UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA MECÂNICA

OTIMIZAÇÃO POR INTELIGÊNCIA DE ENXAMES USANDO ARQUITETURAS PARALELAS PARA APLICAÇÕES EMBARCADAS

DANIEL MAURICIO MUÑOZ ARBOLEDA

ORIENTADOR: CARLOS HUMBERTO LLANOS QUINTERO

COORIENTADOR: LEANDRO DOS SANTOS COELHO - PUCPR

TESE DE DOUTORADO EM MECATRÔNICA

PUBLICAÇÃO: ENM.TD - 02/12

BRASÍLIA/DF: DEZEMBRO - 2012

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA MECÂNICA

OTIMIZAÇÃO POR INTELIGÊNCIA DE ENXAMES USANDO ARQUITETURAS PARALELAS PARA APLICAÇÕES EMBARCADAS

DANIEL MAURICIO MUÑOZ ARBOLEDA

TESE DE DOUTORADO APRESENTADA AO PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS MECATRÔNICOS DO DEPARTAMENTO DE ENGENHARIA MECÂNICA DA UNIVERSIDADE DE BRASÍLIA, COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM MECATRÔNICA.

APROVADA POR:

Prof. Carlos Humberto Llanos Quintero, Dr. (ENM-UnB)
(Orientador)

Prof. Juan Manuel Moreno Arostegui, PhD. (UPC-Espanha)
(Examinador Externo)

Prof. Edward David Moreno, PhD. (UFSe)
(Examinador Externo)

Prof. José Camargo da Costa, PhD. (UnB)
(Examinador Externo)

Prof. Ricardo Jacobi, PhD. (UnB)
(Examinador Interno)

BRASÍLIA/DF, DEZEMBRO DE 2012

FICHA CATALOGRÁFICA

MUÑOZ, D.M.
Otimização por inteligência de enxames baseada em arquiteturas paralelas em aplicações embarcadas. [Distrito Federal] 2012.
xvi, 192p. 210 x 297 mm (ENM/FT/UNB, Doutor, Sistemas Mecatrônicos, 2012).
Tese de Doutorado. Universidade de Brasília. Faculdade de Tecnologia.
Departamento de Engenharia Mecânica.
1. Inteligência de Enxames 2. FPGAs
3. Ponto Flutuante 4. Robótica Móvel
I. ENM/FT/UNB II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

MUÑOZ, D.M. (2012). Otimização por inteligência de enxames usando arquiteturas paralelas para aplicações embarcadas. Tese de Doutorado em Mecatrônica, Publicação ENM.TD - 02/12, Departamento de Engenharia Mecânica , Universidade de Brasília, Brasília, DF, 192p.

CESSÃO DE DIREITOS

NOME DO AUTOR: Daniel Mauricio Muñoz Arboleda

TÍTULO DA TESE DE DOUTORADO: Otimização por inteligência de enxames usando arquiteturas paralelas para aplicações embarcadas.

GRAU / ANO: Doutor / 2012

Daniel Mauricio Muñoz Arboleda
CLN 407 Bloco C, sala 83
70855-530 Brasília - DF - Brasil

Dedicatória

A mi grande familia, con una gran sonrisa

Al mia granda familio, kun granda rideto

Agradecimentos

Ao meu orientador Prof. Carlos Humberto Llanos Quintero pela confiança, compreensão, paciência e dedicação. Suas orientações e conhecimentos transmitidos em diversos aspectos da vida foram inestimáveis para a conclusão desta tese.

Ao Prof. Leandro dos Santos Coelho pela coorientação, sugestões e intercâmbio de ideias nos temas de otimização bio-inspirada. Ao Prof. Mauricio Ayala Rincón pela coorientação, incentivo, suporte e pelo exemplo de dedicação e disciplina.

Aos professores que aportaram ideias e realizaram observações e sugestões durante a fase de desenvolvimento e de avaliação deste trabalho: Michael Hübner, Guilherme Caribé, Ricardo Jacobi e José Camargo.

A todos os colegas que de uma forma ou outra contribuíram com o desenvolvimento deste trabalho, em especial a André Luiz, Jones Yudi, Janier e Magno. A Diego Felipe Sánchez pela colaboração e trabalho em equipe durante o desenvolvimento dos operadores aritméticos em ponto flutuante.

Muito obrigado aos irmãos que encontrei durante estes anos de caminhada evolutiva. A todos os amigos em Brasília e no mundo, assim como aos colegas de laboratório pelo apoio, incentivo e pelos momentos de distração. Obrigado à comunidade colombiana em Brasília e ao povo Brasileiro que me fizeram sentir como em casa.

Sou grato de forma especial à família Carvalho Barbosa pela acolhida e incentivo. A Mariana com quem muito tenho aprendido e quem tem participado deste trabalho de diversas maneiras, obrigado pelo grande coração.

À minha família e de forma especial aos meus irmãos e aos meus pais Alma e Diego pelo carinho e apoio incondicional. Obrigado pela compreensão porque, mesmo fisicamente distantes, sabem estar presentes a cada momento. Obrigado por todos os ensinamentos. Sem seu exemplo de determinação e disciplina este trabalho poderia ter começado mas certamente não o teria acabado.

Ao CNPq, DPP-UnB e FINATEC, pelo suporte financeiro.

RESUMO

OTIMIZAÇÃO POR INTELIGÊNCIA DE ENXAMES USANDO ARQUITETURAS PARALELAS PARA APLICAÇÕES EMBARCADAS

Autor: Daniel Mauricio Muñoz Arboleda
Orientador: Carlos Humberto Llanos Quintero
Programa de Pós-graduação em Sistemas Mecatrônicos
Brasília, Dezembro de 2012

Este trabalho apresenta um estudo da implementação em FPGAs (*Field Programmable Gate Array*) de algoritmos de otimização bioinspirados baseados em inteligência de enxames, voltados principalmente para aplicações embarcadas. Nos problemas de otimização embarcada, a dimensionalidade (número de variáveis de decisão) é relativamente pequena (algumas dezenas), porém, os problemas devem ser resolvidos em uma escala de tempo desde os milissegundos até alguns segundos.

A abordagem utilizada está baseada em uma representação aritmética de ponto flutuante e no uso de operações de fácil implementação em FPGAs, permitindo explorar o paralelismo intrínseco dos algoritmos por inteligência de enxames, visando obter ganhos de desempenho em termos do tempo de execução e da qualidade da solução.

Foram exploradas as arquiteturas de *hardware* dos algoritmos PSO (*Particle Swarm Optimization*), ABC (*Artificial Bee Colony*), FA (*Firefly Algorithm*) e SFLA (*Shuffled Frog Leaping Algorithm*), assim como de algumas variantes propostas para os mesmos. Estudos de consumo de recursos para diferente número de partículas paralelas e dimensionalidade dos problemas foram realizados no intuito verificar a aplicabilidade dos algoritmos em arquiteturas reconfiguráveis. Adicionalmente, a qualidade das soluções obtidas pelas arquiteturas propostas foi validada usando problemas de teste tipo *benchmark*. Os algoritmos estudados foram implementados no processador de *software* embarcado MicroBlaze e em um PC de escritório, permitindo, assim, realizar comparações do tempo de execução entre as implementações de *hardware* e *software*. Finalmente, uma solução de *hardware* foi proposta para a solução de um problema de otimização embarcada, onde é realizado o treinamento *online* de um controlador neural de um robô móvel de pequeno porte.

Os resultados experimentais mostram que a implementação em FPGAs dos algoritmos por inteligência de enxames é viável em termos de consumo de recursos de *hardware*. Foram obtidos fatores de aceleração de três ordens de magnitude em comparação com a implementação *software* no MicroBlaze e de 3.6 vezes em comparação com a solução no PC de escritório.

ABSTRACT

SWARM INTELLIGENCE OPTIMIZATION BASED ON PARALLEL ARCHITECTURES FOR EMBEDDED APPLICATIONS

Author: Daniel Mauricio Muñoz Arboleda
Advisor: Carlos Humberto Llanos Quintero
Programa de Pós-graduação em Sistemas Mecatrônicos
Brasília, December 2012

This work presents a study of the FPGA (Field Programmable Gate Array) implementation of swarm intelligence optimization algorithms, applied to embedded optimization systems. In embedded optimization problems the dimensionality (problem size) is smaller than in conventional ones; however, the problems must be solved at millisecond/second time-scales.

The approach presented in this work is based on the floating-point arithmetic representation as well as on operations that can be easily implemented on FPGAs, allowing the intrinsic parallelism of the swarm intelligence based algorithms to be explored in order to improve the execution time and the quality of the solutions.

Hardware architectures of the PSO (*Particle Swarm Optimization*), ABC (*Artificial Bee Colony*), FA (*Firefly Algorithm*) and SFLA (*Shuffled Frog Leaping Algorithm*) algorithms, as well as some proposed modifications, were mapped on FPGAs. The cost in logic area of the proposed architectures was estimated for different swarm sizes and problem sizes in order to validate the applicability of the algorithms for reconfigurable architectures. In addition, the quality of the solutions obtained by the proposed architectures was validated using two unimodal and two multimodal bechmarks test problems. The algorithms were also implemented on two software processors, the MicroBlaze embedded processor and a conventional Desktop solution, allowing for comparisons of the execution time between the hardware and software implementations. Finally, a hardware solution was proposed for solving the online training process of a neural network controller of a small mobile robot.

The experimental results demonstrate that the FPGA implementation of the swarm intelligence algorithms is feasible in terms of the hardware resources consumption. Speed-up factors of three orders of magnitude and 3.6 times were achieved in comparison with the MicroBlaze and the Desktop solutions, respectively.

Sumário

1 INTRODUÇÃO **2**

1.1 DESCRIÇÃO DO PROBLEMA 5

1.2 OBJETIVOS 6

1.2.1 Objetivo Geral 6

1.2.2 Objetivos específicos 6

1.3 ASPECTOS METODOLÓGICOS DO TRABALHO 7

1.4 CONTRIBUIÇÕES DO TRABALHO 9

1.5 ORGANIZAÇÃO DO TRABALHO 12

2 FUNDAMENTAÇÃO TEÓRICA **14**

2.1 OTIMIZAÇÃO 14

2.1.1 Elementos de um modelo de otimização 15

2.1.2 Complexidade 16

2.1.3 Classificação dos métodos de otimização 17

2.1.4 Algoritmos de otimização bioinspirados 19

2.1.5 Inteligência de enxames 20

2.2 *HARDWARE* RECONFIGURÁVEL 21

2.2.1 Arquiteturas *von Neumann* 23

2.2.2 FPGAs 25

2.2.3 Vantagens da computação reconfigurável 27

2.3 OTIMIZAÇÃO POR ENXAME DE PARTÍCULAS (PSO) 29

2.3.1 Algoritmo PSO básico 30

2.3.2 Modificações do algoritmo PSO 32

2.3.3 Topologias *GBEST* e *LBEST* 33

2.3.4 Técnicas híbridas e adaptativas 34

2.4 OTIMIZAÇÃO POR COLÔNIA DE ABELHAS ARTIFICIAIS (ABC) 35

2.5 OTIMIZAÇÃO POR COLÔNIA DE VAGA-LUMES (FA) 37

2.6 OTIMIZAÇÃO POR EMBARALHAMENTO DE SALTO DE SAPOS (SFLA) 39

2.7 MÉTODOS DE ADIÇÃO DE DIVERSIDADE ARTIFICIAL 44

2.7.1 O método atrativo repulsivo 45
2.7.2 O método de congregação passiva seletiva 45
2.7.3 O método de aprendizado em oposição 46
2.8 IMPLEMENTAÇÕES PARALELAS DOS ALGORITMOS CITADOS . 47
2.8.1 O uso de *clusters* de PCs nos algoritmos de inteligência de enxames 48
2.8.2 O uso de GPUs nos algoritmos de inteligência de enxames . . . 49
2.8.3 O uso de FPGAs nos algoritmos de inteligência de enxames . . 49
2.9 CONCLUSÕES DO CAPÍTULO 52

3 IMPLEMENTAÇÃO DAS UNIDADES DE CÁLCULO ARITMÉTICO E TRIGONOMÉTRICO EM PONTO FLUTUANTE 54
3.1 CONSIDERAÇÕES INICIAIS 54
3.2 O PADRÃO IEEE-754 58
3.3 IMPLEMENTAÇÃO DOS OPERADORES ARITMÉTICOS 58
3.3.1 Soma/subtração em ponto flutuante (*FPadd*) 58
3.3.2 Multiplicação em ponto flutuante (*FPmul*) 60
3.3.3 Divisão em ponto flutuante 61
3.3.4 Raiz quadrada em ponto flutuante 62
3.4 IMPLEMENTAÇÃO DOS OPERADORES TRIGONOMÉTRICOS . . 63
3.4.1 Algoritmo CORDIC 64
3.4.2 Implementação CORDIC das funções trigonométricas 66
3.4.3 Implementação CORDIC para a função exponencial 68
3.5 RESULTADOS DE SÍNTESE 70
3.6 VALIDAÇÃO DAS ARQUITETURAS PROPOSTAS 72
3.7 COMPARAÇÃO DO TEMPO DE EXECUÇÃO 76
3.8 DISCUSÕES FINAIS DO CAPÍTULO E CONTRIBUIÇÕES 79

4 IMPLEMENTAÇÕES E RESULTADOS DOS ALGORITMOS DE OTIMIZAÇÃO POR INTELIGÊNCIA DE ENXAMES 82
4.1 CONSIDERAÇÕES INICIAIS 82
4.2 UNIDADE DE GERAÇÃO DE NÚMEROS ALEATÓRIOS 83
4.3 IMPLEMENTAÇÃO *HARDWARE* DO NÚMERO OPOSTO 85
4.4 IMPLEMENTAÇÃO *HARDWARE* DO ALGORITMO PSO 86
4.5 IMPLEMENTAÇÃO *HARDWARE* DO ALGORITMO O-PSO 88
4.6 IMPLEMENTAÇÃO *HARDWARE* DO ALGORITMO ABC 90
4.7 IMPLEMENTAÇÃO *HARDWARE* DO ALGORITMO O-ABC 94
4.8 IMPLEMENTAÇÃO *HARDWARE* DO ALGORITMO FA 95
4.8.1 Arquitetura do cálculo da atração 96

4.9 IMPLEMENTAÇÃO *HARDWARE* DO ALGORITMO O-FA 98
4.10 IMPLEMENTAÇÃO *HARDWARE* DO ALGORITMO SFLA 99
4.10.1 Arquitetura da unidade de embaralhamento 101
4.11 IMPLEMENTAÇÃO *HARDWARE* DO ALGORITMO O-SFLA 103
4.12 CONJUNTO DE *BENCHMARKS* USADOS PARA VALIDAÇÃO . . 104
4.12.1 Função *Esfera* 105
4.12.2 Função *Quadric* 106
4.12.3 Função *Rosenbrock* 107
4.12.4 Função *Rastrigin* 107
4.13 DESENHO DOS EXPERIMENTOS 108
4.14 RESULTADOS DE SÍNTESE 110
4.15 TESTES PARA VERIFICAÇÃO DAS IMPLEMENTAÇÕES 117
4.16 RESULTADOS DE CONVERGÊNCIA 118
4.17 TESTES DE SIGNIFICÂNCIA ESTATÍSTICA 125
4.18 COMPARAÇÃO DOS RESULTADOS DE CONVERGÊNCIA 128
4.19 COMPARAÇÃO DO TEMPO DE EXECUÇÃO 133
4.20 DISCUSSÕES FINAIS DO CAPÍTULO E CONTRIBUIÇÕES 138

5 APLICAÇÃO EM PROBLEMAS DE OTIMIZAÇÃO EMBARCADA 142
5.1 DESCRIÇÃO DO PROBLEMA 142
5.2 O MODELO NEURAL 143
5.3 DESCRIÇÃO DA ARQUITETURA 145
5.3.1 Implementação *hardware* da função custo 146
5.4 O AMBIENTE DE DESENVOLVIMENTO 148
5.5 RESULTADOS DE SÍNTESE 149
5.6 RESULTADOS DE SIMULAÇÃO 150
5.7 TESTES DE TOLERÂNCIA A FALHA NOS SENSORES 152
5.8 COMPARAÇÃO DO TEMPO DE EXECUÇÃO 156
5.9 DISCUSSÕES FINAIS DO CAPÍTULO E CONTRIBUIÇÕES 158

6 CONCLUSÕES 160
6.1 ASPECTOS GERAIS 160
6.2 UNIDADES DE CÁLCULO ARITMÉTICO E TRIGONOMÉTRICO EM PONTO FLUTUANTE 160
6.3 IMPLEMENTAÇÃO DOS ALGORITMOS DE INTELIGÊNCIA DE ENXAMES 163
6.4 TREINAMENTO SUPERVISIONADO DO CONTROLADOR NEURAL DE UM ROBÔ MÓVEL 164

