on SAT. In *XI Congreso Argentino de Ciencias de la Computación*, 2005.
- [BDT99] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm intelligence: from natural to artificial systems*, volume 4. Oxford University Press, 1999.
- [BFM97] Thomas Back, David B. Fogel, and Zbigniew Michalewicz. *Handbook of evolutionary computation*. Oxford University Press, 1997.

- [BR03] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys (CSUR)*, 35(3):268–308, 2003.
- [Cal] Calypto. Catapult: Product family overview. <http://calypto.com/en/products/catapult/overview/>.
- [CC00] Yun-Ho Choi and Duck Jin Chung. VLSI processor of parallel genetic algorithm. In *Proceedings of the Second IEEE Asia Pacific Conference on ASICs.*, pages 143–146. IEEE, 2000.
- [CCP⁺01] Andrea E.F. Clementi, Pilu Crescenzi, Paolo Penna, Gianluca Rossi, and Paola Vocca. On the complexity of computing minimum energy consumption broadcast subgraphs. In *Proceedings of the 18th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 121–131. Springer, 2001.
- [ČHE02] Mario Čagalj, Jean-Pierre Hubaux, and Christian Enz. Minimum-energy broadcast in all-wireless networks: NP-completeness and distribution issues. In *Proceedings of the 8th annual international conference on Mobile computing and networking*, pages 172–182. ACM, 2002.
- [CP02] Erick Cantú-Paz. On random numbers and the performance of genetic algorithms. In *GECCO*, pages 311–318, 2002.
- [CW11] Yajuan Chen and Qinghai Wu. Design and implementation of PID controller based on FPGA and genetic algorithm. In *International Conference on Electronics and Optoelectronics.*, volume 4, pages 308–311. IEEE, 2011.
- [DDT08] K. M. Deliparaschos, G. C. Doyamis, and S. G. Tzafestas. A parameterised genetic algorithm IP core: FPGA design, implementation and performance evaluation. *International Journal of Electronics*, 95(11):1149–1166, 2008.
- [DMES⁺03] Arindam Kumar Das, Robert Jackson Marks, Mohamed El-Sharkawi, Payman Arabshahi, Andrew Gray, et al. *r-shrink*: A heuristic for improving minimum power broadcast trees in wireless networks. In *Global Telecommunications Conference.*, volume 1, pages 523–527. IEEE, 2003.
- [DOCQ93] Lawrence Davis, David Orvosh, Anthony Cox, and Yuping Qiu. A genetic algorithm for survivable network design. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 408–415, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

- [EPDP09] Charalampos Effraimidis, Kyprianos Papadimitriou, Apostolos Dollas, and Ioannis Papaefstathiou. A self-reconfiguring architecture supporting multiple objective functions in genetic algorithms. In *International Conference on Field Programmable Logic and Applications.*, pages 453–456. IEEE, 2009.
- [FKK⁺10] Pradeep R. Fernando, Srinivas Katkoori, Didier Keymeulen, Ricardo Zebulum, and Adrian Stoica. Customizable FPGA IP core implementation of a general-purpose genetic algorithm engine. *IEEE Transactions on Evolutionary Computation.*, 14(1):133–149, 2010.
- [Fra57] Alex S. Fraser. Simulation of genetic systems by automatic digital computers. i. introduction. *Australian Journal of Biological Sciences*, 10(4):484–491, 1957.
- [GL⁺97] Fred Glover, Manuel Laguna, et al. *Tabu search*, volume 22. Springer, 1997.
- [GN95] Paul Graham and Brent Nelson. A hardware genetic algorithm for the traveling salesman problem on splash 2. In *Field-Programmable Logic and Applications*, pages 352–361. Springer, 1995.
- [Gra] Mentor Graphics. Precision RTL. http://www.mentor.com/products/fpga/synthesis/precision_rtl/.
- [GTL13] Liucheng Guo, David B. Thomas, and Wayne Luk. Customisable architectures for the set covering problem. *SIGARCH Computer Architecture News*, 41(5):101–106, 2013.
- [GTL14] Liucheng Guo, David B. Thomas, and Wayne Luk. Automated framework for general-purpose genetic algorithms in FPGAs. In *Applications of Evolutionary Computation*, pages 714–725. Springer, 2014.
- [GVK04] John C. Gallagher, Saranyan Vighram, and Gregory Kramer. A family of compact genetic algorithms for intrinsic evolvable hardware. *IEEE Transactions on Evolutionary Computation*, 8(2):111–126, 2004.
- [HCF12] Ping-Che Hsiao, Tsung-Che Chiang, and Li-Chen Fu. Particle swarm optimization for the minimum energy broadcast problem in wireless ad-hoc networks. In *Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE, 2012.
- [HGLL06] Gregory S. Hornby, Al Globus, Derek S. Linden, and Jason D. Lohn. Automated antenna design with evolutionary algorithms. In *Proc. AIAA Space Conference*, page 8, 2006.

