Chapter 1
A Reconfigurable Hardware for Genetic Algorithms

Abstract. In this chapter, we propose a massively parallel architecture of a hardware implementation of genetic algorithms. This design is quite innovative as it provides a viable solution to the fitness computation problem, which depends heavily on the problem-specific knowledge. The proposed architecture is completely independent of such specifics. It implements the fitness computation using a neural network. The hardware implementation of the used neural network is stochastic and thus minimises the required hardware area without much increase in response time. Last but not least, we demonstrate the characteristics of the proposed hardware and compare it to existing ones.

1.1 Introduction

Generally speaking, a genetic algorithm is a process that evolves a set of individuals, also called chromosomes, which constitutes the generational population, producing a new population. The individuals represent a solution to the problem in consideration. The freshly produced population is yield using some genetic operators such as selection, crossover and mutation that attempt to simulate the natural breeding process in the hope of generating new solutions that are fitter, i.e. adhere more the problem constraints.

Previous work on hardware genetic algorithms can be found in [5, 10, 12]. Mainly, Earlier designs are hardware/software codesigns and they can be divided into three distinct categories: (i) those that implement the fitness computation in hardware and all the remaining steps including the genetic operators in software, claiming that the bulk computation within genetic evolution is the fitness computation. The hardware is problem-dependent; (ii) and those that implement the fitness computation in software and the rest in hardware, claiming that the ideal candidate are the genetic operators as these exhibit regularity and generality [2, 7]. (iii) those that implement the whole genetic algorithm in hardware [10]. We believe that both approaches are worthwhile but a hardware-only implementation of both the fitness calculation and genetic operators is also valuable. Furthermore, a hardware implementation that is problem-independent is yet more useful.
The remainder of this chapter is divided into five sections. In Section 1.2, we describe the principles of genetic algorithms. Subsequently, in Section 1.3, we propose and describe the overall hardware architecture of the problem-independent genetic algorithm. Thereafter, in Section 1.4, we detail the architecture of each of the component included in the hardware genetic algorithm proposed. Then, in Section 1.5, we assess the performance of the proposed architecture. Finally, we draw some conclusions.

1.2 Principles of Genetic Algorithms

Genetic algorithms maintain a population of individuals that evolve according to selection rules and other genetic operators, such as mutation and crossover. Each individual receives a measure of fitness. Selection focuses on high fitness individuals. Mutation and crossover provide general heuristics that simulate the reproduction process. Those operators attempt to perturb the characteristics of the parent individuals as to generate distinct offspring individuals.

Genetic algorithms are implemented through the procedure described by Algorithm 1.1, wherein parameters $ps$, $ef$ and $gn$ are the population size, the expected fitness of the returned solution and the maximum number of generation allowed respectively.

**Algorithm 1.1. GA – Genetic algorithms basic cycle**

**Require:** population size $ps$, expected fitness $ef$, generation number $gn$

**Ensure:** the problem solution

- generation := 0
- population := initialPopulation()
- fitness := evaluate(population)

repeat
  - parents := select(population)
  - population := mutate(crossover(parents))
  - fitness := evaluate(population)
  - generation := generation + 1
until (fitness[i] = ef, $1 \leq i \leq ps$) OR (generation $\geq gn$)

In Algorithm 1.1 function `initialPopulation` returns a valid random set of individuals that compose the population of first generation while function `evaluate` returns the fitness of a given population storing the result into fitness. Function `select` chooses according to some random criterion that privilege fitter individuals, the individuals that should be used to generate the population of the next generation and function `crossover` and `mutate` implement the crossover and mutation process respectively to actually yield the new population.