6.5 CONTRIBUIÇÕES 165
6.6 TRABALHOS FUTUROS 168
6.6.1 Estimação do consumo de potência 168
6.6.2 Técnicas autoadaptativas para melhoria dos algoritmos 168
6.6.3 Uso de instruções customizadas em processadores embarcados . 168
6.6.4 Aplicações em otimização embarcada 169
6.6.5 Implementação em GPUs 170

REFERÊNCIAS BIBLIOGRÁFICAS **171**

APÊNDICES **186**

A PUBLICAÇÕES REALIZADAS **187**
A.1 TRABALHOS PUBLICADOS 187
A.1.1 Operadores aritméticos e trigonométricos em ponto flutuante . . 187
A.1.2 Algoritmos PSO, SFLA e ABC 187
A.2 TRABALHOS SUBMETIDOS EM JOURNAL INTERNACIONAL . . 188
A.3 PUBLICAÇÕES RELACIONADAS 188

B ESTUDO DE CONVERGÊNCIA DOS MÉTODOS DE DIVERSIDADE ARTIFICIAL **190**

Lista de Tabelas

3.1 Quadro comparativo entre arquiteturas 56
3.2 Modos de operação do algoritmo CORDIC 66
3.3 Resultados de síntese dos operadores implementados (dispositivo xc5vlx110t) 71
3.4 Erro quadrático médio e erro absoluto médio dos operadores aritméticos 74
3.5 Erro quadrático médio e erro absoluto médio dos operadores trigonométricos 75
3.6 Comparação do tempo de execução dos operadores aritméticos 78
3.7 Comparação do tempo de execução dos operadores trigonométricos . . 78

4.1 Experimentos realizados para os algoritmos PSO, ABC e FA 109
4.2 Experimentos realizados para o algoritmo SFLA 110
4.3 Resultados de síntese. Função *Esfera* (dispositivo xc5vlx110t) 111
4.4 Resultados de síntese. Função *Quadric* (dispositivo xc5vlx110t) 112
4.5 Resultados de síntese. Função *Rosenbrock* (dispositivo xc5vlx110t) . . . 113
4.6 Resultados de síntese. Função *Rastrigin* 114
4.7 Condições experimentais 119
4.8 Comparação de convergência entre as arquiteturas HPPSO e HPOPSO 120
4.9 Comparação de convergência entre as arquiteturas HPABC e HPOABC 120
4.10 Comparação de convergência entre as arquiteturas HPFA e HPOFA . . 121
4.11 Comparação de convergência entre as arquiteturas HPSFLA e HPOSFLA 122
4.12 Convergência das implementações em *hardware*. Função *Esfera* 122
4.13 Convergência das implementações em *hardware*. Função *Quadric* 123
4.14 Convergência das implementações em *hardware*. Função *Rosenbrock* . . 123
4.15 Convergência das implementações em *hardware*. Função *Rastrigin* . . . 124
4.16 Mediana e significância estatística para a arquitetura HPPSO 126
4.17 Mediana e significância estatística para a arquitetura HPABC 126
4.18 Mediana e significância estatística para a arquitetura HPFA 127
4.19 Mediana e significância estatística para a arquitetura HPSFLA 127
4.20 Medianas e significância estatística entre as arquiteturas 128
4.21 Convergência das implementações em *software*. Função *Esfera* 129
4.22 Convergência das implementações em *software*. Função *Quadric* 129
4.23 Convergência das implementações em *software*. Função *Rosenbrock* . . 130

4.24 Convergência das implementações em *software*. Função *Rastrigin* . . . 130
4.25 Comparação de convergência HW/SW para o algoritmo O-PSO 131
4.26 Comparação de convergência HW/SW para o algoritmo GBOABC . . . 132
4.27 Comparação de convergência HW/SW para o algoritmo GOFA 132
4.28 Comparação de convergência HW/SW para o algoritmo O-SFLA 133
4.29 Comparação do tempo de execução por iteração, 10 dimensões 136
4.30 Fator de aceleração por iteração *hardware* versus MicroBlaze 136
4.31 Fator de aceleração por iteração *hardware* versus PC escritório 137
4.32 Comparação do tempo de execução total, 10 dimensões 137
4.33 Comparação do custo, convergência e desempenho 140

5.1 Configuração de parâmetros, 10 partículas 149
5.2 Resultados de síntese da arquitetura HPOPSO-RNA 150
5.3 Pesos da rede obtidos pela arquitetura HPOPSO-RNA 151
5.4 Pesos da rede obtidos pela arquitetura HPOPSO-RNA quando falhas nos sensores IR são detectadas 154
5.5 Comparação do tempo de execução por iteração 158

B.1 Condições experimentais para os algoritmos PSO, arPSO, PSOpc e O-PSO 190
B.2 Resultados de convergência dos métodos de adição de diversidade artificial 191

Lista de Figuras

1.1 Foco do trabalho 6

2.1 Classificação das metaheurísticas bioinspiradas 20
2.2 Formas de inteligência na natureza 21
2.3 Taxonomia para CIs 22
2.4 Arquitetura *von Neumann* 24
2.5 (a) Bloco lógico Altera StratixII (b) LUT de duas entradas e uma saída 25
2.6 (a) Estrutura geral da FPGA (b) bloco de switch (c) bloco de conexão 26
2.7 Topologias: (a) estrela ou *gbest* (b) anel ou *lbest* com k=2 34

3.1 O padrão IEEE-754 58
3.2 Arquitetura da unidade de soma/subtração *FPadd* 59
3.3 Arquitetura da unidade de multiplicação *FPmul* 61
3.4 Divisão *Newton-Raphson* 62
3.5 Raiz quadrada *Newton-Raphson* 64
3.6 Arquitetura iterativa da unidade *Cordicsincos* 67
3.7 Arquitetura iterativa da unidade *Cordicatan* 68
3.8 Arquitetura iterativa da unidade *CordicTaylorexp* 69
3.9 Validação da arquitetura *CordicTaylorexp* para argumentos pequenos . 77

4.1 Unidade de geração de números aleatórios em ponto flutuante 84
4.2 Arquitetura de *hardware* do cálculo do número oposto para espaços de busca simétricos 85
4.3 Arquitetura de uma partícula no algoritmo PSO 86
4.4 Arquitetura de *hardware* do algoritmo PSO (HPPSO) 87
4.5 Arquitetura da partícula com aprendizado em oposição (antipartícula) . 90
4.6 Arquitetura *hardware* do algoritmo O-PSO (HPOPSO) 90
4.7 Arquitetura de movimento de uma abelha operária ou seguidora 92
4.8 Arquitetura *hardware* do algoritmo ABC (HPABC) 93
4.9 Arquitetura *hardware* do algoritmo GBOABC (HPOABC) 94
4.10 Arquitetura de um vaga-lume 96
4.11 Arquitetura do cálculo de atração 97

4.12 Arquitetura *hardware* do algoritmo GFA (HPFA) 98
4.13 Arquitetura geral do algoritmo GOFA (HPOFA) 100
4.14 Arquitetura de atualização do sapo 100
4.15 Arquitetura da unidade de embaralhamento 102
4.16 Arquitetura de *hardware* do algoritmo SFLA (HPSFLA) 102
4.17 Arquitetura de *hardware* do algoritmo O-SFLA (HPOSFLA) 104
4.18 Arquitetura da função custo *Esfera* 106
4.19 Arquitetura da função custo *Quadric* 107
4.20 Arquitetura da função custo *Rosenbrock* 108
4.21 Arquitetura da função custo *Rastrigin* 109
4.22 Comparação do consumo de recursos para a função *Esfera* 111
4.23 Comparação do consumo de recursos para a função *Quadric* 112
4.24 Comparação do consumo de recursos para a função *Rosenbrock* 113
4.25 Comparação do consumo de recursos para a função *Rastrigin* 115
4.26 Ambiente de validação das arquiteturas de *hardware* 118
4.27 Implementação MicroBlaze 134

5.1 (a) Descrição do robô móvel, (b) Modelo do controlador neural 144
5.2 Arquitetura HPOPSO-RNA para treinamento do controlador neural . . 145
5.3 (a)Arquitetura do modelo neural (b) Arquitetura da função custo . . . 147
5.4 Plataforma de validação 148
5.5 Ambiente para obtenção dos dados de treinamento 149
5.6 Resultados de simulação para o comportamento de evasão de obstáculos 151
5.7 Resultados de simulação para ambientes desconhecidos 153
5.8 Resultados de simulação com tolerância a falha nos sensores 155
5.9 Tempo de execução da arquitetura HPOPSO-RNA 157

LISTA DE SÍMBOLOS Siglas

ASIC	Application Specific Integrated Circuits
CMOS	Complementary Metal Oxide Semiconductor
CORDIC	COordinate Rotation DIgital Computer
DLL	Delay-Locked Loop
DSPs	Digital Signal Processing
FF	Flip-flops
FPGAs	Field Programmable Gate Array
FSM	Finite State Machine
FMP	Função de Massa de Probabilidade
GNA	Gerador de Números Aleatórios
GPPs	General Purpose Processors
GPU	Graphics Processing Unit
HW/SW	Hardware/Software
LFSR	Linear Feedback Shift Register
LUT	Look Up Table
MAE	Mean Absolute Error
MSE	Mean Square Error
OBL	Opposition-based Learning
PAR	Place and Route
PC	Personal Computer
PLB	Processor Local Bus
PLL	Phase-Locked Loop
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RNA	Rede Neural Artificial
ROM	Read Only Memory
TTL	Transistor-Transistor Logic
ULA	Unidade lógica aritmética
VHDL	Very High Speed Integrated Circuits Hardware Description Language
VLSI	Very Large Scale Integration
XST	Xilinx Synthesis Tool

Arquiteturas de *hardware*

Cordicatan	arquitetura CORDIC para a função arco-tangente

Cordicsincos	arquitetura CORDIC para as funções seno e cosseno
CordicTaylorexp	arquitetura CORDIC para a função exponencial
FPadd	Floating-point Addition/subtraction unit
FPCordic	Floating-point CORDIC unit
FPdiv	Floating-point Division unit
FPfrac	Floating-point Fractional part
FPmul	Floating-point Multiplication unit
FPTaylor	Floating-point Taylor expansion unit
HPABC	Hardware Parallel Artificial Bee Colony
HPOABC	Hardware Parallel Opposition-based Artificial Bee Colony
HPFA	Hardware Parallel Firefly Algorithm
HPOFA	Hardware Parallel Opposition-based Firefly Algorithm
HPPSO	Hardware Parallel Particle Swarm Optimization
HPOPSO	Hardware Parallel Opposition-based Particle Swarm Optimization
HPOPSO-RNA	Hardware Parallel Opposition-based Particle Swarm Optimization para treinamento de RNAs
HPSFLA	Hardware Parallel Shuffled Frog Leaping Algorithm
HPOSFLA	Hardware Parallel Opposition-based Shuffled Frog Leaping Algorithm

Algoritmos

ABC	Artificial Bee Colony
FA	Firefly Algorithm
GBABC	Global Bee Artificial Bee Colony
GBOABC	Global Bee Opposition-based Artificial Bee Colony
GFA	Global Firefly Algorithm
GOFA	Global Opposition-based Firefly Algorithm
GS	Algoritmo Goldschmidt
NR	Algoritmo Newton-Raphson
PSO	Particle Swarm Optimization
O-PSO	Oppostion-based Particle Swarm Optimization
OPSO-RNA	Oppostion-based Particle Swarm Optimization para treinamento de RNAs
SFLA	Shuffled Frog Leaping Algorithm
O-SFLA	Oppostion-based Shuffled Frog Leaping Algorithm
vFPUgen	VHDL Flotaing-point Unit Generator

Capítulo 1 INTRODUÇÃO

A implementação de técnicas de otimização é de grande importância em áreas como física, química, engenharia, administração, economia, entre outras. A otimização objetiva a solução de problemas visando encontrar uma resposta que proporcione o melhor resultado entre um conjunto de alternativas disponíveis, permitindo assim, encontrar maiores ganhos, maior produção e menor custo. Frequentemente, esses problemas envolvem a utilização mais eficiente de recursos de tempo, dinheiro, maquinaria e pessoal [1].

De forma geral, a solução de um problema de otimização requer da formulação de um modelo matemático que preserve a equivalência com o problema real. Problemas que envolvem um único ótimo global são chamados de *problemas unimodais* ou de *otimização convexa*. Por outro lado, problemas que envolvem ótimos locais ou mais de um ótimo global são chamados de *problemas multimodais*.

Diversas técnicas têm sido propostas pela comunidade científica para a solução de problemas de otimização. As meta-heurísticas são métodos eficientes em casos em que os modelos dos problemas são conhecidos, porém tem-se pouca informação sobre a aparência da solução ótima [2]. Também são usadas quando se tem uma descrição da saída desejada e, então, procura-se conhecer as entradas que levam a esta saída. Nestes casos, os métodos exatos são impraticáveis devido ao alto custo computacional e à possibilidade de encontrar soluções subótimas [2], especificamente quando os problemas envolvem não linearidades e interações entre as variáveis de projeto.

Neste sentido, algoritmos baseados em populações têm sido propostos nos últimos anos [3], [4], [5]. Os métodos de otimização baseados em populações consideram a interação entre agentes da mesma população, sua interação com o ambiente assim como sua evolução. Os modelos computacionais utilizados para simular a evolução natural englobam um conjunto de técnicas inspiradas no comportamento social de algumas espécies naturais. Essas técnicas, conhecidas como técnicas de *inteligência de enxames* são atualmente motivo de pesquisas que visam aplicações em diversas áreas do conhecimento. Estas técnicas têm sido acolhidas pela comunidade científica

dada a sua *filosofia de fácil implementação* e à *qualidade* aceitável das soluções obtidas (aproximação ao valor ótimo), principalmente quando aplicados a problemas complexos em que as soluções exatas são impraticáveis. Entretanto, a principal desvantagem destas técnicas de inteligência de enxames é o elevado tempo de execução dos algoritmos envolvidos.

Algoritmos de otimização por inteligência de enxames, e em geral os algoritmos baseados em populações, possuem uma natureza essencialmente paralela. Em alguns grupos de animais, ou nas colônias de insetos ou bactérias, os indivíduos exploram o seu habitat simultaneamente mediante a realização de tarefas cooperativas, intercambiando informação por meio de processos de agregação ou congregação, visando encontrar as condições ótimas para alcançar um determinado objetivo.

Diversas técnicas híbridas e adaptativas têm sido aplicadas melhorando o desempenho dos algoritmos de otimização [6], [7], [8]. No entanto, muitos problemas de engenharia continuam sendo impraticáveis devido ao alto custo computacional requerido. Este último fato torna-se evidente em aplicações embarcadas em que a capacidade computacional é limitada.

Sistemas computacionais embarcados representam um campo amplo de desenvolvimento, responsável por um mercado de trabalho crescente [9]. Sistemas embarcados podem ser entendidos como sistemas computacionais especializados, desenvolvidos para uma tarefa específica. Comumente, são projetados em unidades de processamento com capacidade limitada e funcionam em conjunto com sensores que lhes permitem coletar dados, viabilizando a tomada de decisões e a realização de ações de forma automática [10].

Uma das principais diferenças entre sistemas embarcados e sistemas computacionais de propósitos gerais, como por exemplo, os computadores pessoais (PCs), é que os sistemas embarcados usualmente requerem *hardware* e *software* de propósito específico [9], [10]. Atualmente existe uma grande demanda de sistemas embarcados que permitam não apenas realizar tarefas computacionais de alto desempenho, senão, que também cumpram requerimentos de robustez, tolerância a falhas, baixo custo e baixo consumo de energia [11].

Alguns tipos de sistemas embarcados possuem características de adaptabilidade em função das condições de operação, por exemplo, mudanças estatísticas dos valores de

entrada do sistema, ou distúrbios originados por mudanças no ambiente [12]. Nestes sistemas, conhecidos como *sistemas adaptativos*, comumente são executados algoritmos de ajuste de parâmetros que permitem garantir convergência e estabilidade global. Tais algoritmos estão baseados na solução de *problemas de otimização embarcada*. Atualmente, as soluções existentes para este tipo de problemas estão baseadas nos métodos de otimização convexa (minimização de funções convexas), permitindo a solução de problemas em uma escala de tempo da ordem de milissegundos [13], [14]. Entretanto, tais soluções não são adequadas para problemas multimodais. Surge, assim, a necessidade de propor soluções de alto desempenho para problemas de otimização embarcada.