- [HLG99] Georges R. Harik, Fernando G. Lobo, and David E. Goldberg. The compact genetic algorithm. *IEEE Transactions on Evolutionary Computation*, 3(4):287–297, 1999.
- [HM09] Ali B. Hashemi and Mohammad Reza Meybodi. Cellular PSO: A PSO for dynamic environments. In *Advances in computation and intelligence*, pages 422–433. Springer, 2009.
- [Hol75] John H. Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence*. University Michigan Press, 1975.
- [HT11] Pauline C. Haddow and Andy M. Tyrrell. Challenges of evolvable hardware: past, present and the path to a promising future. *Genetic Programming and Evolvable Machines*, 12(3):183–215, 2011.
- [Ini] Accellera Systems Initiative. SystemC. <http://www.accellera.org/downloads/standards/systemc>.
- [JADM06] Stefan Janson, Enrique Alba, Bernabé Dorronsoro, and Martin Middendorf. Hierarchical cellular genetic algorithm. In *Evolutionary Computation in Combinatorial Optimization*, pages 111–122. Springer, 2006.
- [Jar12] Jiri Jaros. Multi-GPU island-based genetic algorithm for solving the knapsack problem. In *Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE, 2012.
- [JC08] Yutana Jewajinda and Prabhas Chongstitvatana. FPGA implementation of a cellular compact genetic algorithm. In *NASA/ESA Conference on Adaptive Hardware and Systems.*, pages 385–390. IEEE, 2008.
- [JKFA06] Mehdi Salmani Jelodar, Mehdi Kamal, Sied Mehdi Fakhraie, and Majid Nili Ahmadabadi. SOPC-based parallel genetic algorithm. In *IEEE Congress on Evolutionary Computation.*, pages 2800–2806. IEEE, 2006.
- [KAM⁺99] Osamu Kitaura, Hideaki Asada, Motoaki Matsuzaki, Takamitsu Kawai, Hideki Ando, and Toshio Shimada. A custom computing machine for genetic algorithms without pipeline stalls. In *IEEE International Conference on Systems, Man, and Cybernetics.*, volume 5, pages 577–584. IEEE, 1999.
- [KE95] J. Kennedy and R. C. Eberhart. Particle swarm optimization. In *Proceedings of IEEE International Conference on Neural Networks*, pages 1942–1948, 1995.

- [KES01] Mohan Krishnamoorthy, Andreas T. Ernst, and Yazid M. Sharaiha. Comparison of algorithms for the degree constrained minimum spanning tree. *Journal of heuristics*, 7(6):587–611, 2001.
- [KJV83] Scott Kirkpatrick, D. Gelatt Jr., and Mario P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.
- [KP04] Intae Kang and Radha Poovendran. Broadcast with heterogeneous node capability [wireless ad hoc or sensor networks]. In *Global Telecommunications Conference*, volume 6, pages 4114–4119. IEEE, 2004.
- [LA11] Gabriel Luque and Enrique Alba. *Parallel Genetic Algorithms: Theory and Real World Applications*, volume 367. Springer, 2011.
- [L'E99] Pierre L'Ecuyer. Tables of maximally equidistributed combined LFSR generators. *Mathematics of Computation of the American Mathematical Society*, 68(225):261–269, 1999.
- [LKM⁺99] Pedro Larranaga, Cindy M. H. Kuijpers, Roberto H. Murga, Inaki Inza, and Sejla Dizdarevic. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial Intelligence Review*, 13(2):129–170, 1999.
- [LLHD05] Giampaolo L. Libralao, Telma W. Lima, Karen Honda, and Alexandre CB Delbem. Node-depth encoding for directed graphs. In *The 2005 IEEE Congress on Evolutionary Computation.*, volume 3, pages 2196–2201. IEEE, 2005.
- [Mar95] George Marsaglia. DIEHARD. <http://www.stat.fsu.edu/pub/diehard/>, 1995.
- [MB97] Graham M. Megson and Ian M. Bland. Generic systolic array for genetic algorithms. *IEE Proceedings - Computers and Digital Techniques*, 144(2):107–119, 1997.
- [MB98] G.M. Megson and I.M. Bland. Synthesis of a systolic array genetic algorithm. In *International Parallel Processing Symposium*, pages 316–320. IEEE Computer Society, 1998.
- [MBM11] Ujjwal Maulik, Sanghamitra Bandyopadhyay, and Anirban Mukhopadhyay. *Multiobjective Genetic Algorithms for Clustering: Applications in Data Mining and Bioinformatics*. Springer, 2011.

- [MF99] Mark M. Meysenburg and James A. Foster. Randomness and GA performance, revisited. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 425–432, 1999.
- [MGD05] Roberto Montemanni, Luca Maria Gambardella, and Arindam Kumar Das. The minimum power broadcast problem in wireless networks: a simulated annealing approach. In *Wireless Communications and Networking Conference*, volume 4, pages 2057–2062. IEEE, 2005.
- [MGSK88] Heinz Mühlenbein, Martina Gorges-Schleuter, and Ottmar Krämer. Evolution algorithms in combinatorial optimization. *Parallel Computing*, 7(1):65–85, 1988.
- [MK94] Makoto Matsumoto and Yoshiharu Kurita. Twisted GFSR generators II. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 4(3):254–266, 1994.
- [MN98] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.
- [Mos89] Pablo Moscato. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. *Caltech concurrent computation program, C3P Report*, 826, 1989.
- [MVBG⁺12] Wim Meeus, Kristof Van Beeck, Toon Goedemé, Jan Meel, and Dirk Stroobandt. An overview of today’s high-level synthesis tools. *Design Automation for Embedded Systems*, 16(3):31–51, 2012.
- [NBKHM13] Vishnu P. Nambiar, Sathivellu Balakrishnan, Mohamed Khalil-Hani, and Muhammad N. Marsono. HW/SW co-design of reconfigurable hardware-based genetic algorithm in FPGAs applicable to a variety of problems. *Computing*, 95(9):863–896, 2013.
- [NdMM07] Nadia Nedjah and Luiza de Macedo Mourelle. An efficient problem-independent hardware implementation of genetic algorithms. *Neurocomputing*, 71(1):88–94, 2007.
- [NHM11] Vahid Noroozi, Ali B. Hashemi, and Mohammad Reza Meybodi. Cellularde: a cellular based differential evolution for dynamic optimization problems. In *Adaptive and natural computing algorithms*, pages 340–349. Springer, 2011.