Neste sentido, é importante destacar que os sistemas embarcados comumente são projetados para um propósito definido e, portanto, pode-se usar a benefício a informação disponível sobre os requisitos que o sistema deve cumprir, assim como as restrições às que está sujeito [10]. Em geral, os sistemas embarcados são projetados para operar sobre restrições de portabilidade (tamanho e peso), consumo de recursos, desempenho, baixo consumo de energia e dissipação de potência [12]. Esta última restrição penaliza implementações com alta frequência de operação, portanto, novas soluções devem ser analisadas visando cumprir os requerimentos do produto desejado. Uma resposta a este quesito é o uso de soluções de *hardware* que explorem o paralelismo intrínseco dos algoritmos a serem embarcados, permitindo gerar implementações com bom desempenho que operam com baixas frequências de relógio.

Dispositivos FPGAs (*Field Programmable Gate Arrays*) são uma tecnologia que permite o desenvolvimento de sistemas embarcados baseados em arquiteturas paralelas, as quais podem ser implementadas diretamente em *hardware*, oferecendo soluções com alta capacidade de processamento. Portanto, dispositivos FPGAs são uma solução factível para explorar as capacidades de paralelismo dos algoritmos de otimização por inteligência de enxames, nos quais uma melhoria no desempenho pode ser alcançada tanto pela implementação de indivíduos em paralelo, como pela execução simultânea de operações, oferecendo assim um bom balanço entre o tempo de execução e custo em termos do consumo de recursos.

Outro ponto importante no projeto de sistemas embarcados é a necessidade de garantir uma boa precisão nas operações aritméticas, o que pode ser cumprido usando uma representação aritmética de ponto flutuante. Neste sentido, é importante realizar um estudo de factibilidade da implementação em FPGAs de unidades de cálculo aritmético e trigonométrico em ponto flutuante, visando atingir requisitos de precisão e faixa

dinâmica. Estes estudos devem ser direcionados visando analisar as diversas variáveis de projeto tais como tamanho de palavra, precisão do resultado, custo em área lógica, latência, frequência de operação e consumo de potência.

1.1 DESCRIÇÃO DO PROBLEMA

De forma geral, quando se soluciona um problema de otimização o objetivo é encontrar uma solução global ótima em uma quantidade de tempo aceitável. Muitos problemas a serem resolvidos pertencem à classe de problemas NP-difíceis, caracterizada pela impossibilidade de se encontrar a solução ótima em tempo polinomial.

O tempo de execução elevado da solução de algoritmos de otimização é um problema crítico, especificamente quando os algoritmos são aplicados em sistemas embarcados, nos quais normalmente são utilizados microprocessadores que operam a frequências baixas se comparado com processadores de propósito geral (GPPs). Neste sentido, existe um novo campo de aplicação, conhecido como *otimização embarcada*, em que os sistemas não podem suspender o seu funcionamento no intuito de resolver um problema de otimização com um elevado tempo de execução. Sistemas de alocação de recursos em tempo real, processamento de sinais, filtros adaptativos, síntese de som, sistemas de navegação automática, processos de controle adaptativo, aprendizado de máquinas, robótica móvel e de manipuladores são exemplos de aplicações que utilizam sistemas de otimização embarcados. A figura 1.1 descreve a dimensionalidade dos problemas a serem resolvidos no campo de sistemas de otimização embarcada em função do tempo de execução. Observe-se que os problemas devem ser resolvidos em uma faixa de tempo da ordem dos milissegundos, sendo que a dimensionalidade dos problemas (número de variáveis de decisão) é relativamente pequena (algumas dezenas).

Os algoritmos de otimização por inteligência de enxames possuem uma capacidade de paralelização inerente que pode ser utilizada para o melhoramento do tempo de execução e da qualidade das soluções. No contexto deste trabalho, a qualidade das soluções é medida em termos da aproximação à solução ótima. Uma revisão bibliográfica do estado da arte mostra que são poucos os estudos que têm sido realizados visando paralelizar estas técnicas de otimização, principalmente, quando aplicadas a problemas de otimização embarcada.

É com base na análise das técnicas de otimização por inteligência de enxames e com foco

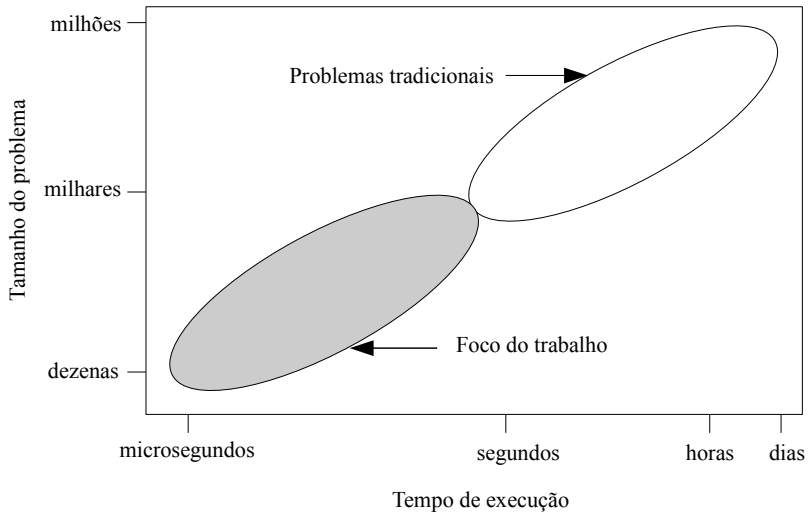


Figura 1.1: Foco do trabalho (Modificado Boyd, 2009 [13])

em soluções simples baseadas em dispositivos FPGAs para sistemas embarcados, que proporcionem melhorias em desempenho, que o presente estudo tem a sua motivação.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

O objetivo geral deste trabalho é propor algoritmos paralelos de otimização baseados em inteligência de enxames, aplicados a problemas em sistemas embarcados utilizando arquiteturas reconfiguráveis, visando analisar o ganho de desempenho em termos de velocidade de execução e da qualidade das soluções obtidas.

1.2.2 Objetivos específicos

Espera-se alcançar os seguintes objetivos específicos:

- Realização do estudo teórico de algoritmos de otimização bioinspirados baseados em inteligência de enxames.
- Projeto, implementação e análise de arquiteturas paralelas dos algoritmos escolhidos usando arquiteturas reconfiguráveis.

- Obtenção de dados empíricos que demonstrem a viabilidade dos algoritmos propostos para a solução de problemas de otimização global e local em sistemas embarcados.
- Explorar a potencialidade das arquiteturas e algoritmos propostos na solução de problemas de otimização embarcada.

1.3 ASPECTOS METODOLÓGICOS DO TRABALHO

A metodologia utilizada no desenvolvimento do presente trabalho é composta de seis etapas, a saber:

- Na primeira etapa foi realizada uma revisão bibliográfica do estado da arte dos algoritmos de otimização por inteligência de enxames, levando em conta aspectos de implementação *hardware* e *software*, assim como possíveis aplicações em sistemas embarcados.
- Na segunda etapa foram feitas as implementações em *hardware* das bibliotecas parametrizáveis de cálculo aritmético e trigonométrico em ponto flutuante. A escolha da aritmética de ponto flutuante para representação numérica justifica-se levando em conta os requerimentos de precisão e alta faixa dinâmica presentes nos problemas de otimização em diversos campos da engenharia. Desta forma, aplicações embarcadas podem usufruir de cálculos de alto desempenho em termos de precisão, faixa dinâmica, tempo de execução e consumo de potência.

  É importante salientar que fabricantes de FPGAs fornecem núcleos de propriedade intelectual (IPcores), entre os quais é possível encontrar bibliotecas para cálculo aritmético em ponto flutuante. No entanto, neste trabalho foi descartada esta possibilidade pelos seguintes motivos: (a) os IPcores são exclusivos para algumas famílias de um mesmo fabricante, dificultando a portabilidade das implementações; (b) tipicamente os IPcores consomem muitos recursos de *hardware*; (c) não todos os operadores trigonométricos estão implementados nos IPcores dos fabricantes de FPGAs; (d) as arquiteturas são fixas, impedindo realizar ajustes nos parâmetros internos dos algoritmos, ou mesmo usar outras abordagens de cálculo numérico, no intuito de melhorar o erro associado. Portanto, neste trabalho foi decidido desenvolver as bibliotecas para cálculo aritmético em ponto flutu-

ante, de forma que as mesmas possam se ajustar apropriadamente às aplicações almejadas pelo grupo de pesquisa do GRACO-ENM-UnB.

- Na terceira etapa foram caracterizados os operadores implementados. Esta etapa envolveu uma análise do compromisso entre o tamanho da palavra da representação em ponto flutuante, o erro associado, o consumo de recursos e o tempo de execução. Para isto, foi necessário realizar uma análise comportamental dos algoritmos de cálculo aritmético e trigonométrico em função dos seus parâmetros de ajuste (número de iterações, número de sementes iniciais, etc.). Por outro lado, comparações numéricas do erro associado entre as implementações de *hardware* e *software* foram realizadas para validar as arquiteturas propostas. Para isto, os resultados do Matlab foram usados como estimador estatístico.

- Na quarta etapa foram realizadas as implementações de *hardware* das arquiteturas paralelas dos algoritmos de otimização por inteligência de enxames. Para isto foram usadas as bibliotecas de cálculo aritmético e trigonométrico, previamente implementadas e caracterizadas. Funções de teste (*benchmark*) unimodais e multimodais foram utilizadas para propósitos de validação das arquiteturas propostas. Adicionalmente, uma análise comparativa do custo em área lógica no circuito, tempo de execução e qualidade das soluções foi realizada.

- Na quinta etapa foram analisadas algumas técnicas de adição de diversidade artificial visando a melhoria das soluções obtidas. A escolha das técnicas de adição de diversidade foi direcionada pelo uso de operações pouco complexas, obedecendo à filosofia de fácil implementação (*hardware* ou *software*) dos algoritmos de inteligência de enxames, permitindo economizar recursos computacionais.

  Implementações em *software* utilizando processadores de propósito geral permitiram realizar uma análise comparativa da qualidade das soluções e do tempo de execução. Por outro lado, no intuito de validar as arquiteturas propostas para aplicações embarcadas, os resultados de convergência e tempo de execução obtidos pelas implementações de *hardware* foram comparados com as respectivas implementações de *software* utilizando um microprocessador embarcado mapeado no mesmo dispositivo FPGA, operando na mesma frequência de relógio.

- Na sexta e última etapa, a arquitetura com a melhor relação custo/benefício foi escolhida com base nos resultados obtidos. Esta arquitetura foi aplicada em um caso de estudo que requer da solução de problemas de otimização embarcada. A aplicação consiste no treinamento *online* de um controlador neural de um

robô móvel de pequeno porte movimentando-se em um ambiente de evasão de obstáculos.

1.4 CONTRIBUIÇÕES DO TRABALHO

Este trabalho originou o conjunto de publicações listadas no Apêndice A. As mesmas constituem o conteúdo deste documento, descrito nos capítulos 2 a 5. As principais contribuições são as seguintes:

1. *Implementação em hardware de unidades de cálculo aritmético e trigonométrico em ponto flutuante*: a primeira contribuição do trabalho foi o desenvolvimento de arquiteturas de *hardware* parametrizáveis para as operadores de soma, subtração e multiplicação usando uma representação aritmética de ponto flutuante. Os algoritmos *Goldschmidt's* (GS) e $Newton - Raphson$ (NR) foram utilizados para o desenvolvimento das arquiteturas iterativas para os operadores de divisão e raiz quadrada. Por outro lado, abordagens baseadas no algoritmo CORDIC e nas expansões por séries de Taylor foram utilizadas no desenvolvimento dos operadores de cálculo das funções trigonométricas *seno*, *cosseno*, $arco - tangente$ e da função exponencial.

   As implementações foram simuladas, posteriormente mapeadas em FPGAs e validadas usando o Matlab como estimador estatístico. Os experimentos demonstraram que a o algoritmo NR para divisão e raiz quadrada e o algoritmo CORDIC para as funções trigonométricas apresentam os melhores resultados de consumo de recursos de *hardware* assim como de precisão e exatidão. A implementação parametrizável das unidades de cálculo permitiu validar os resultados para diferentes tamanhos de palavra, entre elas a precisão simples (32 bits) e dupla (64 bits). Segundo os resultados obtidos, a melhor relação custo/benefício foi obtida para uma implementação de 27 bits (1 bit de sinal, 8 bits de expoente e 18 bits de mantissa). Portanto, a implementação final dos algoritmos de otimização por inteligência de enxames estão baseadas neste tamanho de palavra.

2. *Implementação em hardware de algoritmos de otimização por inteligência de enxames*: foram implementadas em FPGAs abordagens paralelas dos algoritmos de otimização por enxame de partículas (HPPSO - *Hardware Parallel Particle Swarm Optimization*), otimização por colônia de abelhas artificial (HPABC

- *Hardware Parallel Artificial Bee Colony Algorithm*), otimização por colônia de vaga-lumes (HPFA - *Hardware Parallel Firefly Algorithm*) e o algoritmo de otimização por embaralhamento de salto aleatório de sapos (HPSFLA - *Hardware Parallel Shuffled Frog Leaping Algorithm*). Considerando o estado da arte em computação evolucionaria, as três últimas implementações de *hardware* são inéditas. Algumas modificações foram aplicadas nos algoritmos originais ABC e FA no intuito de facilitar a sua implementação *hardware*, de forma que o paralelismo intrínseco dos mesmos pudesse ser explorado eficientemente. Neste sentido, foram propostos os algoritmos GBABC (*Global Bee Artificial Bee Colony*) e GFA (*Global Firefly Algorithm*), os quais fazem uso da informação social contida no indivíduo com o melhor valor de aptidão para melhorar a posição do resto de indivíduos do enxame.

3. *Implementação em hardware das técnicas de adição de diversidade artificial*: dentre as técnicas estudadas foram mapeadas em FPGAs aquelas baseadas em operadores aritméticos de fácil implementação, visando manter um baixo consumo de recursos de *hardware*. As técnicas atrativa-repulsiva, congregação passiva seletiva e de aprendizado em oposição (OBL - *Opposition based Learning*) foram inicialmente implementadas no algoritmo PSO e sua respectiva implementação em *hardware* constitui um aporte original na área de otimização bioinspirada. Com base nos dados de síntese e execução foi possível demonstrar que a técnica de aprendizado em oposição apresenta a melhor relação custo/beneficio e, portanto, foi aplicada nos outros três algoritmos bioinspirados.

   Neste sentido, foram propostos os algoritmos GBOABC (*Global Bee Opposition based Artificial Bee Colony*), GOFA (*Global Opposition based Firefly Algorithm*) e O-SFLA (*Opposition based Shuffled Frog Leaping Algorithm*), os quais exploram a capacidade de busca global da técnica OBL. A implementação em *hardware* destes algoritmos são também um aporte original deste trabalho.

4. *Aplicação das arquiteturas propostas em problemas de otimização embarcada*: com base nos resultados experimentais de consumo de recursos, qualidade da solução e tempo de execução, foi possível concluir que os algoritmos de otimização por inteligência de enxames podem ser mapeados em FPGAs viabilizando possíveis aplicações em sistemas de otimização embarcados.

   Neste sentido, uma contribuição adicional foi a aplicação da arquitetura HPOPSO (*Hardware Parallel Opposition-based Particle Swarm Optimization*) no treinamento supervisionado de redes neurais. Uma análise de viabilidade da arquitetura proposta para o ajuste *online* dos pesos de conexão do controlador neural

de um robô móvel autônomo foi realizada usando uma ferramenta de simulação de robôs móveis. Dois tipos de comportamento foram aplicados por meio de uma operação manual do robô. Os resultados obtidos mostram que a arquitetura proposta permite aproximar os modelos de comportamento, obtendo trajetórias similares às realizadas pelo operador humano. Adicionalmente, os experimentos realizados mostram que a arquitetura HPOPSO permite realizar o ajuste *online* do controlador neural na presença de falha em um ou mais sensores de medição de distância, visando garantir um comportamento específico.

5. *Contribuições adicionais*: em adição às contribuições acima mencionadas podem-se destacar as seguintes soluções de *software*.

   Uma ferramenta de geração automática de código VHDL foi desenvolvida no Matlab no intuito de facilitar a implementação *hardware* dos algoritmos de otimização por inteligência de enxames. Esta ferramenta, chamada de *Swarm Generator*, permite que o projetista selecione o tipo de algoritmo, o número de partículas, a dimensionalidade do problema de otimização, o tipo de técnica de adição de diversidade, o tamanho de palavra da representação em ponto flutuante, entre outros parâmetros de projeto. Isto facilita a sua implementação em FPGAs acelerando o tempo de desenvolvimento de um sistema de otimização embarcado.

   O desenvolvimento desta ferramenta de geração de código VHDL é justificada dado que pequenas modificações dos algoritmos, tais como ajustes do número de partículas paralelas, dimensionalidade dos problemas ou modificações nos parâmetros do problema de otimização, requerem uma grande quantidade de mudanças simples, porém repetitivas, nas arquiteturas de *hardware*. A grande quantidade de mudanças repetitivas demanda tempo e atenção por parte do projetista e, portanto, é um processo propenso a erros humanos. Contudo, a natureza essencialmente paralela dos algoritmos de inteligência de enxames permite que as arquiteturas de *hardware* possam ser implementadas usando-se estruturas regulares, facilitando a sua descrição de *hardware* e o desenvolvimento de ferramentas de geração de código VHDL.

   É importante ressaltar que a descrição *hardware* da função custo que modela o problema de otimização é específica de cada aplicação, portanto deve ser realizada pelo projetista. A ferramenta *Swarm Generator* fornece a descrição *hardware* do algoritmo de otimização e entrega um *template* com as entradas e saídas da entidade que irá descrever a função custo.

   Uma segunda contribuição baseada em *software* é a proposta do algoritmo OSP-

SOpc que faz uso de algumas das técnicas de adição de diversidade artificial estudadas no presente trabalho, tais como aprendizado em oposição, embaralhamento das soluções e congregação passiva seletiva. Este algoritmo híbrido foi aplicado na solução do problema de síntese de som FM com modulação aninhada simples e dupla.

Uma metodologia de avaliação de significância baseada nos testes de Wilconxon e Kruskal-Wallis foi aplicada para um nível de confiança de 95%. Desta forma, foi validada a diferença estatística entre as diferentes abordagens do PSO aplicadas no problema de síntese de som. Estes resultados permitem confirmar que o algoritmo híbrido proposto alcança melhores resultados em comparação com cada uma das abordagens de adição de diversidade do PSO.

Finalmente, foi desenvolvida uma ferramenta de *software* $vFPUgen$ para geração automática de código VHDL dos operadores aritméticos e trigonométricos usando ponto flutuante. Esta ferramenta permite configurar o tamanho de palavra da representação IEEE-754 e os parâmetros dos algoritmos, de forma que as arquiteturas implementadas cumpram requerimentos específicos de precisão, faixa dinâmica, custo em área lógica e desempenho. É importante destacar que as unidades de cálculo aritmético e trigonométrico desenvolvidas neste trabalho são usadas como blocos funcionais de diversos projetos baseados em plataformas reconfiguráveis no grupo de pesquisa GRACO-ENM-UnB. Entre estes projetos podem-se mencionar aplicações em algoritmos de inversão de matrizes, implementação de redes neurais artificiais, aceleração do cálculo da cinemática direta de robôs manipuladores e o projeto de filtros de Kalman para fusão de sensores e monitoramento de falhas em processos de soldagem.

1.5 ORGANIZAÇÃO DO TRABALHO

O presente trabalho divide-se da seguinte maneira: O Capítulo 2 apresenta a fundamentação teórica apresentando o conceito de otimização, a classificação das técnicas e metodologias comumente utilizadas na solução de problemas de otimização local e global. Posteriormente, são apresentados os conceitos da área de computação reconfigurável e tecnologias de implementação usando FPGAs. Finalmente, aborda-se uma revisão dos algoritmos de otimização por inteligência de enxames, apresentando os princípios biológicos utilizados pelos algoritmos, assim como as suas variantes, vantagens e desvantagens com relação a outras técnicas e implementações usando arquitetu-

ras paralelas. O Capítulo 3 descreve as implementações realizadas para os operadores de cálculo aritmético e trigonométrico, os resultados obtidos e uma análise dos mesmos. O Capítulo 4 descreve as implementações dos algoritmos paralelos HPPSO, HPSLFA, HPABC e HPFA, assim como sua variante usando o operador OBL (aprendizado baseado em oposição). Adicionalmente são apresentados os resultados de síntese, convergência e de comparação do tempo de execução. O capítulo 5 descreve a aplicação da arquitetura HPOPSO na solução de um problema de otimização embarcada que consiste no treinamento *online* de um controlador neural de um robô móvel. Finalmente, o Capitulo 6 apresenta as conclusões do trabalho.

Capítulo 2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta uma revisão das definições básicas relacionadas à otimização. Uma breve descrição dos problemas de otimização, a sua classificação e os métodos de solução mais utilizados. Posteriormente, são definidos os conceitos básicos relacionados com *hardware* reconfigurável e finalmente é abordado o tema de otimização por inteligência de enxames, apresentando uma revisão do estado da arte das implementações em arquiteturas paralelas e a nomenclatura utilizada neste trabalho.

2.1 OTIMIZAÇÃO

O conceito de otimização está relacionado à obtenção do melhor resultado dentre um conjunto finito ou infinito de possíveis soluções sujeitas a certas restrições [1]. Os processos de otimização oferecem um suporte à tomada de decisões em que o objetivo final é minimizar esforços ou maximizar o lucro. Em situações práticas os esforços requeridos ou os lucros desejados podem ser expressos usando uma função de certas variáveis de decisão. Desta forma, para que uma solução a um problema de otimização tenha significado, a mesma deve representar-se na forma de uma expressão matemática, chamada *função objetivo* (f), que considere uma ou mais *variáveis de decisão* (x_i), as quais devem ser definidas. O objetivo da otimização é encontrar os valores destas variáveis, de forma que a expressão matemática tenha o maior valor numérico possível (maximização) ou o menor valor numérico possível (minimização).

No contexto deste trabalho, todos os problemas de otimização são considerados de minimização, os quais podem ser definidos da seguinte maneira: Seja $f : \Re^n \rightarrow \Re$, encontrar $x^* \in \Re^n$, para o qual $f(x^*) < f(x), \forall x \in \Re^n$, i.e., x=(x_1,. . . ,x_n).

Entretanto, problemas de engenharia frequentemente apresentam não linearidades e interações complexas entre as variáveis do projeto x e as restrições do sistema, $f_i(x) \leq b_i$, com $f_i : \Re^n \rightarrow \Re$, $i = 1, \ldots, m$ e as constantes $b_1, \ldots, b_m$ sendo os limites das restrições, formando espaços de busca S, contendo em alguns casos várias soluções ótimas. As

técnicas para solução destes problemas podem ser divididas em duas categorias: *otimização local* e *otimização global*. Um ponto x^* é um *ótimo local* de f se existe uma vizinhança $V = x : |x^* - x| < \epsilon$ em torno de x^* tal que $f(x^*) < f(x), \forall x \in V \subset S \subseteq \Re^n$. Um ponto x^* é um ótimo global de f se $f(x*) < f(x), \forall x \in S \subseteq \Re^n$, sendo S o espaço de busca.

2.1.1 Elementos de um modelo de otimização

A solução de um problema de otimização requer como primeira instancia uma formulação matemática de um modelo que preserve uma equivalência coerente com o problema real. A abstração de um modelo de otimização envolve múltiplas etapas de tentativa e erro, comumente dependendo de critérios subjetivos, tais como experiência, criatividade e poder de síntese, que não podem ser regulados pelo estabelecimento de regras fixas. Os elementos básicos de um modelo de otimização são descritos a seguir:

1. *Variáveis de decisão:* são as incógnitas que definem a solução do modelo.
2. *Espaço de busca:* é o conjunto de pontos que representam as soluções factíveis e infactíveis ao problema de otimização. É determinado pelo limite superior e inferior das variáveis de decisão.
3. *Função objetivo:* é a função matemática que deve ser maximizada ou minimizada e que representa o problema de otimização e as interações entre as variáveis de decisão. Também é referenciada como *função custo*. O valor da função objetivo avaliada em uma possível solução é conhecido como *valor de aptidão*.
4. *Restrições:* funções matemáticas que limitam o espaço de soluções factíveis do problema de otimização. São determinadas pelas limitações físicas do sistema.
5. *Ótimo local:* é um ponto máximo ou mínimo que ocorre em um subespaço do espaço de busca.
6. *Ótimo global:* é o ponto do espaço de busca onde a função objetivo alcança o valor máximo ou mínimo.

Problemas que envolvem muitos ótimos locais são chamados de *multimodais*. Entretanto, problemas com um único ótimo global são chamados de *unimodais* ou *problemas convexos*.

2.1.2 Complexidade

Uma análise de complexidade pode ser efetuada tanto nos problemas de otimização quanto nos algoritmos que resolvem estes problemas. Nesta seção discute-se o primeiro tópico, apresentando as classes de complexidade de problemas de otimização e os métodos de classificação destes. A medida de complexidade dos problemas de otimização é um fator importante que direciona o projeto da solução final, identificando a dificuldade relativa para resolvê-los, permitindo limitar os recursos de tempo e armazenamento necessários para a solução [15].

A complexidade de um algoritmo pode ser expressada em função das operações fundamentais e do volume de dados que são tratados, as quais influenciam diretamente no tempo de resolução de um problema. Desta forma, a complexidade de um problema de otimização é definida como uma função da dimensionalidade do problema, representada pelo número de variáveis de decisão ou tamanho de entrada do problema [16].

É importante salientar que a teoria de complexidade foi desenvolvida para problemas de decisão e não para problemas de otimização. Em problemas de decisão existem apenas duas possíveis soluções: *Sim* ou *Não*. No entanto, segundo Mendes [16], um problema de otimização sempre tem instancias de problemas de decisão que são iguais ou mais difíceis de serem resolvidas. Um problema de otimização pode ser classificado segundo a sua complexidade da seguinte maneira:

1. *Classe P:* um problema de otimização pertence à classe P se existe um algoritmo cujo tempo de execução é uma função polinomial das entradas do problema, capaz de encontrar a solução ótima. Isto é, o tempo de execução é polinomial de ordem k, o que pode ser expressado como $O(n^k)$, sendo k uma constante e n o número de entradas do problema. Estes tipos de problemas se caracterizam porque a relação entre o algoritmo e a dimensionalidade do problema é uma função polinomial e, portanto, podem ser resolvidos por uma máquina de $Turing$ determinística em tempo polinomial.

2. *Classe NP:* a classe de complexidade NP é o conjunto de problemas *verificáveis* em tempo polinomial. O termo *verificável* indica que, para uma instância do problema e uma solução candidata, chamada de *certificado*, existe um algoritmo de verificação que fornece como saída *sim* ou *não* dependendo se o certificado é ou não uma solução do problema. Um problema de otimização é considerado

da classe NP se existe um algoritmo cujo tempo de execução é uma função polinomial das entradas do problema, isto é $O(n^k)$, capaz de avaliar (ou verificar) o valor da função custo para qualquer solução válida. No entanto, a principal diferença com os problemas de otimização da classe P é que não existem algoritmos determinísticos que possam resolver problemas de otimização NP em tempo polinomial. É importante destacar que embora as classes P e NP são infinitas, a cardinalidade dos problemas NP é maior do que a cardinalidade dos problemas P, e consequentemente, existem mais problemas de otimização da classe NP.

3. *Classe NP-completo:* são um subconjunto dos problemas NP, caracterizados por ser a classe dos problemas mais difíceis de NP. Neste caso os algoritmos de resolução são não polinomiais e, portanto, intratáveis por máquinas determinísticas. Além disso, se pelo menos um dos problemas da classe NP-completo for solúvel em tempo polinomial, todos da classe o serão. Esta questão é uma das mais fundamentais da área de pesquisa em complexidade de algoritmos. É importante destacar que os problemas de otimização NP-completos costumam ter soluções subótimas úteis, que são calculadas por algoritmos polinomiais ou por algoritmos não determinísticos ou uma mistura das duas técnicas.

4. *Classe NP-difícil:* subtraindo da classe NP os problemas NP-completos, encontra-se a classe de problemas NP-difícil. Diz-se então, que todo problema NP-difícil é pelo menos tão complexo quanto qualquer problema NP-completo, e todo problema NP-completo é NP-difícil, porém o último pode conter problemas não verificáveis em tempo polinomial. Nesta classe, não existem algoritmos polinomiais para encontrar soluções subótimas. A maioria dos problemas de otimização combinatória são do tipo NP-difícil e, portanto, requerem de métodos de solução não determinísticos.

2.1.3 Classificação dos métodos de otimização

Diversas técnicas de otimização têm sido propostas pela comunidade cientifica, sendo divididas em três categorias: (a) métodos exatos, (b) métodos heurísticos e (c) metaheurísticos.

2.1.3.1 Métodos exatos

Os métodos exatos geralmente utilizam algoritmos determinísticos para realizar uma busca exaustiva do espaço de soluções no intuito de encontrar o ótimo global da função objetivo. Técnicas de programação matemática como o algoritmo *Simplex* para programação linear, métodos baseados em derivadas para programação não linear ou algoritmos de programação dinâmica fazem parte dos métodos exatos. Em muitos problemas de otimização a relação entre as soluções candidatas e o valor de aptidão é complexa, fazendo com que o problema seja difícil de resolver de forma determinística. Adicionalmente, problemas de alta dimensionalidade conduzem a tempos de execução elevados (horas ou dias) e, portanto, algoritmos não determinísticos devem ser considerados [1].

Antes de discutir os métodos de otimização não exatos, é necessário introduzir o conceito de *máquina probabilística*. Uma máquina probabilística pode ser definida como aquela em que as transições entre estados dependem além do estado atual, também de uma escolha *aleatória* para decidir o próximo passo de computação. Algoritmos que implementam máquinas probabilísticas são conhecidos como *algoritmos probabilísticos*, cujo comportamento é guiado pela probabilidade de ocorrência de um evento com uma distribuição pré-definida.

2.1.3.2 Métodos heurísticos

São métodos que utilizam uma ou mais informações referentes ao problema para guiar o processo de busca de uma solução ótima [17]. Os métodos heurísticos não garantem encontrar a solução ótima de um problema, porém, podem encontrar soluções aceitáveis (subótimas) em um tempo polinomial. A principal desvantagem dos métodos heurísticos é que são desenvolvidos para resolver classes específicas de problemas, carecendo de generalidade.

2.1.3.3 Métodos metaheurísticos

Metaheurísticas são métodos de generalização de heurísticas que permitem resolver diversos tipos de problemas sem precisar grandes mudanças nos algoritmos. Estes

métodos geralmente utilizam estatísticas obtidas de amostras do espaço de busca que permitem resolver uma ampla gama de problemas, sendo comum a utilização de modelos baseados em fenômenos naturais ou processos físicos no intuito de aplicar uma filosofia que permita o desenvolvimento de algoritmos heurísticos. Entre as técnicas de otimização metaheurísticas mais utilizadas podem-se mencionar os *Algoritmos Genéticos* [18], *Recozimento Simulado* [19], *Otimização por Colônia de Formigas* [20], *Otimização por Enxame de Partículas* [21], entre outros. Os métodos de otimização metaheurísticos são o foco do presente trabalho.

2.1.4 Algoritmos de otimização bioinspirados

Nas últimas décadas tem surgido uma nova ciência computacional baseada em metaheurísticas inspiradas na natureza, na biologia e em processos físicos. Esta ciência denominada *computação natural* [22] compreende áreas de atuação em processos de otimização, inteligência artificial (redes neurais artificiais, lógica nebulosa, etc.), biocomputação para análise de DNA, fractais, entre outros.

Algoritmos de otimização bioinspirados baseados em populações, dentre os quais se destacam os algoritmos evolutivos e os algoritmos de enxames, fazem parte dos métodos de computação natural. Estes algoritmos utilizam técnicas computacionais baseadas nos princípios biológicos evolutivos encontrados na natureza, tais como a seleção natural e herança genética, mutação e comportamentos coletivos para intercâmbio de informação [4]. Estas técnicas consideram algoritmos probabilísticos que fornecem mecanismos de busca adaptativa baseados no princípio Darwiniano da sobrevivência dos mais aptos, visando refinar o conjunto de soluções candidatas a um problema de otimização [23], [24].

A figura 2.1 apresenta uma classificação dos algoritmos de otimização bioinspirados (baseados em populações) mais utilizados e aplicados em diversos campos do conhecimento [25], [22]. Os *Algoritmos Evolutivos* incluem os algoritmos genéticos, a programação evolutiva, estratégias evolutivas e a programação genética [26], [27]. Estas técnicas utilizam operadores matemáticos que se aproximam aos conceitos genéticos no intuito de simular o principio de sobrevivência dos mais fortes, reproduzindo apenas os indivíduos (possíveis soluções) que se aproximam à solução ótima de um problema. Por outro lado, os *Algoritmos de Enxames* consideram um conjunto de técnicas baseadas no comportamento coletivo de algumas espécies naturais, termo melhor referenciado como

Inteligência de Enxames, permitindo um intercâmbio eficiente de informação entre os indivíduos de uma colônia. Exemplos do sucesso deste tipo de comportamento social é a descoberta do caminho ótimo percorrido pelas formigas, o agrupamento de cardumes de peixes e bandos de aves na procura de comida e a estrutura organizacional das abelhas. Este tipo de algoritmos fornecem características favoráveis para aplicações em sistemas auto-organizados, flexíveis e dinâmicos [27]. O presente trabalho tem foco no segundo tipo de algoritmos.