- [NI10] Nasimul Noman and Hitoshi Iba. Cellular differential evolution algorithm. In *AI 2010: Advances in Artificial Intelligence*, pages 293–302. Springer, 2010.
- [NYYY07] Hung Dinh Nguyen, Ikuo Yoshihara, Kunihito Yamamori, and Moritoshi Yasunaga. Implementation of an effective hybrid GA for large-scale traveling salesman problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 37(1):92–99, 2007.
- [PD05] Malay K. Pakhira and Rajat K. De. A hardware pipeline for function optimization using genetic algorithms. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 949–956. ACM, 2005.
- [PJS10] Petr Pospíchal, Jiri Jaros, and Josef Schwarz. Parallel genetic algorithm on the CUDA architecture. In *Applications of Evolutionary Computation*, pages 442–451. Springer, 2010.
- [PK94] Charles C. Palmer and Aaron Kershenbaum. Representing trees in genetic algorithms. In *Proceedings of the First IEEE Conference on Evolutionary Computation.*, pages 379–384. IEEE, 1994.
- [PK14] Riccardo Poli and John Koza. *Genetic Programming*. Springer, 2014.
- [PSJ10] Petr Pospíchal, Josef Schwarz, and Jiri Jaros. Parallel genetic algorithm solving 0/1 knapsack problem running on the GPU. In *16th International Conference on Soft Computing MENDEL*, volume 2010, pages 64–70, 2010.
- [PZZ06] Chunyi Peng, Haitao Zheng, and Ben Y. Zhao. Utilization and fairness in spectrum assignment for opportunistic spectrum access. *Mobile Networks and Applications*, 11(4):555–576, 2006.
- [RGH02] Franz Rothlauf, David E. Goldberg, and Armin Heinzl. Network random keys - a tree representation scheme for genetic and evolutionary algorithms. *Evolutionary Computation*, 10(1):75–97, 2002.
- [Rot06] Franz Rothlauf. *Representations for genetic and evolutionary algorithms*. Springer, 2006.
- [SB11] Alok Singh and Wilson Naik Bhukya. A hybrid genetic algorithm for the minimum energy broadcast problem in wireless ad hoc networks. *Applied Soft Computing*, 11(1):667–674, 2011.

- [SDJ96] Jayshree Sarma and Kenneth De Jong. An analysis of the effects of neighborhood size and shape on local selection algorithms. In *Parallel Problem Solving From Nature - PPSN IV*, pages 236–244. Springer, 1996.
- [SdR99] Birgitt Schönfisch and André de Roos. Synchronous and asynchronous updating in cellular automata. *BioSystems*, 51(3):123–143, 1999.
- [SF02] Iouliia Skliarova and António B. Ferrari. FPGA-based implementation of genetic algorithm for the traveling salesman problem and its industrial application. In Tim Hendtlass and Moonis Ali, editors, *Developments in Applied Artificial Intelligence*, volume 2358 of *Lecture Notes in Computer Science*, pages 77–87. Springer Berlin Heidelberg, 2002.
- [SGK05] Kumara Sastry, David Goldberg, and Graham Kendall. Genetic algorithms. In *Search Methodologies*, pages 97–125. Springer, 2005.
- [SLGZ11] Yang Shi, Hongcheng Liu, Liang Gao, and Guohui Zhang. Cellular particle swarm optimization. *Information Sciences*, 181(20):4460–4493, 2011.
- [SP97] Rainer Storn and Kenneth Price. Differential evolution—a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359, 1997.
- [SS12] A. Swarnalatha and A.P. Shanthi. Optimization of single variable functions using complete hardware evolution. *Applied Soft Computing*, 12(4):1322–1329, 2012.
- [SSC⁺01] Barry Shackleford, Greg Snider, Richard J. Carter, Etsuko Okushi, Mitsuhiro Yasuda, Katsuhiko Seo, and Hiroto Yasuura. A high-performance, pipelined, FPGA-based genetic algorithm machine. *Genetic Programming and Evolvable Machines*, 2(1):33–60, 2001.
- [SSS95] Stephen D. Scott, Ashok Samal, and Shared Seth. HGA: a hardware-based genetic algorithm. In *International symposium on Field-programmable gate arrays*, pages 53–59. ACM, 1995.
- [STCS02] Barry Shackleford, Motoo Tanaka, Richard J. Carter, and Greg Snider. FPGA implementation of neighborhood-of-four cellular automata random number generators. In *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 106–112. ACM, 2002.

- [SWSS95] Nathan Sitkoff, Mike Wazlowski, Aaron Smith, and Harvey Silverman. Implementing a genetic algorithm on a parallel custom computing machine. In *IEEE Symposium on FPGAs for Custom Computing Machines.*, pages 180–187. IEEE, 1995.
- [TA95] Brian C.H. Turton and Tughrul Arslan. A parallel genetic VLSI architecture for combinatorial real-time applications-disc scheduling. In *International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications*, pages 493–498. IET, 1995.
- [TAH94] Brian C.H. Turton, Tughrul Arslan, and David H. Horrocks. A hardware architecture for a parallel genetic algorithm for image registration. In *IEE Colloquium on Genetic Algorithms in Image Processing and Vision*, pages 11/1–11/6. IET, 1994.
- [TÄW11] Ville Tirronen, Sami Äyrämö, and Matthieu Weber. Study on the effects of pseudorandom generation quality on the performance of differential evolution. In *Adaptive and Natural Computing Algorithms*, pages 361–370. Springer, 2011.
- [THC11] Ching-Chih Tsai, Hsu-Chih Huang, and Cheng-Kai Chan. Parallel elite genetic algorithm and its application to global path planning for autonomous robot navigation. *IEEE Transactions on Industrial Electronics.*, 58(10):4813–4821, 2011.
- [TMS⁺06] Tatsuhiro Tachibana, Yoshihiro Murata, Naoki Shibata, Keiichi Yasumoto, and Minoru Ito. General architecture for hardware implementation of genetic algorithm. In *IEEE Symposium on Field-Programmable Custom Computing Machines.*, pages 291–292. IEEE, 2006.
- [TSP] TSPLIB. <http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/>.
- [Tur50] Alan M. Turing. Computing machinery and intelligence. *Mind*, pages 433–460, 1950.
- [TY04] Wallace Tang and Leslie Yip. Hardware implementation of genetic algorithms using FPGA. In *MWSCAS: Midwest symposium on circuits and systems*, volume 1, pages 549–52. IEEE, 2004.
- [TZFS13] Elias Z. Tragos, Sherali Zeadally, Alexandros G. Fragkiadakis, and Vasilios A. Siris. Spectrum assignment in cognitive radio networks: A comprehensive survey. *IEEE Communications Surveys and Tutorials*, 15(3):1108–1135, 2013.