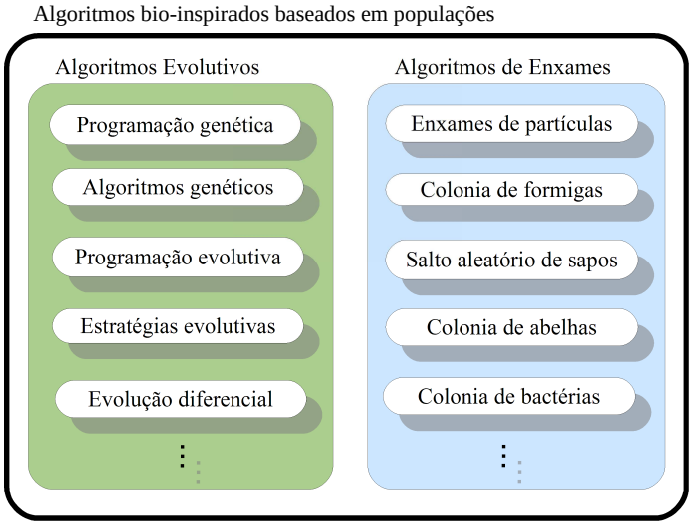


Figura 2.1: Classificação das metaheurísticas bioinspiradas [25], [22]

2.1.5 Inteligência de enxames

O termo inteligência de enxames (*swarm intelligence*) foi utilizado inicialmente para resolver problemas de sistemas auto-organizados, utilizando células de trabalho robotizadas, visando desenvolver máquinas com inteligência distribuída com capacidades adaptativas e de auto-organização [28], [29]. Posteriormente, a definição do termo *inteligência de enxames* foi estendida no intuito de abordar não apenas os sistemas robóticos, senão também qualquer algoritmo ou método de solução de problemas inspirado no comportamento social de espécies animais [20].

Desde um ponto de vista evolutivo, a inteligência pode ser interpretada como um mecanismo de sobrevivência e adaptação tanto em humanos como em algumas espécies animais. A figura 2.2 apresenta as formas de inteligência identificadas na natureza. Seres humanos e alguns grupos de animais utilizam a inteligência individual, produto

da sua alta capacidade de raciocínio, para solucionar problemas complexos ou realizar tarefas cotidianas. Entretanto, insetos, anticorpos e grupos de bactérias manifestam interações sociais oriundas de uma inteligência distribuída. Neste caso, os indivíduos possuem pouca capacidade de raciocínio para realizar tarefas complexas, porém têm altas capacidades de auto-organização, permitindo-lhe à colônia utilizar mecanismos de colaboração. Tais mecanismos de colaboração são usados para desenvolver *simultaneamente* diferentes atividades como, por exemplo, aprendizado, exploração, busca de alimento, transporte, defesa, reprodução, entre outras.

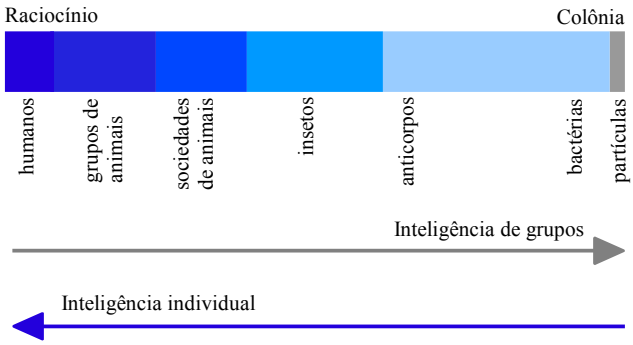


Figura 2.2: Formas de inteligência na natureza

Nos modelos computacionais baseados em inteligência coletiva, o termo *exame* pode ser aplicado de uma forma mais generalizada, sendo entendido como qualquer tipo de comportamento coletivo. Desta forma, os enxames podem ser aplicados em espaços de alta dimensionalidade em que as colisões entre os agentes (ocupação do mesmo espaço no mesmo instante de tempo) não têm maior importância, como poderia acontecer no mundo natural. No entanto, dita colisão pode ser entendida como uma agregação matemática e, portanto, mantém um significado probabilístico [30].

Diversos algoritmos que visam imitar o comportamento social de agentes têm sido propostos. Estes algoritmos podem ser utilizados para simular interações sociais entre indivíduos, podendo ser aplicados para o intercambio de informação aplicado na resolução de problemas de otimização [16].

2.2 *HARDWARE* RECONFIGURÁVEL

Desde a introdução do primeiro microprocessador comercial, o Intel 4004, no final do ano 1971, os sistemas digitais têm evoluído de forma considerável. Este pequeno mi-

croprocessador integrava 2300 transistores em uma única pastilha de silício cujo custo inicial oscilava os 200 dólares americanos. A complexidade dos microprocessadores, medida segundo o número de transistores dentro do *chip*, é dobrada cada 18 meses desde a aparição do 4004 [31]. Esta observação, conhecida como *lei de Moore*, tem se mostrado válida para qualquer gama de circuitos digitais. Com a evolução da tecnologia os circuitos integrados conseguem atualmente integrar cada vez mais transistores, atingindo alguns bilhões no ano de escrita do presente documento.

Uma classificação dos circuitos integrados que permitem a implementação de uma lógica digital é mostrada na figura 2.3.

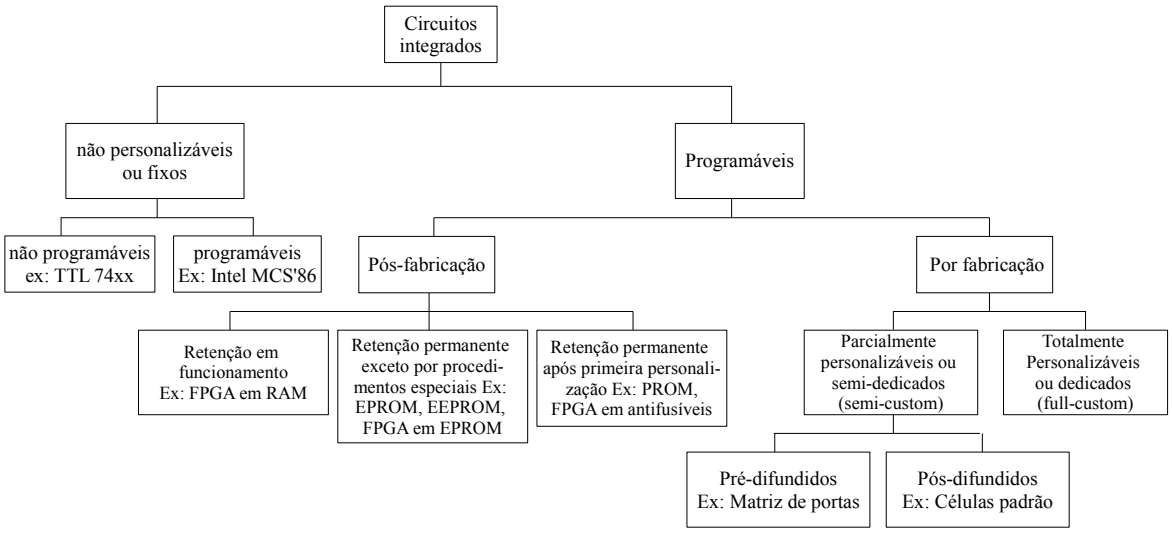


Figura 2.3: Taxonomia para CIs (modificado Calazans, 1998 [15])

Os circuitos integrados não personalizáveis executam uma lógica padrão realizando operações pré-definidas pelo fabricante; portanto, o usuário deve conectar diferentes tipos de circuitos para executar uma lógica específica, trazendo inconvenientes como perda de desempenho, aumento da área e alto consumo de energia.

A tecnologia VLSI (*Very Large Scale Integration*), definida como uma tecnologia de integração de alta escala, permite ao usuário determinar a lógica a ser implementada em um circuito. Entretanto, os circuitos são implementados por fabricantes especializados utilizando processos de fabricação de alta tecnologia, atendendo requerimento de baixo consumo de potência e alto desempenho. Exemplos de tecnologias VLSI são microprocessadores e memórias RAM utilizadas em PCs. As principais desvantagens da tecnologia VLSI são o custo de implementação elevado e a funcionalidade fixa, impossibilitando a execução de *upgrades* ou modificações na lógica que vai ser executada.

Entre os circuitos integrados projetados usando a tecnologia VLSI destacam-se os ASICs (*Application Specific Integrated Circuits*) nos quais os circuitos integrados são personalizávies, porém, construídos para tarefas específicas. Entre as metodologias de projeto de ASICs podem-se mencionar o projeto por células padrão (*standard cells*), matriz de portas (*gate arrays*), projeto estruturado (*structured array*) e projeto totalmente customizado (*full-custom design*) [32], [33]. Por outro lado, o tempo de projeto e fabricação de ASICs eleva o custo de desenvolvimento, justificado o investimento para uma grande quantidade de circuitos integrados. Contudo, este tipo de tecnologia alcança velocidades de processamento altas em uma área reduzida, assim como um baixo consumo de energia.

Dispositivos PLDs (*Programmable Logic Device*), CPLDs (*Complex Programmable Logic Device*) e FPGAs (*Field Programmable Gate Arrays*) possuem operações funcionais internas definidas pelo usuário após o processo de fabricação. Estes dispositivos são construídos a partir de um arranjo matricial de elementos lógicos reprogramáveis, permitindo implementar, teoricamente, qualquer tipo de circuito digital. Isto faz com que empresas especializadas em fabricação de circuitos integrados utilizem FPGAs na fase de projeto, prototipação e verificação [34]. Apesar da flexibilidade que oferecem estes dispositivos, os mesmos apresentam desvantagens de desempenho, área, consumo de potência e tempo de configuração [35].

2.2.1 Arquiteturas *von Neumann*

Os processadores e microcontroladores encontrados no mercado possuem uma arquitetura convencional tipo *von Neumann*, na qual uma unidade lógica aritmética (ULA) executa uma única instrução por vez dentro de um conjunto de instruções, armazenando e lendo dados em uma memória RAM, vide figura 2.4. No modelo de *von Neumann* o fluxo de instruções trafega entre o processador e a memória, levando ao uso de registradores especiais para seu controle, por exemplo, por meio do *contador de programa*. Este tipo de arquitetura formulada na década de 1960 constituiu durante anos o dispositivo computacional mais flexível sobre o qual diversas aplicações têm sido realizadas [36]. Outras arquiteturas de processadores têm sido propostas e eficientemente aplicadas, porém, a maioria delas são baseadas no mesmo principio de acesso a memórias de instruções e memórias de dados.

Embora o uso massivo do modelo de *von Neumann* na computação, o mesmo apresenta

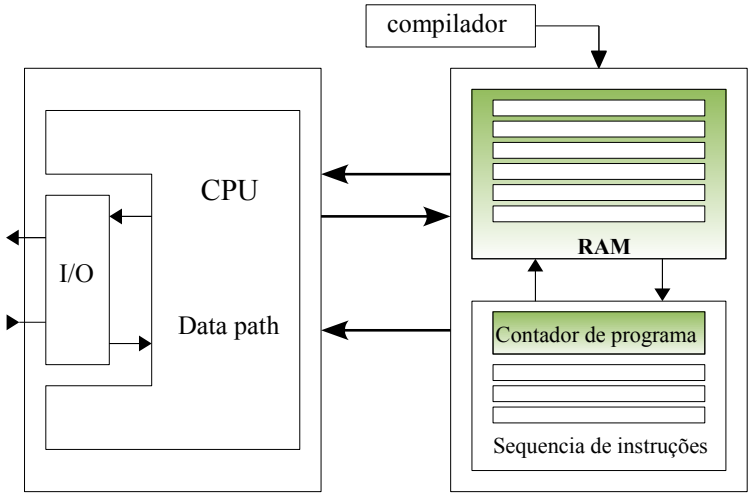


Figura 2.4: Arquitetura *von Neumann*

uma série de restrições. A primeira delas é originada pela disparidade de desempenho entre a velocidade de acesso das memórias e o processamento de dados na ULA. A evolução da indústria de memórias de acesso aleatório (RAM) tem crescido consideravelmente em desempenho e capacidade de armazenamento; no entanto, a velocidade funcional das memórias, isto é, a velocidade com que são escritos ou lidos os dados na memória, não consegue acompanhar a velocidade com que são processados os dados na ULA. Esta disparidade de desempenho, conhecida como *memory wall* [37], forma hoje em dia um dos maiores inconvenientes na computação de alto desempenho, em que arquiteturas computacionais convencionais se tornam ineficientes à medida que os processadores alcançam maiores desempenhos em termos do número de instruções executadas por unidade de tempo. A execução sequencial (intrínseca no modelo), a limitação da transferência de dados entre memória e processador (devido a limitações dos barramentos, especialmente no barramento de dados), assim como a barrera de memória (*memory wall*) determinam o denominado *"gargalo de von Neumann"*.

Os processadores baseados em arquiteturas *von Neumann* têm evidenciado, durante décadas, fatores de crescimento de desempenho favoráveis. Por outro lado, simultaneamente ao crescimento de desempenho dos processadores, observaram-se crescimentos, algumas vezes mais acelerados, dos requerimentos de largura de banda e complexidade computacional das aplicações [9]. A dinâmica destes fatores pode ser resumida nas seguintes formulações: (a) *lei de Moore*: estabelece que o número de transistores em um circuito integrado é dobrado a cada 18 meses, (b) *lei de Shannon*: estabelece que a complexidade algorítmica implementada em processadores é dobrada a cada 14 meses, (c) *lei de Nilsen*: estabelece que os requerimentos de largura de banda é dobrada a

cada 12 meses e (d) *lei de Franz Rammig*: estabelece que o *software* embarcado em processadores é dobrado a cada 10 meses [9].

Adicionalmente, existem limites físicos e termodinâmicos que impedem aos processadores aumentar ainda mais a sua densidade computacional ou as frequências de operação. O limite termodinâmico foi atingido há poucos anos, enquanto o limite físico da capacidade de integração dos circuitos integrados está próximo a ser alcançado [36]. Neste sentido, a exploração de arquiteturas computacionais que evitem o aceso sequencial de memórias, explorando assim o paralelismo intrínseco dos algoritmos é um quesito importante na computação de alto desempenho.

2.2.2 FPGAs

Internamente os FPGAs contém copias do mesmo *bloco lógico configurável* (LB) organizados matricialmente. Cada bloco lógico configurável contém um número pequeno de entradas e uma saída, e dentro dele são encontradas pequenas células formadas por um ou dois *flip-flops* onde é possível armazenar valores de '0' ou '1'. Os tipos de blocos lógicos mais comumente encontrados são baseados em LUTs (*LookUp Tables*) que, por meio do controle de um grupo de multiplexadores e portas, permitem o fluxo de dados desde as células de armazenamento até a saída do bloco lógico, permitindo assim a implementação de funções lógicas.

O arranjo de células dentro da LUT é utilizado para armazenar a *tabela verdade* das funções lógicas. A figura 2.5(a) mostra a estrutura de um bloco lógico da FPGA StratixII da *Altera* [34]. A figura 2.5(b) mostra um exemplo de LUT de duas entradas e uma saída.

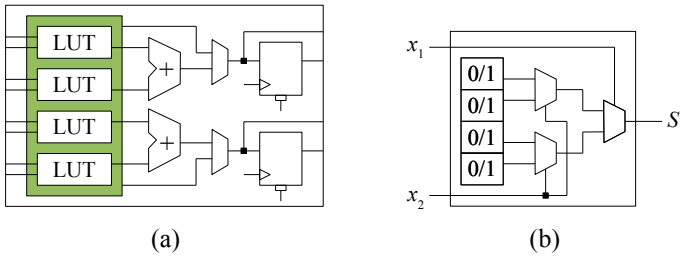


Figura 2.5: (a) Bloco lógico Altera StratixII (b) LUT de duas entradas e uma saída

Para realizar operações complexas, os elementos lógicos podem ser conectados uns com

outros através de chaves de interconexão programáveis. Na figura 2.6(a) é apresentada a estrutura geral de um FPGA, contendo os seguintes três tipos de recursos: (a) blocos de I/O, utilizados para realizar a interface com os pinos de entrada e saída do dispositivo; (b) blocos lógicos para implementação de funções por meio da programação das células de armazenamento; (c) rede de interconexão para roteamento horizontal e vertical entre as linhas e colunas dos blocos lógicos.