- [VA10] Pablo Vidal and Enrique Alba. A multi-GPU implementation of a cellular genetic algorithm. In *Congress on Evolutionary Computation*, pages 1–7. IEEE, 2010.
- [VLMT10] Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. GPU-based island model for evolutionary algorithms. In *Proceedings of the 12th annual Conference on Genetic and Evolutionary Computation*, pages 1089–1096, 2010.
- [VPP09] Michalis Vavouras, Kyprianos Papadimitriou, and Ioannis Papaefstathiou. High-speed FPGA-based implementations of a genetic algorithm. In *International Symposium on Systems, Architectures, Modeling, and Simulation.*, pages 9–16. IEEE, 2009.
- [VRGGAR⁺05] Miguel A. Vega-Rodriguez, Raul Gutierrez-Gil, Jose M. Avila-Roman, Juan Manuel Sanchez-Perez, and Juan A. Gomez-Pulido. Genetic algorithms using parallelism and FPGAs: the TSP as case study. In *International Conference Workshops on Parallel Processing.*, pages 573–579. IEEE, 2005.
- [WNE00] Jeffrey E. Wieselthier, Gam D. Nguyen, and Anthony Ephremides. On the construction of energy-efficient broadcast and multicast trees in wireless networks. In *Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 2, pages 585–594. IEEE, 2000.
- [WS12] Kai Wang and Zhen Shen. A GPU-based parallel genetic algorithm for generating daily activity plans. *IEEE Transactions on Intelligent Transportation Systems*, 13(3):1474–1480, 2012.
- [XAD⁺08] Fatos Xhafa, Enrique Alba, Bernabé Dorronsoro, Bernat Duran, and Ajith Abraham. Efficient batch job scheduling in grids using cellular memetic algorithms. In *Metaheuristics for Scheduling in Distributed Computing Environments*, pages 273–299. Springer, 2008.
- [Xila] Xilinx. ISE design suite. <http://www.xilinx.com/products/design-tools/ise-design-suite.html>.
- [Xilb] Xilinx. MicroBlaze soft processor core. <http://www.xilinx.com/products/design-tools/microblaze.html>.
- [Xilc] Xilinx. Petalinux. <http://www.wiki.xilinx.com/PetaLinux>.

-
- [Xild] Xilinx. Ultrascale architecture and product overview. http://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf.
- [Xile] Xilinx. Virtex-6 family overview. http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf.
- [Xilf] Xilinx. Virtex-6 FPGA ML605 evaluation kit. <http://www.xilinx.com/products/boards-and-kits/ek-v6-ml605-g.html>.
- [Xilg] Xilinx. Xilinx github. <https://github.com/xilinx>.
- [Xilh] Xilinx. Xilinx open source linux. <http://www.wiki.xilinx.com/Open+Source+Linux>.
- [ZMC07a] Zhenhuan Zhu, David Mulvaney, and Vassilios Chouliaras. A novel genetic algorithm designed for hardware implementation. *International Journal of Computational Intelligence*, 3(4):281–288, 2007.
- [ZMC07b] Zhenhuan Zhu, David J. Mulvaney, and Vassilios A. Chouliaras. Hardware implementation of a novel genetic algorithm. *Neurocomputing*, 71(1):95–106, 2007.
- [ZPZS09] Zhijin Zhao, Zhen Peng, Shilian Zheng, and Junna Shang. Cognitive radio spectrum allocation using evolutionary algorithms. *IEEE Transactions on Wireless Communications*, 8(9):4421–4425, 2009.

Appendix A

C++ classes to describe the cGAP

This appendix presents the C++ classes used in Catapult HLS to describe the processing element (PE) and the cellular genetic controller (cGAC). The source code is presented as well as a description of the methods used by the classes.

A.1 Class `command_type_cGA`

The class `command_type_cGA`, used to manipulate a command data type in Catapult HLS, is presented in this section. This data type is used when describing the PE and cGAC to process the commands that are transferred among them using the communication infrastructure of the cellular genetic algorithm processor (cGAP).