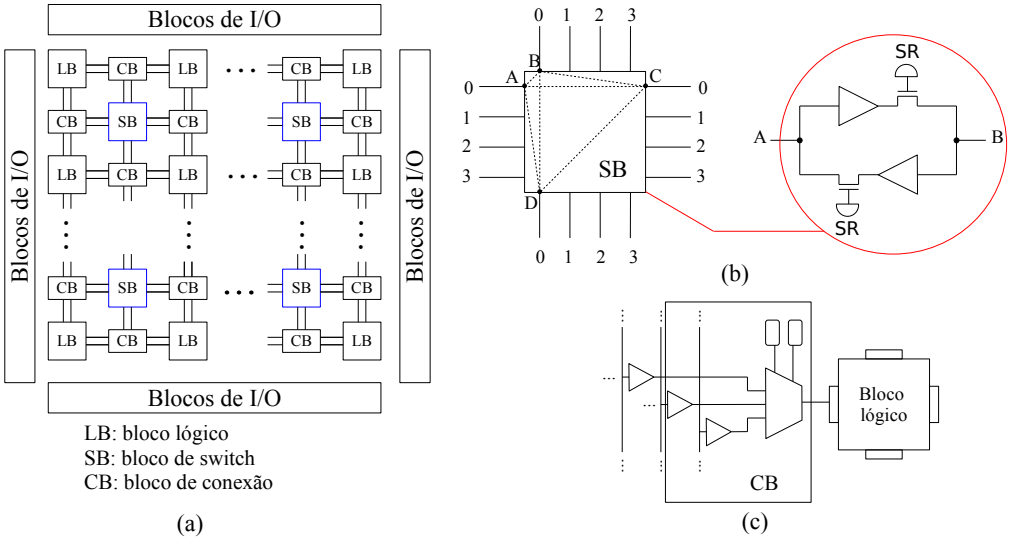


Figura 2.6: (a) Estrutura geral da FPGA (b) bloco de switch (c) bloco de conexão

A figura 2.6(b) mostra a estrutura dos *blocos de switch*, utilizados para conectar blocos lógicos não vizinhos. Adicionalmente, a figura 2.6(c) mostra a estrutura dos *blocos de conexão*, que comunicam elementos lógicos vizinhos. Nos FPGAs de última geração os canais de roteamento entre os componentes podem existir em planos diferentes. A figura 2.6 representa uma estrutura de um 2D-FPGA, no qual os blocos de *switch* estão em um plano diferente aos blocos de conexão e blocos lógicos. Em outros FPGAs utilizam-se três planos (3D) o que permite um roteamento mais eficiente, tempos de propagação menores entre os elementos lógicos e diminuição do consumo de potência [38].

As arquiteturas computacionais baseadas em CLPDs ou FPGAs, possibilitam a configuração dos seus blocos lógicos executando funções lógicas por meio de um *fluxo de dados*, sem a necessidade de ler sequências de instruções (como realizado em processadores de propósito geral). Por outro lado, a transferência de informações não depende da velocidade com que podem ser lidos ou escritos os dados dentro de uma memória.

2.2.3 Vantagens da computação reconfigurável

Desde um ponto de vista operacional, um FPGA é um tipo de dispositivo reconfigurável que permite realizar, em um ciclo de relógio, tarefas que para um processador tipo *von Neumann* podem exigir dezenas ou centenas de ciclos de relógio [39]. É importante salientar que no modelo de processador clássico apenas o algoritmo é variável, enquanto os recursos de *hardware* são fixos (*hardwired*). Entretanto, no modelo de computação reconfigurável ambos o algoritmo e os recursos de *hardware* podem ser programáveis. Neste sentido, existem dois conceitos importantes: (a) *flowware* em que existe uma programação dos fluxos de dados, em lugar de um fluxo de instruções, os quais são controlados por contadores e sequenciadores de dados, permitindo modificar a lógica implementada; (b) *configware* em que existe uma programação estrutural dos recursos usando ferramentas de mapeamento (*placement and routing*) e, portanto, permitem a realocação e reconfiguração do *hardware* [9].

Para trabalhar com o modelo de fluxo de dados (*flowware*), os algoritmos são previamente mapeados nos blocos lógicos do dispositivo FPGA. Neste tipo de dispositivos configuráveis os fluxos de dados podem ser vistos como tarefas implementadas espacialmente constituídas de operadores simples, os quais são ligados uns aos outros, através de fios elétricos. Em contraste, nos processadores tradicionais, as tarefas ou instruções são sequenciadas temporalmente, usando registros ou memórias para armazenar os resultados intermédios [39]. A consequência mais significativa deste tipo de abordagem é um ganho significativo no desempenho, se comparado com processadores convencionais, devido à enorme redução de ciclos de relógio necessários para acessar às memórias. Por outro lado, nos FPGAs os blocos lógicos podem ser configurados para permitir um fluxo de dados de forma independente e paralela. Isto possibilita a execução, em uma única pastilha, de diferentes processos concorrentes, aumentando a quantidade de dados processados por segundo (*throughput*) e, portanto, melhorando o desempenho dos algoritmos implementados em relação às soluções baseadas em *software* [40].

Dispositivos FPGA modernos contêm, além dos blocos lógicos configuráveis, arranjos de blocos dedicados para processamento digital de sinais (DSP - *Digital Signal Processing*), os quais disponibilizam recursos de multiplicadores, somadores e acumuladores. Adicionalmente, dispositivos FPGAs também fornecem blocos de memória RAM, blocos de controle do relógio usando PLLs ou DLLs, entre outras funcionalidades. Tendo em vista que a maioria dos algoritmos de processamento de sinais possuem um consumo massivo de multiplicadores e acumuladores, os FPGAs são uma solução factível,

eficiente e econômica para a implementação deste tipo de algoritmos [41].

O projeto de sistemas embarcados de alto desempenho é um desafio que se apresenta, hoje em dia, na área de sistemas digitais. Neste sentido, os dispositivos FPGAs podem aproveitar a sua flexibilidade e capacidade de processamento paralelo, fazendo uso de processadores de *software* e ferramentas correlatas para implementar sistemas embarcados com alto desempenho e baixo consumo de potência [9]. Aplicações na indústria automotiva, dispositivos médicos, redes sem fio, multimídia, processamento de imagens e vídeo, processamento digital de sinais, robótica de manipuladores e robótica móvel, são exemplos de aplicações em que os FPGAs têm mostrado resultados promissórios no âmbito do projeto de sistemas embarcados [42], [9], [43].

Neste contexto, o consumo de potência tem passado a ser um importante critério de projeto de circuitos, especificamente para aplicações portáveis [44]. Em adição à flexibilidade e aos ganhos de desempenho, outra característica chamativa dos dispositivos FPGAs é o seu baixo consumo de potência se comparado com soluções de *software* usando computadores convencionais [9]. No entanto, o consumo de potência dos FPGAs, continua sendo maior se comparado com soluções baseadas em ASICs [45], ou soluções baseadas em processadores RISC (*Reduced Instruction Set Computer*) [46]. Portanto, existe um compromisso fundamental que deve ser resolvido pelo projetista quando trabalha na área de sistemas embarcados: um dispositivo FPGA pode demandar um consumo de energia maior do que um único processador RISC, porém permite melhoras significativas no desempenho computacional.

Como explicado na figura 1.1, o foco do presente trabalho se concentra em sistemas de otimização embarcados com restrições de baixa dimensionalidade (na ordem de dezenas) e tempos de execução na ordem de milissegundos até alguns segundos. Aplicações como planejamento/replanejamento de trajetória de robôs móveis, robótica multiagente, visão computacional, redes de sensores distribuídas, estimação de parâmetros em tempo real, sintonização de parâmetros em tempo real para processos industriais, treinamento *online* de redes neurais artificiais em sistemas portáveis, entre outros, são exemplos de aplicações embarcadas em que frequentemente etapas de otimização devem ser realizadas visando melhorar os resultados e/ou a tomada de decisões. Todavia, este tipo de aplicações exigem que os algoritmos de otimização sejam executados cumprindo requerimentos de alto desempenho em termos de tempo de execução e precisão dos cálculos, assim como baixo consumo de potência.

No contexto do presente trabalho os dispositivos FPGAs são utilizados para explorar as capacidades de paralelismo intrínseco dos algoritmos de otimização por inteligência de enxames, visando diminuir o tempo de execução e aumentar a qualidade das soluções, mantendo um consumo de potência baixo em relação a implementações em *software*. Neste sentido, os algoritmos estudados e as suas respectivas arquiteturas paralelas podem ser direcionados para problemas de otimização embarcadas, em que a capacidade computacional limitada das soluções microprocessadas convencionais é um obstáculo na resolução de problemas de otimização em uma quantidade de tempo aceitável.

2.3 OTIMIZAÇÃO POR ENXAME DE PARTÍCULAS (PSO)

O algoritmo PSO é uma técnica de otimização estocástica por inteligência de enxames, proposta por Kennedy e Eberhart [21], [47], inspirada no comportamento social de cardumes de peixes e bandos de aves na procura de alimento. O algoritmo PSO tomou como base as primeiras simulações de computação gráfica do comportamento social observado no voo sincronizado de bandos de aves [48], [49]. No PSO a população é chamada de enxame e cada indivíduo é chamado de *partícula*. A posição de uma partícula representa uma possível solução ao problema de otimização. O termo partícula foi inicialmente proposto para simulação gráfica de elementos nebulosos como nuvens, fogo ou fumaça [50]. No entanto, o termo partícula é utilizado como um compromisso no qual os indivíduos de um enxame podem ser interpretados como pontos (sem massa e sem volume).

No algoritmo PSO cada partícula tem uma velocidade aleatória associada, permitindo assim que as soluções potenciais se movimentem pelo espaço de busca do problema de otimização. Entretanto, cada partícula mensura a sua aptidão mediante a avaliação da função objetivo e conserva seu conhecimento do melhor valor de aptidão, por meio de uma memória individual e uma memória coletiva. A memória individual permite que a partícula lembre a posição (no espaço de busca) em que encontrou um melhor valor de aptidão, e a memória coletiva permite que as partículas lembrem a posição em que o enxame encontrou o melhor valor global de aptidão.

A nomenclatura para o PSO utilizada neste trabalho é a seguinte:

1. *Partícula ou agente*: único individuo do enxame.

2. *enxame*: coleção de agentes.

3. *Posição ($\boldsymbol{x}$)*: coordenadas de uma partícula no espaço N-dimensional que representa uma possível solução ao problema.

4. *Aptidão*: valor que representa quão boa é uma solução. Geralmente é a avaliação de uma função objetivo $f(\boldsymbol{x}) : \Re^n \rightarrow \Re$

5. *pbest ($\boldsymbol{y}_i$)*: posição da melhor aptidão para uma determinada partícula.

6. *gbest ($\boldsymbol{y}_s$)*: posição da melhor aptidão para o enxame inteiro.

7. v_{max}: velocidade máxima permitida em uma dada direção.

2.3.1 Algoritmo PSO básico

Considerando um espaço de busca N-dimensional e um enxame com S partículas, a posição da i-ésima (i^{th}) partícula do enxame na j-ésima (j^{th}) dimensão pode ser atualizada mediante as equações 2.1 e 2.2.

$$v_{ij}^{(t+1)} = v_{ij}^{(t)} + c_1 U_{1j}(y_{ij}^{(t)} - x_{ij}^{(t)}) + c_2 U_{2j}(y_{sj}^{(t)} - x_{ij}^{(t)}) \tag{2.1}$$

$$x_{ij}^{(t+1)} = x_{ij}^{(t)} + v_{ij}^{(t+1)} \tag{2.2}$$

em que U_{1j} e U_{2j} são números aleatórios uniformemente distribuídos entre 0 e 1, y_{ij} é a melhor posição individual da i^{th} partícula na j^{th} dimensão e y_{sj} é a melhor posição global entre todas as partículas na j^{th} dimensão. As velocidades v_{ij} estão limitadas na faixa $[-v_{max}, v_{max}]$ evitando assim que as partículas abandonem o espaço de busca.

Observe-se que existem dois parâmetros: o coeficiente cognitivo c_1 e o coeficiente social c_2. O comportamento do algoritmo PSO muda radicalmente com os coeficientes cognitivo e social. Um valor grande do coeficiente cognitivo (c_1) indica partículas com alta autoconfiança na sua experiência enquanto um valor grande do coeficiente social (c_2) proporciona às partículas maior confiança no enxame [30]. Para funções objetivo unimodais é aconselhável utilizar pequenos valores do coeficiente cognitivo e valores grandes para o coeficiente social, enquanto para funções multimodais é necessário encontrar um balanço entre os dois coeficientes visando melhorar o desempenho do algoritmo [21], [30].

O pseudocódigo do PSO básico é apresentado no Algoritmo 1. O PSO é um algoritmo iterativo, em que a cada iteração uma nova posição é calculada para cada partícula no enxame. No intuito de conferir a conveniência das posições geradas, as mesmas são avaliadas utilizando a função objetivo f. Para cada partícula, se o valor da função objetivo na posição atual $f(\boldsymbol{x}_i)$ é menor que o valor da função objetivo da melhor posição individual $fmin_k$, vide linha 12, então a melhor posição individual é substituída pela posição atual da partícula ($\boldsymbol{y}_i = \boldsymbol{x}$). Se o valor da função objetivo da posição atual $f(\boldsymbol{x}_i)$ é menor que o valor da função objetivo da melhor posição global $f(\boldsymbol{y}_s)$ então a melhor posição global é substituída pela posição atual da partícula ($\boldsymbol{y}_s = \boldsymbol{x}$).

Algoritmo 1: Pseudocódigo para o algoritmo PSO básico

Entrada: S, N, f, c_1, c_2, x_{max}, v_{max}, Max_{iter}, $threshold$

Saída: posição da melhor partícula: $\boldsymbol{x}$ e o seu melhor valor de aptidão: $f(\boldsymbol{x})$

```
1  início
2      Inicializa enxame:
3      para k = 1 : S faça
4          para j = 1 : N faça
5              v_kj = −v_max + 2U[0,1]v_max;
6              x_kj = −x_max + 2U[0,1]x_max;
7          fim para
8      fim para
9      repita
10         Avaliação e detecção:
11         para k = 1 : S faça
12             se f(x_k) ≤ fmin_k então
13                 y_ik = x_k;
14                 fmin_k = f(x_k);
15             fim se
16         fim para
17         calcule y_s usando os S valores de aptidão f(y_ik);
18         Atualização usando equações 2.1 e 2.2:
19         para k = 1 : S faça
20             para j = 1 : N faça
21                 v_kj = v_kj + c1U1[0,1](y_ikj − x_kj) + c2U2[0,1](y_sj − x_kj);
22                 x_kj = x_kj + v_kj;
23             fim para
24         fim para
25     até f(y_s) ≤ threshold;
26 fim
```

2.3.2 Modificações do algoritmo PSO

Diversos aportes e modificações têm sido propostos para o algoritmo PSO. Um dos primeiros aportes realizados foi uma versão binária do PSO [51], na qual a velocidade das partículas é definida em termos de probabilidades e a posição é calculada usando-se uma transformação sigmóide da velocidade ($S(v_{ij})$), tomando valores discretos 0 ou 1, permitindo trabalhar com variáveis discretas binárias. Esta abordagem possibilita realizar comparações com codificações binárias dos algoritmos genéticos assim como aplicações em problemas binários por natureza, por exemplo, a otimização das conexões de uma rede neural artificial [52]. Mais tarde foram introduzidos os primeiros aportes visando melhoramento da convergência do PSO. Neste sentido destacam-se o uso de fatores de inércia, proposto por Shi [53] e o uso de fatores de constrição [54].

O *fator de inércia* é aplicado durante o cálculo da velocidade, sendo utilizado como um fator de escala para a velocidade atual de cada partícula, como indicado na equação 2.3. Esta técnica é amplamente utilizada configurando o fator de inércia w para decrescer linearmente desde valores grandes até valores pequenos durante a execução do algoritmo. Neste sentido, Shi e Eberhart [55] realizaram experimentos utilizando diferentes funções objetivo demonstrando que o PSO tem um bom comportamento quando w decresce linearmente entre 0.9 até 0.4 durante o tempo de execução do algoritmo. O fator de inércia controla a capacidade de exploração das partículas. Valores grandes de w resultam em uma busca global enquanto valores pequenos de w permitem às partículas explorar localmente a vizinhança de uma possível solução. O algoritmo PSO com o uso do fator de inercia w e coeficientes c_1 e c_2 é conhecido academicamente como algoritmo PSO canônico.

$$v_{ij}^{(t+1)} = w v_{ij}^{(t)} + c_1 U_{1j}(y_{ij}^{(t)} - x_{ij}^{(t)}) + c_2 U_{2j}(y_{sj}^{(t)} - x_{ij}^{(t)}) \tag{2.3}$$

Por outro lado, o uso de fatores de constrição mostram-se úteis para assegurar a convergência no algoritmo [54]. O fator de constrição facilita na escolha dos parâmetros w, c_1 e c_2, mediante as seguintes relações:

$$v_{ij}^{(t+1)} = \chi(v_{ij}^{(t)} + c_1 U_{1j}(y_{ij}^{(t)} - x_{ij}^{(t)}) + c_2 U_{2j}(y_{sj}^{(t)} - x_{ij}^{(t)})) \tag{2.4}$$

$$\chi = \frac{2}{\left|2 - \phi - \sqrt{\phi^2 - 4\phi}\right|} \tag{2.5}$$

sendo, $\phi = c_1 + c_2$, $\phi > 4$. Desta forma é possível aplicar o PSO sem impor restrições na trajetória das partículas, especificamente no valor de velocidade máxima permitida v_{max}. O método de constrição mais encontrado na literatura é utilizar $\phi = 4.1$ com

constantes $c_1 = c_2 = 2.05$ ($\chi = 0.729$), que resulta no decrescimento gradual da amplitude da trajetória das partículas, assegurando assim a convergência do algoritmo [56].

Outras técnicas para melhoramento da qualidade das soluções têm sido propostas. Bergh e Engelbrech [57] apresentaram o algoritmo GCPSO (*Guaranteed Convergence* PSO), no qual um método de convergência rápida é utilizado para diminuir o tempo de execução do algoritmo, porém aumentando a susceptibilidade de convergência em mínimos locais no caso de funções multimodais. Uma abordagem cooperativa do PSO (CPSO) foi proposta por Bergh, na qual o vetor solução N-dimensional é separado em N enxames, cada um otimizando a solução em uma dimensão [58]. Esta proposta resultou em um melhoramento significativo do desempenho em termos de qualidade da solução e robustez, quando aplicado a problemas multimodais. Porém, pode apresentar problemas quando as variáveis de decisão estão correlacionadas entre si (*epistasis*). Posteriormente, uma técnica de partição da população chamada de NichePSO [59] foi utilizada para localizar e refinar os múltiplos ótimos em problemas de otimização multimodal. Esta técnica utiliza enxames guiados de forma cognitiva, os quais são gerados na região de uma possível solução do problema de otimização.

2.3.3 Topologias *GBEST* e *LBEST*

Quando o PSO é utilizado para resolver problemas de otimização multimodal é importante considerar a sociometria do enxame, diferenciando a forma como a informação deve ser compartilhada entre as partículas, no intuito de melhorar o desempenho do algoritmo [60]. Diversas topologias de enxame têm sido propostas, porém as topologias mais utilizadas são conhecidas como *GBEST* (estrela ou melhor global) e *LBEST* (anel ou melhor local), vide figura 2.7.

Na topologia *gbest* a vizinhança de cada partícula é o enxame inteiro, portanto, cada partícula influencia todas as outras, existindo uma conexão total entre todos os indivíduos. Entretanto, na topologia *lbest* a fonte social de influencia de cada partícula são os k vizinhos adjacentes. Segundo Kennedy, um algoritmo PSO com topologia *gbest* converge rapidamente devido a que todas as partículas são fortemente atraídas pela melhor partícula (partícula com melhor aptidão) no enxame, produzindo assim resultados subótimos quando a melhor partícula encontra-se presa num mínimo local. Por outro lado, um algoritmo PSO com topologia *lbest* converge lentamente, porém

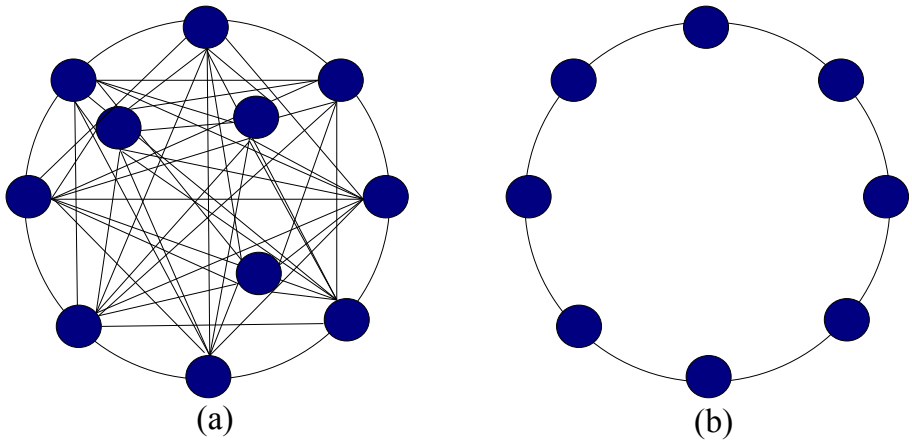


Figura 2.7: Topologias: (a) estrela ou *gbest* (b) anel ou *lbest* com k=2

com maiores chances de encontrar o ótimo global, devido a que cada partícula é influenciada apenas pelos vizinhos adjacentes e, portanto, grupos de vizinhos podem explorar distintas regiões ou ótimos locais no espaço de busca [61].

2.3.4 Técnicas híbridas e adaptativas

Os principais inconvenientes observados durante a execução do algoritmo PSO para problemas multimodais são: (a) *convergência prematura* do algoritmo resultando em partículas estagnadas em soluções subótimas e (b) tempos de execução elevados. Além dos anteriormente explicados, outros aportes têm sido propostos para solucionar estes problemas, destacando-se os *métodos híbridos* e as *técnicas adaptativas*.

Os métodos híbridos adotam ideias inspiradas em outras técnicas evolutivas ou bioinspiradas. Angeline [62] e Lovbjerg *et al.* [63] analisaram o efeito da utilização dos conceitos de reprodução e recombinação, adotados dos algoritmos genéticos, na velocidade de convergência do PSO. Maeda e Matsubita [64] utilizaram o método de perturbação simultânea no intuito de obter informação local do gradiente sem calculá-lo diretamente, constituindo assim um método de busca global e local simultaneamente. Uma abordagem híbrida do PSO baseada no sistema imunológico artificial foi proposta para resolver problemas discretos [65]. PSO com evolução diferencial [66], métodos híbridos Kalman PSO para a atualização das partículas [67] e métodos adaptativos Fuzzy PSO [68] têm sido estudados.

Como mencionado anteriormente, o comportamento do algoritmo PSO muda signi-

ficativamente com a escolha dos parâmetros. Abordagens adaptativas do algoritmo PSO têm sido propostas objetivando evitar o problema de convergência prematura, principalmente quanto utilizada a topologia *gbest*, na qual um número significativo de vizinhos produz uma convergência rápida, porém com uma forte atração a mínimos locais. Uma das técnicas mais eficientes para solucionar o problema de convergência prematura é o uso de diversidade artificial no enxame, evitando que as partículas se aglomerem em uma região do espaço de busca. Estas técnicas podem ser aplicadas a outros algoritmos evolutivos e serão explicadas na seção 2.7.

2.4 OTIMIZAÇÃO POR COLÔNIA DE ABELHAS ARTIFICIAIS (ABC)

O algoritmo ABC (*Artificial Bee Colony*) é uma das metaheurísticas inspiradas no comportamento social de colônias de abelhas durante a coleta de alimento. Este algoritmo, proposto por Karaboga [69], usa dois princípios básicos da inteligência coletiva: (1) agentes com um comportamento individual baseado em regras simples e (2) mecanismos de comunicação e compartilhamento da informação. Cada agente (abelha) possui capacidades limitadas e um conhecimento limitado do ambiente, geralmente registrando a posição geográfica, cor, forma e odor de uma florada. Por outro lado, o enxame desenvolve uma inteligência coletiva que permite a tomada de decisões mediante processos de intercâmbio de informação por meio de sons, danças, substâncias químicas e/ou estímulos eletromagnéticos [70], [27].

Modelos matemáticos desenvolvidos a partir dos estudos biológicos do comportamento de colônias de abelhas fazem uso de três componentes essenciais: (1) fontes de alimento (2) abelhas operárias e (3) abelhas não operárias [71], [72]. As fontes de alimento representam as possíveis fontes de néctar em uma florada. A escolha da fonte de alimento depende da proximidade à colmeia, qualidade e concentração de néctar.

As abelhas operárias estão associadas a uma fonte de alimento particular que estão explorando e carregam a respectiva informação: qualidade, distância e orientação. Após a colheita de néctar a abelha operária volta à colmeia, deposita o néctar nos favos e por meio de uma dança compartilha a informação sobre a fonte de alimento. A vivacidade da dança indica a distância da colmeia até a fonte de néctar e a orientação da dança indica a direção com relação ao sol.

Existem dois tipos de abelhas não operárias, as seguidoras e as escoteiras. As abelhas

seguidoras aguardam no interior da colmeia e observam a dança das operárias no intuito de seguir uma fonte de alimento. Uma vez esgotada uma fonte de alimento a abelha operária torna-se escoteira, cuja função é percorrer o ambiente aleatoriamente em busca de novas fontes de alimento.

Uma abelha seguidora pode realizar os seguintes comportamentos: (a) observar mais de uma operária e escolher, com determinada probabilidade, uma fonte de alimento em função da qualidade do néctar e da proximidade à colmeia; (b) se tornar uma abelha escoteira e explorar aleatoriamente ao redor da colmeia. Por outro lado, uma vez dentro da colmeia, as abelhas operárias podem realizar os seguintes tipos de ações: (a) dançar e recrutar seguidoras; (b) continuar a explorar a fonte de alimento; (c) abandonar a fonte de alimento e se tornar uma seguidora. Nestes modelos, o intercâmbio de informação entre abelhas é o processo mais importante na formação do conhecimento coletivo.

No algoritmo de otimização ABC, uma fonte de alimento representa uma possível solução ao problema de otimização, sendo a qualidade de uma fonte de alimento indicada por uma quantidade numérica, geralmente o valor da função custo. O número de operárias ou de seguidoras é igual ao número de fontes de alimento em torno da colmeia [73]. O intercâmbio de informação é simulado pelo cálculo de probabilidade em função da qualidade da fonte de alimento, como mostrado na equação 2.6.

$$p_i = \frac{f_i}{max(f)} \tag{2.6}$$

em que f_i é o valor de aptidão da i^{th} fonte de alimento, com i=1,...,S, sendo S o tamanho do enxame, p_i é a probabilidade da i^{th} solução e $max(f)$ o valor máximo da função custo entre toda a população de soluções. As abelhas seguidoras usam o vetor de probabilidades para escolher as fontes de alimento a ser exploradas e criam novas soluções por meio de uma busca local usando a equação 2.7.

$$x_{ij}^{(t+1)} = x_{ij}^{(t)} + \phi_{ij}(x_{ij}^{(t)} - x_{kj}^{(t)}) \tag{2.7}$$

sendo k=1,...,S, j=1,...,N, onde N é a dimensionalidade do problema, ϕ_{ij} é um numero aleatório com distribuição uniforme na faixa $[-1, 1]$. Nesta equação k e j são aleatoriamente gerados com $k \neq i$.

O pseudocódigo do ABC é apresentado no Algoritmo 2. Cada iteração é dividida em duas partes: fase das operárias e fase das seguidoras.

Na primeira fase, vide linhas de 5 a 8, para cada solução i determina-se aleatoriamente um vizinho k ($k \neq i$) e uma dimensão j. Posteriormente, usando a equação 2.7,

se determina a nova posição de cada operária. Para cada nova posição calcula-se o valor de aptidão. Se o valor da função objetivo é menor que o valor anterior (caso de minimização) então a nova posição é atualizada, caso contrário se incrementa o contador $trial_i$.

Na segunda fase, vide linhas de 10 a 18, são calculadas as probabilidades p_i segundo os valores de aptidão (equação 2.6). Passo seguinte, são enviadas as seguidoras de acordo com os valores de p_i, calculando uma nova posição para cada seguidora. Se o valor da função objetivo é menor que o valor anterior então a nova posição é atualizada, caso contrario incrementa-se o contador $trial_i$.

Finalmente, se para cada solução i o contador $trial_i$ é igual ao número máximo de tentativas permitidas sem melhoramento no valor de aptidão ($maxTrial$) então a fonte é abandonada e então é enviada uma escoteira para explorar aleatoriamente novas soluções, vide linha 19. Observe-se que para cada iteração a função custo é avaliada $2S$ vezes, primeiro na fase das operarias e depois na fase das seguidoras.

2.5 OTIMIZAÇÃO POR COLÔNIA DE VAGA-LUMES (FA)

O algoritmo FA (*firefly algorithm*) é uma metaheurística baseada em populações recentemente estudada e inspirada no comportamento social de insetos lampirídeos como vaga-lumes ou pirilampos. Os lampirídeos se caracterizam por produzir lampejos rítmicos, em que cada espécie possui um padrão único de lampejo. Este padrão é utilizado para acasalamento ou para atrair uma possível presa, sendo o ritmo, a velocidade e duração do lampejo um conjunto de características usadas para comunicação entre indivíduos da mesma espécie. Em algumas espécies de lampirídeos, as fêmeas podem observar os sinais de bioluminescência, e incluso imitar os padrões de lampejo de outras espécies, permitindo atrair os machos de outras espécies para servir de alimento.

No algoritmo FA, introduzido por Yang [74], [22], os padrões de lampejo são utilizados para simular os mecanismos de comunicação entre indivíduos da mesma espécie. Neste algoritmo os lampejos de luz e sua intensidade estão associados com o valor da função objetivo a ser otimizada. Por simplicidade três regras básicas foram consideradas na elaboração do algoritmo: (a) os vaga-lumes atraem-se entre si sem importar o sexo; (b) a atração é proporcional à intensidade e decresce à medida que a distância entre

Algoritmo 2: Pseudocódigo para o algoritmo ABC

Entrada: S, N, f, $[x_{min}, x_{max}]$, $maxTrial$, $maxIter$

Saída: posição da melhor fonte de alimento: $\boldsymbol{x}_g$ e o seu melhor valor de aptidão: $f(\boldsymbol{x}_g)$

```
início
    Gerar as posições aleatórias para as S fontes de alimento e determinar aptidões f(x)
    repita
        // Enviar operarias para explorar as fontes de alimento:
        Para cada solução i determinar um vizinho k e dimensão j
        Criar uma nova solução usando a equação 2.7
        Calcular os valores de aptidão f(x)
        Atualizar posições se f(x_i) melhora o valor anterior
        // Enviar seguidoras para explorar as fontes de alimento segundo p_i:
        Calcular o vetor de probabilidades p_i usando equação 2.6
        para i = 1 : S faça
            se rand() > p_i então
                Determinar um vizinho k e dimensão j
                Criar uma nova solução x_i usando a equação 2.7
                Calcular o valor de aptidão f(x_i)
                Atualizar a posição se f(x_i) melhora o valor anterior
            fim se
        fim para
        Determine as soluções abandonadas e envie as abelhas escoteiras para buscar novas
        fontes de alimento
        Atualize a melhor solução x_g segundo as aptidões
        iter = iter + 1
    até iter ≤ maxIter;
fim
```

dois indivíduos aumenta, assim vaga-lumes de menor brilho são atraídos pelos vaga-lumes de maior brilho; (c) a intensidade do lampejo é determinado pelo valor da função objetivo a ser otimizada, isto é, $I(\boldsymbol{x}) \propto f(\boldsymbol{x})$.

A atração β é proporcional à intensidade do lampejo I. Entretanto, dado que a luz é absorvida pelo médio, a intensidade decresce com a distância desde a fonte. Considerando um médio com coeficiente de absorção constante γ, pode-se calcular a atração entre dois vaga-lumes adjacentes usando qualquer função monótona decrescente, tal como mostrado no seguinte modelo generalizado.

$$\beta = \beta_0 e^{-\gamma \cdot r_{ik}^m}, \ (m \geq 1). \tag{2.8}$$

sendo β_0 a atração a $r_{ik} = 0$ e r_{ik} a distância entre os vaga-lumes i e k. O parâmetro γ caracteriza a variação da atração, sendo importante na velocidade de convergência do

algoritmo. Em teoria, $\gamma \in [0, \infty)$, porém na prática γ é determinado pelo comprimento característico Γ (escala ou faixa de valores possíveis) do sistema a otimizar. Comumente utiliza-se $m = 2$, permitindo relacionar o comprimento característico e o coeficiente de absorção da seguinte forma $\Gamma = 1/\sqrt{\gamma}$, definindo assim uma atração que varia significativamente na faixa $[\beta_0, \beta_0 e^{-1}]$.