Listings A.1 and A.2 show the C++ source code of the class and Table A.1 presents a description of its methods.

```
1  #ifndef _COMMAND_TYPE_CGA_H
2  #define _COMMAND_TYPE_CGA_H
3
4  #include "cGA_parameters.h"
5  #include "ac_int.h"
6  #include "log2ceil.h"
7
8  // member of the class (contain all the fields in commands used by the cGAP):
9  //   'data'           - 32-bit data
10 //   'command'        - command code
11 //   'cmd_all_PEs'    - broadcast bit
12 //   'coord_i'        - row of PE in cGAA
13 //   'coord_j'        - column of PE in cGAA
14 class command_type_cGA{
15 private:
16     ac_int<32,false>          data;          // LSB
17     ac_int<N_BIT_COMMAND_GA_PROC,false>    command;
```

```

18     bool cmd_all_PEs;
19     ac_int<LOG2_CEIL<ARRAY_DIM_I>::val,false> coord_i;
20     ac_int<LOG2_CEIL<ARRAY_DIM_J>::val,false> coord_j; // MSB
21 public:
22     command_type_cGA(){}
23     ac_int<N_BIT_COMMAND_GA_PROC,false> get_command();
24     ac_int<32,false> get_data();
25     ac_int<LOG2_CEIL<ARRAY_DIM_I>::val,false> get_coord_i();
26     ac_int<LOG2_CEIL<ARRAY_DIM_J>::val,false> get_coord_j();
27     void set_command(ac_int<N_BIT_COMMAND_GA_PROC,false> cmd);
28     void set_data(ac_int<32,false> data_value);
29     void set_cmd_all_PEs(bool cmd_all_PEs_value);
30     void set_coord_i(
31         ac_int<LOG2_CEIL<ARRAY_DIM_I>::val,false> coord_i_val);
32     void set_coord_j(
33         ac_int<LOG2_CEIL<ARRAY_DIM_J>::val,false> coord_j_val);
34 };
35
36 #endif

```

LISTING A.1: C++ code of the `command_type_cGA` class definition used in Catapult HLS (header file `command_type_cGA.h`).

```

1  #include "command_type_cGA.h"
2
3  ac_int<N_BIT_COMMAND_GA_PROC,false> command_type_cGA::get_command()
4  {
5      return command;
6  }
7
8  ac_int<32,false> command_type_cGA::get_data()
9  {
10     return data;
11 }
12
13 void command_type_cGA::set_command(ac_int<N_BIT_COMMAND_GA_PROC,false> cmd)
14 {
15     command = cmd;
16 }
17
18 void command_type_cGA::set_data(ac_int<32,false> data_value)
19 {
20     data = data_value;
21 }
22
23 void command_type_cGA::set_cmd_all_PEs(bool cmd_all_PEs_value)
24 {
25     cmd_all_PEs = cmd_all_PEs_value;
26 }
27
28 void command_type_cGA::
29     set_coord_i(ac_int<LOG2_CEIL<ARRAY_DIM_I>::val,false> coord_i_val)
30 {
31     coord_i = coord_i_val;
32 }
33
34 void command_type_cGA::
35     set_coord_j(ac_int<LOG2_CEIL<ARRAY_DIM_J>::val,false> coord_j_val)
36 {
37     coord_j = coord_j_val;
38 }
39
40 ac_int<LOG2_CEIL<ARRAY_DIM_I>::val,false> command_type_cGA::get_coord_i()
41 {
42     return coord_i;
43 }
44
45 ac_int<LOG2_CEIL<ARRAY_DIM_J>::val,false> command_type_cGA::get_coord_j()
46 {

```

```

47     return coord_j;
48 }

```

LISTING A.2: C++ code of the `command_type_cGA` class implementation used in Catapult HLS (file `command_type_cGA.cpp`).

TABLE A.1: Description of the C++ class `command_type_cGA` methods used in Catapult HLS.

<code>ac_int<N_BIT_COMMAND_GA_PROC, false> get_command()</code>	
description:	Gets command code. Used by cGAC or PEs.
parameters:	None.
return value:	Command code value.
<code>ac_int<32, false> get_data()</code>	
description:	Gets the data member data. This value and the command code constitute the payload to a PE or cGAC. Used by cGAC or PEs.
parameters:	None.
return value:	32-bit data value (used together with the command code).
<code>ac_int<LOG2_CEIL<ARRAY_DIM_I>::val, false> get_coord_i()</code>	
description:	Gets the data member <code>coord_i</code> , the row identification of the PE in the cGAA. Used by cGAC to identify the PE where the command comes from.
parameters:	None.
return value:	Row identification of the PE in the cGAA.
<code>ac_int<LOG2_CEIL<ARRAY_DIM_J>::val, false> get_coord_j()</code>	
description:	Gets the data member <code>coord_j</code> , the column identification of the PE in the cGAA. Used by cGAC to identify the PE where the command comes from.
parameters:	None.
return value:	Column identification of the PE in the cGAA.
<code>void set_command(ac_int<N_BIT_COMMAND_GA_PROC, false> cmd)</code>	
description:	Sets the command code. Used by cGAC and PEs.
parameters:	<code>cmd</code> : Command code.
return value:	None.
<code>void set_data(ac_int<32, false> data_value)</code>	
description:	Sets the data member data. Used by cGAC and PEs.
parameters:	<code>data_value</code> : 32-bit data value (used together with the command code).
return value:	None.
<code>void set_cmd_all_PEs(bool cmd_all_PEs_value)</code>	
description:	Sets the data member <code>cmd_all_PEs</code> . Used by cGAC to broadcast a command to all PEs.
parameters:	<code>cmd_all_PEs_value</code> : Boolean value to enable the broadcast of a command.
return value:	None.
<code>void set_coord_i(ac_int<LOG2_CEIL<ARRAY_DIM_I>::val, false> coord_i_var)</code>	
description:	Sets the data member <code>coord_i</code> . Used by cGAC to send a command to a specific PE in the cGAA.
parameters:	<code>coord_i_var</code> : Row identification of the PE in the cGAA.
return value:	None.
<code>void set_coord_j(ac_int<LOG2_CEIL<ARRAY_DIM_J>::val, false> coord_j_var)</code>	
description:	Sets the data member <code>coord_j</code> . Used by cGAC to send a command to a specific PE in the cGAA.
parameters:	<code>coord_j_var</code> : Column identification of the PE in the cGAA.
return value:	None.

A.2 Class `request_channel`

The class `request_channel` is presented in this section. This data type implements the interface that ensures the handshake mechanism between a PE and a subpopulation memory. Therefore, this class is used by a PE to request a access to a solution in the subpopulation.

Listings A.3 and A.4 show the C++ source code of the class and Table A.2 presents a description of its methods. Section 5.2.1.3 provides further details of the algorithm structure that must be followed to ensure the correct implementation of the access arbitration to the subpopulation memories.

```

1  #ifndef _REQUESTS_CHANNEL_H
2  #define _REQUESTS_CHANNEL_H
3
4  #include <ac_int.h>
5  #include <ac_channel.h>
6  #include "cGA_parameters.h"
7
8  struct request_type{
9      ac_int<3,false> command;    // LSBs
10     ac_int<N_BIT_MAX_SOL_PER_SUBPOP,false> solution; // MSBs
11 };
12
13 class request_channel{
14 private:
15     ac_channel<request_type> request; // request from PE to subpopulation mem.
16     ac_channel<bool> req_ack;        // acknowledge from subpopulation mem.
17 public:
18     request_channel(){}
19     bool send_request(ac_int<3,false> cmd,
20                     ac_int<N_BIT_MAX_SOL_PER_SUBPOP,false> sol);
21     void send_request(ac_int<3,false> cmd);
22 };
23
24 #endif

```

LISTING A.3: C++ code of the definition of the `request_channel` class used in Catapult HLS (header file *request_channel.h*).

```

1  #include "request_channel.h"
2
3  bool request_channel::send_request(ac_int<3,false> cmd,
4                                   ac_int<N_BIT_MAX_SOL_PER_SUBPOP,false> sol)
5  {
6      request_type new_request;
7
8      new_request.command = cmd;
9      new_request.solution = sol;
10     request.write(new_request); // send request
11     return req_ack.read();      // acknowledge
12 }
13
14 void request_channel::send_request(ac_int<3,false> cmd)
15 {
16     request_type new_request;
17
18     new_request.command = cmd;
19     request.write(new_request); // send request

```

```

20     req_ack.read();                // to avoid undesired schedules
21 }

```

LISTING A.4: C++ code of the implementation of the request_channel class used in Catapult HLS (file *request_channel.cpp*).

TABLE A.2: Description of the C++ class request_channel methods used in Catapult HLS.

<pre>bool send_request(ac_int<3,false> cmd, ac_int<N_BIT_MAX_SOL_PER_SUBPOP,false> sol)</pre>	
description:	Sends a request to the subpopulation memory to allocate a solution (a memory slot). Returns a boolean value if request has been accepted or not. This method should be used with the commands CMD_READ_SOL_1, CMD_READ_SOL_2 or CMD_WRITE depending on the desired access to the solution.
parameters:	cmd: Command code. sol: Solution number.
return value:	Boolean value if request command has been accepted or not by subpopulation memory.

<pre>void send_request(ac_int<3,false> cmd)</pre>	
description:	Sends a request to the subpopulation memory to release a solution (a memory slot). This method should be used with the commands CMD_RELEASE_SOL_1, CMD_RELEASE_SOL_2, CMD_RELEASE_WRITE or CMD_RELEASE_ALL for the solution to be released (last command releases all the allocated solutions). Additionally, the command CMD_START_SELECTION must be used to start a selection procedure. The request command to the subpopulation memory is always accepted.
parameters:	cmd: Command code.
return value:	None.

Appendix B

cGAP MicroBlaze access

This appendix presents the application programming interface (API) used by the MicroBlaze soft-core processor to access the cellular genetic algorithm processor (cGAP). Additionally, C code examples that use the API are provided that run in the processor to control the cGAP.

B.1 Application programming interface

As the MicroBlaze accesses to the cGAP as a memory mapped device using AXI4-Lite protocol, which is a register-style interface, we have defined a set of 32-bit registers in the interface circuits of the cGAP (c.f. Figure 5.5) that allow the communication from the processor to the cGAP. Table B.1 presents a list and description of these registers, where the number associated with each name's register is used by the processor as the offset address to access them.

Table B.1 presents a description of the C functions that constitute the API (defined in the file `ip_cGAP_intf.c` of the cGAP API library), and Listing B.1 provides their code implementation.

TABLE B.1: List of registers and their description used by the MicroBlaze to communicate with the cGAP.

Register	Description	Access
reg0[0]	Indicates if cGAC is processing a command sent by the host processor. Equal to 1 is processing, 0 otherwise.	Read
reg0[1]	Indicates if cGAA (any PE) is processing. Equal to 1 is processing, 0 otherwise.	Read
reg1	Issues a new command to the cGAC.	Write
reg2	General purpose register used when command is sent to cGAC (reg1). Carries any 32-bit additional data required by the command.	Write
reg3	Issues a read/write operation to the cGA dual-port memory. Memory address is defined from bit numbers 15 to 0 (MSB to LSB). Bit 16 defines operation: 1 for writing, 0 for reading.	Write
reg4	32-bit data read/written from the cGA dual-port memory. Used with reg3.	Read/Write

TABLE B.2: Description of the application programming interface C functions used by the MicroBlaze to access the cGAP as a memory mapped device.

void IP_cga_init()	description: Maps the cGAP into memory. parameters: None. return value: None.
void IP_cga_clean()	description: Deletes mapping of the cGAP into memory. parameters: None. return value: None.
void write_reg(int reg_num, int value)	description: Writes value to register in the cGAP. parameters: reg_num: Register number; value: Value (32 bits) to be written. return value: None.
int read_reg(int reg_num)	description: Reads value from register in the cGAP. parameters: reg_num: Register number. return value: Register value (32 bits).
void write_BRAM(int addr, int value)	description: Writes value to cGA dual-port memory. parameters: addr: memory address; value: Value (32 bits) to be written. return value: None.
int read_BRAM(int addr)	description: Reads value from cGA dual-port memory. parameters: addr: memory address. return value: Value (32 bits) read from memory.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <fcntl.h>
4 #include <unistd.h>
5 #include <sys/mman.h>
6 #include "ip_cga_intf.h"
7
8 #define MEM_BASE_ADDRESS    0x87000000
9 #define MAP_SIZE            0x00001000
10 #define MAP_MASK            (MAP_SIZE - 1)

```

```

11
12 static int memfd;
13 static off_t dev_base = MEM_BASE_ADDRESS; // address defined on EDK (Xilinx)
14 static void *mapped_base; // address returned by mmap
15 static void *mapped_dev_base; // address returned by mmap with
16 // page alignment correction
17
18 void IP_cga_init()
19 {
20     memfd = open("/dev/mem", O_RDWR | O_SYNC);
21     if (memfd == -1) {
22         fprintf(stderr, "Can't open /dev/mem.\n");
23         exit(0);
24     }
25
26     mapped_base = mmap(0, MAP_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, memfd,
27                       dev_base & ~MAP_MASK);
28
29     if (mapped_base == (void *) -1) {
30         fprintf(stderr, "Can't map the memory to user space.\n");
31         exit(0);
32     }
33
34     mapped_dev_base = mapped_base + (dev_base & MAP_MASK);
35 }
36
37 void IP_cga_clean()
38 {
39     if (munmap(mapped_base, MAP_SIZE) == -1) {
40         fprintf(stderr, "Can't unmap memory from user space.\n");
41         exit(0);
42     }
43     close(memfd);
44 }
45
46 void write_reg(int reg_num, int value)
47 {
48     *((volatile unsigned long *) (mapped_dev_base + 4*reg_num)) = value;
49 }
50
51 int read_reg(int reg_num)
52 {
53     return ( *((volatile unsigned int *) (mapped_dev_base + 4*reg_num)) );
54 }
55
56 void write_BRAM(int addr, int value)
57 {
58     int addr_2_IP;
59
60     addr_2_IP = addr & 0x0000ffff;
61     addr_2_IP = addr_2_IP | 0x00010000;
62
63     write_reg(4, value);
64     write_reg(3, addr_2_IP);
65 }
66
67 int read_BRAM(int addr)
68 {
69     int addr_2_IP;
70
71     addr_2_IP = addr & 0x0000ffff;
72
73     write_reg(3, addr_2_IP);
74     return read_reg(4);
75 }

```

LISTING B.1: C code of the functions that define the application programming interface used by the MicroBlaze to access the cGAP as a memory mapped device.

B.2 cGAP control

With the API that grants to the MicroBlaze the access to the cGAP hardware, new C functions can be built to control the cGAP and the algorithm supported by it. These functions, that are defined in the file `ip_cgac_control.c` in the design flow, must be adapted to fulfil the requirements of each optimization problem and how the processor controls the cGAP. Listing B.2 provides several C code examples of such functions.

```

1  #include <stdio.h>
2
3  #include "ip_cgac_control.h"
4  #include "ip_cgac_intf.h"
5
6  #define CMD_RESET_CGAP      0
7  #define CMD_START_PE        1
8  #define CMD_SET_PARAMETER   2
9
10
11 /* cga_cmd_ready:
12  *   Waits till cGAC can receive a new command.
13  */
14 void cga_cmd_ready()
15 {
16     int value;
17
18     // waits till cGAC can receive new command (reg0[0] = 0)
19     do {
20         value = read_reg(0);
21         value = value & 0x00000001;
22     } while (value == 0x00000001);    // reg0[0] = 1 -> busy
23 }
24
25
26 /* cga_PEs_stopped:
27  *   Waits till cGAA (all PEs) are stopped (idle state).
28  */
29 void cga_PEs_stopped()
30 {
31     int value;
32
33     // waits till all PEs stop (reg0[1] = 0)
34     do {
35         value = read_reg(0);
36         value = value & 0x00000002;
37     } while (value == 0x00000002);    // reg0[1] = 1 -> PEs running
38 }
39
40
41 /* cga_reset:
42  *   Sends reset command do the cGAP.
43  */
44 void cga_reset()
45 {
46     int value;
47     value = CMD_RESET_CGAP & 0x00007fff;
48
49     write_reg(1, value);
50 }
51
52
53 /* cga_start_all_PEs:
54  *   Command to start the genetic algorithm in all the PEs.
55  *   parameter: number of generations ('max_itr') to run in each PE.
56  *
57  *   This function shows how to send a command to the cGAC that

```

```

58 *   targets all the PEs:
59 *       1st: wait till (or check if) previous command has been processed by
60 *           the cGAC (cga_cmd_ready()).
61 *       2nd: write in reg2 the required 32-bit data (only if required).
62 *       3rd: write in reg1 the command code (bits 14-0) and set as a command
63 *           to all PEs (bit 15).
64 *   Note: It is the responsibility of the cGAC to forward correctly
65 *         the command to the PEs.
66 *   This sequence of operations can also be used to send commands
67 *   whose recipient is the cGAC (and not the PEs). In this
68 *   case, bit 15 of reg1 can be neglected.
69 */
70 void cga_start_all_PEs(unsigned int max_itr)
71 {
72     int value;
73
74     cga_cmd_ready();
75     write_reg(2, max_itr);
76
77     value = CMD_START_PE & 0x00007fff;
78     value = value | 0x00008000;
79     write_reg(1, value);
80 }
81
82
83 /* cga_set_parameter_PE:
84 *   Command to send a parameter to a specific PE in the cGAA.
85 *   parameter: coordinates {i,j} of the PE ('coord_i' and 'coord_j') and
86 *               parameter value ('param').
87 *
88 *   This function shows how to send a command to the cGAC that targets
89 *   a single PE:
90 *       1st: wait till (or check if) previous command has been processed by
91 *           the cGAC (cga_cmd_ready())
92 *       2nd: write in reg2 the required 32-bit data (only if required).
93 *       3rd: write in reg1 the command code (bits 14-0) and the coordinates
94 *           of the PE (bits 31-24 and 23-16 for the {i,j} PE coordinates).
95 *           Bit 15 must be 0.
96 *   Note: It is the responsibility of the cGAC to forward correctly
97 *         the command to the PE.
98 */
99 void cga_set_parameter_PE(int coord_i, int coord_j, int param)
100 {
101     int value;
102     int value_i, value_j;
103
104     value_i = coord_i & 0x000000ff;
105     value_j = coord_j & 0x000000ff;
106     value_i = value_i << 24;
107     value_j = value_j << 16;
108
109     value = CMD_SET_PARAMETER & 0x00007fff;
110     value = value_i | value_j | value;
111
112     cga_cmd_ready();
113     write_reg(2, param);
114     write_reg(1, value);
115 }

```

LISTING B.2: C code examples of functions that use the API used by the MicroBlaze to control the execution of the cGAP.

Appendix C

Design flow libraries

This appendix presents the libraries used in the design flow to build the cellular genetic algorithm processor (cGAP).

C.1 cGAP HLS library

Table C.1 presents a description of the files that constitute the cGAP HLS library. This library is used to describe the PE and cGAC in the C++ language so that they are translated to hardware with Catapult HLS tool.

TABLE C.1: High-level synthesis library developed to describe the PE and the cGAC in C++ with Catapult HLS.

File	Description
command_type_cGA.cpp command_type_cGA.h	Define the C++ class <code>command_type_cGA</code> . This class is used for manipulating the data type that defines the commands sent among the PEs and the cGA controller. Details can be found in Appendix A.1.
request_channel.cpp request_channel.h	Define the C++ class <code>request_channel</code> . Used by the PEs for the interface between a PE and a subpopulation memory. Details can be found in Appendix A.2.
log2ceil.h	Given an integer, it provides the logarithmic to the base 2 rounded to the smallest following (ceiling) integer number. Used mainly to manipulate parameters.
pow2.h	It provides the 2 raised to the power of a given integer number. Used mainly to manipulate parameters.

C.2 cGAP RTL library

Table C.2 presents a description of the files that constitute the cGAP RTL library. This library contains all the hardware module that describe the cGAP infrastructure (except the PE and cGAC that are implemented with HLS), and are described at the register-transfer level (RTL) using Verilog hardware description language.

TABLE C.2: cGAP RTL library used to describe the cGAP.

File	Description
ca50745.v	Implements the RNG described in Section 4.6.1. Requires the file rule_50745.v.
cellular_ga.v	Builds the cGAA (Fig. 4.2) and the communication infrastructure (Fig. 4.8). Requires the files ga_cell.v, cmd_to_cgac_line.v, cmd_from_cgac_line.v, and subpopulation.v (last one only for non-toroidal configurations of the cGAA).
channel_control.v	Implements a FIFO that communicates with a ac_channel data type interfaces produced by a Catapult HLS hardware module (Fig. 4.3 or 4.7).
cmd_from_cgac.v	Implements a routing FIFO that sends a command from the cGAC to a PE along the horizontal axis of the cGAA configuration (Fig. 4.8).
cmd_from_cgac_line.v	Implements a routing FIFO that sends a command from the cGAC to a PE along the vertical axis of the cGAA configuration (Fig. 4.8).
cmd_to_cgac.v	Implements a routing FIFO that sends a command from a PE to the cGAC along the vertical axis of the cGAA configuration (Fig. 4.8).
cmd_to_cgac_line.v	Implements a routing FIFO that sends a command from a PE to the cGAC along the horizontal axis of the cGAA configuration (Fig. 4.8).
DP_RAM.v	Describes the cGA dual-port memory (Fig. 4.7).
ga_cell.v	Defines a cell used to build the regular shape of the cGAA (Fig. 4.2). Requires the files pe_wrapper, subpopulation.v, cmd_from_cgac.v, and cmd_to_cgac.v.
pe_wrapper	Implements the interface (a wrapper) to the PE module produced by Catapult HLS. Requires the files channel_control.v and PE.v (last one from HLS).
rule_50745.v	Implements a single cell used to build the RNG using CA techniques (Fig. 4.13).
subpopulation.v	Describes a subpopulation memory of the cGAP (Fig. 4.4). Requires the file subpopulation_ram.v.
subpopulation_ram.v	Describes a dual-port RAM used for implementing a subpopulation memory (Fig. 4.4).
user_logic.v	cGAP top level module that implements the complete cGAP. Includes the interface to the cGAC module produced by Catapult HLS. Additionally, it describes the required logic for the AXI4-Lite interface so that the cGAP communicates with a host computer. Requires the files DP_RAM.v, cellular_ga.v, channel_control.v, ca50745.v, and cGA_controller.v (last one from HLS).

C.3 cGAP API library

Table C.3 presents a description of the files that constitute the cGAP API library. This library includes the C functions that build the application programming interface (API) and are used to ensure the access to the interface of the cGAP. Therefore, a host computer uses this API to access the cGAP (as memory mapped device).

TABLE C.3: cGAP API library used to communicate with the cGAP.

File	Description
ip_cga_interf.c ip_cga_interf.h	Define the API to access the cGAP, mainly mapping the hardware to memory and access to the registers of the cGAP. Appendix B.1 provides the details of these functions.
ip_cga_mem_addr.h	Defines the macros used to set the memory addresses needed to map the cGAP as a memory mapped peripheral of the MicroBlaze.