O movimento do vaga-lume i, atraído por um vaga-lume vizinho k com lampejo de maior intensidade, está determinado pela equação 2.9.

$$x_i^{(t+1)} = x_i^{(t)} + \beta(x_k^{(t)} - x_i^{(t)}) + \alpha(U_i - 1/2) \tag{2.9}$$

sendo U_i um número aleatório com distribuição uniforme na faixa $[0, 1]$ e α um coeficiente de ajuste de aleatoriedade. Observe-se que a equação de movimento esta definida por três componentes: (a) componente individual; (b) componente de atração entre indivíduos e (c) componente de aleatoriedade. Na prática utiliza-se $\beta_0 = 1$ e $\alpha \in [0, 1]$ podendo o valor de α decrescer no tempo no intuito de refinar a solução durante as últimas iterações. No algoritmo original a equação de atualização do coeficiente α é não linear, como mostrado na equação 2.10, onde $\delta \approx 1, \delta \leq 1$.

$$\alpha^{(t+1)} = \alpha^{(t)} \cdot \delta \tag{2.10}$$

O pseudocódigo do FA é apresentado no Algoritmo 3. Observe-se que no algoritmo FA original não existe a etapa de comparação do valor de aptidão individual. Portanto, as partículas não retornam à sua posição original quando a aptidão não é melhorada, sempre se movimentando em direção do seu melhor vizinho mais próximo. Embora este fato facilite uma possível implementação *hardware* do algoritmo FA, pode esperar-se uma perda de desempenho do algoritmo, em termos da qualidade da solução se comparado com outros algoritmos de inteligência de enxames, dado que no FA é permitido às partículas explorar soluções não ótimas.

2.6 OTIMIZAÇÃO POR EMBARALHAMENTO DE SALTO DE SAPOS (SFLA)

O algoritmo SFLA (*shuffled frog leaping algorithm*), introduzido por Eusuff [75], é uma técnica de otimização estocástica por inteligência de enxames inspirada no comportamento social de grupos de sapos saltando em um pântano na procura de comida.

Algoritmo 3: Pseudocódigo para o algoritmo FA

Entrada: S, N, f, $[x_{min}, x_{max}]$, $\alpha^{(1)}$, δ, $maxIter$

Saída: posição do melhor vaga-lume: $\boldsymbol{x}_g$ e o seu melhor valor de aptidão: $f(\boldsymbol{x}_g)$

```
início
    Gerar as posições aleatórias x_i para os S vaga-lumes
    Calcular as intensidades I_i avaliando a função custo f(x_i)
    Determinar o coeficiente de absorção γ
    repita
        para i = 1 : S faça
            para k = 1 : S faça
                se I_k > I_i então
                    Calcular a distância r_ik e a atração β usando equação 2.8
                    Criar uma nova solução x_i usando a equação 2.9
                    Avaliar a nova solução f(x_i) e atualizar a intensidade I_i
                fim se
            fim para
        fim para
        Ordena os vaga-lumes segundo o valor de aptidão e calcula o melhor global
        Atualize a melhor solução x_g
        Calcula novo valor de α usando equação 2.10
        iter = iter + 1
    até iter ≤ maxIter;
fim
```

Este algoritmo se fundamenta nas capacidades de busca local do algoritmo PSO e no intercâmbio global de informação adotado dos algoritmos meméticos [76]. Desta forma, foi desenvolvida uma nova heurística para problemas de otimização com poucos parâmetros de ajuste [75], [77].

Dado que em parte o SFLA tomou como base os princípios dos algoritmos meméticos, é importante esclarecer estes princípios e definir a terminologia a ser utilizada. Os algoritmos meméticos, inicialmente propostos por Moscato [76], fundamentam seu nome do termo inglês *meme*, utilizado por Dawkins como um análogo do *gene* no contexto da evolução cultural. Exemplos de *memes* são melodias, ideias, gírias, moda, formas de construção, etc. Assim como os *genes* se transmitem através do código genético, os *memes* se transmitem de cérebro para cérebro através de um processo que poderia chamar-se de imitação [78].

Algoritmos de otimização meméticos utilizam agentes otimizadores. Cada agente é uma das soluções dentre uma população de diversas soluções para um problema dado.

Os agentes podem se relacionar entre si por meio de um processo de cooperação e competição análogo ao processo de intercâmbio genético entre indivíduos da mesma espécie. Os agentes também fazem uso explícito do conhecimento do problema adquirido previamente, mediante um processo de busca local [79]. Desta forma, os algoritmos meméticos se baseiam na exploração sistemática do conhecimento do problema, assim como de ideias tomadas de metaheurísticas baseadas em populações e metaheurísticas de busca local.

No algoritmo SFLA o pântano representa o espaço de busca N-dimensional e a posição dos sapos representa uma possível solução ao problema de otimização. O algoritmo opera particionando o conjunto de sapos no pântano em subconjuntos paralelos, conhecidos como *memeplexes*, onde os sapos de cada *memeplex* realizam um processo de busca local. Neste processo apenas o pior sapo (sapo com pior valor de aptidão) é melhorado, utilizando para isto a posição do melhor sapo no *memeplex* e a posição do melhor sapo do pântano [77], [80]. Após um número definido de iterações locais, os sapos são embaralhados e novos *memeplexes* são criados, começando um novo processo de busca local.

A nomenclatura usada para o algoritmo SFLA é a seguinte:

1. *Pântano*: representa o espaço de busca do problema de otimização.
2. *Sapo ou agente*: único individuo do pântano.
3. *Memeplex*: ilha no pântano (subconjunto do espaço de busca) que contém uma *cultura de sapos*, os quais representam as posições de possíveis soluções.
4. *Meme*: unidade da cultura de sapos formada por um bloco de informação chamado de *ideia* que é transmitida para melhorar as posições dos sapos no *memeplex*, por meio de um processo de *infecção de ideias*, também chamado de evolução memética.
5. *Memotipo*: conteúdo informático de um *meme*, por exemplo, a posição de um sapo.
6. *Salto* ($\boldsymbol{l}$): memotipo utilizado para melhorar a posição de um sapo.
7. *Posição* ($\boldsymbol{x}$): coordenadas de um sapo no espaço N-dimensional que representa uma possível solução ao problema.

8. *Aptidão*: valor que representa quão boa é uma solução. Geralmente é a avaliação de uma função objetivo $f(\boldsymbol{x}) : \Re^n \rightarrow \Re$

9. *lmax* (l_{max}): salto máximo permitido em uma dada direção.

10. *xworst* ($\boldsymbol{x}_w$): posição de um sapo no *memeplex* com o pior valor de aptidão.

11. *xbest* ($\boldsymbol{x}_b$): posição de um sapo no *memeplex* com o melhor valor de aptidão.

12. *xglobal* ($\boldsymbol{x}_g$): posição do sapo com a melhor aptidão no pântano.

O pseudocódigo do algoritmo é apresentado no Algoritmo 4 [80]. Primeiramente a população de S sapos é aleatoriamente inicializada e estes organizados em ordem decrescente do valor de aptidão. A população organizada de sapos é dividida em M subconjuntos (*memeplexes*) cada um contendo F sapos, de forma que $S = MF$. A seguinte estratégia de divisão é aplicada: o primeiro sapo vai para o primeiro *memeplex*, o segundo sapo para o segundo *memeplex*, o M^{th} sapo para o M^{th} *memeplex*, o M^{th+1} sapo vai para o primeiro *memeplex*, etc. No passo seguinte o sapo com a melhor posição global ($\boldsymbol{x}_g$) é identificado e essa informação é compartilhada entre todos os *memeplexes*. Adicionalmente cada *memeplex* identifica as posições do melhor sapo ($\boldsymbol{x}_b$) e do pior sapo ($\boldsymbol{x}_w$) de acordo com o seu valor de aptidão.

Posteriormente, um processo de busca local, vide linhas 7 a 16, em cada *memeplex* é realizado no intuito de melhorar a posição do pior sapo ($\boldsymbol{x}_w$), a qual é atualizada usando a posição do melhor sapo ($\boldsymbol{x}_b$), como indicado nas equações 2.11 e 2.12.

$$\boldsymbol{l}_w^{(t+1)} = U_w \cdot (\boldsymbol{x}_b^{(t)} - \boldsymbol{x}_w^{(t)}) \tag{2.11}$$

$$\boldsymbol{x}_w^{(t+1)} = \boldsymbol{l}_w^{(t+1)} + \boldsymbol{x}_w^{(t)} \tag{2.12}$$

em que U_w é um número uniformemente distribuído entre 0 e 1, $\boldsymbol{l}_w$ é o salto calculado para o pior sapo. Os saltos estão limitados na faixa $[-l_{max}, l_{max}]$ evitando que os sapos abandonem o espaço de busca. Se a nova posição do pior sapo melhora a sua aptidão atual, então a posição é atualizada, caso contrario as equações 2.11 e 2.12 são novamente executadas, porém substituindo a melhor posição no *memeplex* ($\boldsymbol{x}_b$) pela a melhor posição global ($\boldsymbol{x}_g$). Se a nova posição do pior sapo não melhora a sua aptidão então o sapo é eliminado e um novo sapo é criado em uma posição aleatória do espaço de busca. O processo anteriormente descrito é realizado até completar um número previamente definido de iterações locais.

Após o processo de busca local os sapos são embaralhados e organizados em ordem decrescente de acordo com seu valor de aptidão, divididos em M grupos e então um novo processo de busca local é inicializado.

Algoritmo 4: Pseudocódigo para o algoritmo SFLA

Entrada: S, M, f, F, N, l_{max}, x_{max}, Max_{LS}, $threshold$
Saída: posição do melhor sapo: $\boldsymbol{x}$ e o seu melhor valor de aptidão: $f(\boldsymbol{x})$

```
início
    Gerar as posições aleatórias para os S sapos;
    Dividir os S sapos em M memeplexes
    repita
        Avaliação das S aptidões f(x)
        Identificar a melhor posição global x_g e respectiva aptidão fmin
        para i = 1 : Max_LS faça
            Identificar a melhor e a pior posição (x_b, x_w) de acordo com a aptidão
            Atualizar x_w^(t+1) usando as equações 2.11 e 2.12
            se f(x_w^(t+1)) ≥ f(x_w^(t)) então
                Atualizar x_w^(t+1) com 2.11 e 2.12 usando x_g em lugar de x_b
                se f(x_w^(t+1)) ≥ f(x_w^(t)) então
                    x_w^(t+1) = −x_max + 2U[0, 1]x_max;
                fim se
            fim se
        fim para
        Embaralhar os M memeplexes evoluídos
    até fmin ≤ threshold;
fim
```

O algoritmo SFLA tem recebido atenção por parte da comunidade científica devido às suas capacidades de busca global. São encontrados alguns aportes na literatura abarcando tanto aplicações como modificações do algoritmo. Por exemplo, Eusuff e Lansey [75] propuseram e aplicaram o algoritmo para a otimização do tamanho discreto de tubos no projeto de em uma rede de distribuição de água. Aplicações em problemas de otimização combinatória [77], [81], gerenciamento de estruturas em problemas de engenharia construtiva [82], sintonização de controladores PID [83], projeto de controladores nebulosos [84], classificação de documentos [85], *clusters* de dados [86], segmentação de imagens [87], entre outras, são encontradas na literatura.

Por outro lado, as principais modificações visando melhorias no desempenho do algoritmo são as seguintes: (a) Huynh *et al.* acrescentaram incertezas em cada dimensão na equação de cálculo do salto visando evitar o problema de convergência prematura [83], [84]; (b) Um algoritmo SFLA discreto (análogo à versão binária do PSO) foi de-

senvolvido utilizando um espaço de busca binário, aplicando uma função sigmoide após o cálculo do salto, visando simplificações e ganhos de desempenho [88]; (c) processos de seleção por clonagem baseados no sistema imunológico foram aplicados no algoritmo SFLA para evitar problemas de estagnação, especificamente quando a distância entre o melhor sapo e o pior sapo é pequena [87].

2.7 MÉTODOS DE ADIÇÃO DE DIVERSIDADE ARTIFICIAL

Segundo os estudos realizados por Mendes [16] existe uma forte relação entre a topologia social do enxame e a robustez à *convergência prematura*. A diversidade do enxame é um fator de importância na execução do algoritmo PSO e, em geral, nos algoritmos baseados em populações. Os métodos de adição de diversidade são parte das técnicas adaptativas utilizadas para evitar o problema de convergência prematura, especificamente quando são usadas topologias em que o enxame segue o indivíduo com melhor desempenho, por exemplo, na topologia GBEST. Neste caso, a convergência é rápida, porém existe uma forte tendência a encontrar soluções subótimas quando o melhor indivíduo encontra-se preso em um mínimo local.

Algumas propostas têm sido realizadas visando manter a diversidade do enxame. Estas propostas têm sido aplicadas principalmente no algoritmo PSO, entre as quais se encontram os métodos de incremento de diversidade mediante técnicas de evasão de colisões entre as partículas [89], entropia negativa [90] e partículas com extensão espacial (tratadas como esferas) [91] e [92]. Lovbjerg e Krink [93] utilizaram um algoritmo PSO auto-organizado no intuito de ajudar o enxame a manter a diversidade fazendo com que seja menos vulnerável a ótimos locais. Entretanto, a maioria dos métodos de adição de diversidade artificial têm provocado um aumento na complexidade computacional dos algoritmos, dificultando assim a sua implementação e aumentando o consumo dos recursos computacionais.

Neste trabalho foram estudados três mecanismos de adição de diversidade que se caracterizam pelo uso de operadores simples, preservando a filosofia de fácil implementação dos algoritmos, possibilitando assim seu mapeamento em *hardware*. As técnicas implementadas são: (a) o método atrativo repulsivo; (b) o método de congregação passiva seletiva; (c) o método de aprendizado em oposição (*opposition based learning* - OBL). Antes de realizar uma explicação das técnicas de adição de diversidade estudadas é necessário destacar que as mesmas foram aplicadas inicialmente no algoritmo PSO e

a sua respectiva implementação em FPGAs foi analisada em termos de consumo de recursos e convergência dos resultados. Estes resultados permitiram demonstrar que a técnica OBL requer de menos recursos de *hardware* e apresenta resultados de convergência satisfatórios (vide apêndice B), sendo, portanto, escolhida para ser aplicada nos outros algoritmos de otimização estudados neste trabalho.

2.7.1 O método atrativo repulsivo

O método atrativo repulsivo foi proposto por Riget e Vesterstrom [94]. Neste método, realiza-se uma medida da diversidade do enxame para guiar o seu comportamento. A medida de diversidade é feita utilizando a distância Euclidiana entre as partículas, vide equações 2.13 e 2.14.

$$diversidade(S) = \frac{1}{SL}\sum_{i=1}^{S}\sqrt{\sum_{j=1}^{N}(x_{ij} - \bar{x_j})^2} \tag{2.13}$$

$$\bar{x_j} = \frac{1}{S}\sum_{i=1}^{S} x_{ij} \tag{2.14}$$

em que S é o tamanho do enxame, L é comprimento da maior diagonal no espaço de busca e N é a dimensionalidade do problema de otimização.

Esta técnica, inicialmente aplicada no algoritmo PSO (arPSO), calcula uma medida da diversidade a cada iteração para comutar entre a fase de atração e de repulsão. Para isso a equação modificada do cálculo da velocidade das partículas (vide equação 2.15) é utilizada. Quando a diversidade é inferior a um valor mínimo ($dlow$), o enxame muda para a fase repulsiva ($dir = -1$) e então o enxame aumenta a sua diversidade. Quando a diversidade ultrapassa o limite superior ($dhigh$), o enxame entra na fase atrativa ($dir = 1$) e então as partículas começam a convergir novamente (algoritmo PSO original).

$$v_{ij}^{(t+1)} = wv_{ij}^{(t)} + dir[c_1U_{1j}(y_{ij}^{(t)} - x_{ij}^{(t)}) + c_2U_{2j}(y_{sj}^{(t)} - x_{ij}^{(t)})] \tag{2.15}$$

2.7.2 O método de congregação passiva seletiva

O método de congregação passiva está baseado nos estudos sobre organização espaço-temporal de grupos de animais realizados por Parrish e Hammer [95]. Tais estudos

demonstram que existem forças naturais que operam usando diferentes mecanismos de intercâmbio de informação, permitindo aos enxames se movimentar sem perder a densidade e forma. Parrish e Hammer classificaram em dois grupos as forças biológicas que atuam na organização de grupos de animais, *agregação* e *congregação*. A agregação de enxames pode acontecer por meio de processos físicos (agregação ativa) ou mediante a busca de recursos, comida e/ou espaço (agregação passiva). Por outro lado, a congregação é uma atração por processos sociais. Na congregação ativa existe uma forte correlação entre os indivíduos do enxame, permitindo a divisão de labores usando indivíduos especializados. A congregação passiva acontece quando um indivíduo atrai os outros, incluso quando não existe nenhuma relação genética entre eles.

O primeiro modelo de congregação passiva foi proposto por He [96] e aplicado ao algoritmo PSO. Tal modelo preserva a fácil implementação do algoritmo mediante uma simples modificação na equação de cálculo da velocidade, como mostrado na equação 2.16

$$v_{ij}^{(t+1)} = v_{ij}^{(t)} + c_1 U_{1j}(y_{ij}^{(t)} - x_{ij}^{(t)}) + c_2 U_{2j}(y_{sj}^{(t)} - x_{ij}^{(t)}) + c_3 U_{3j}(sel_{ij}^{(t)} - x_{ij}^{(t)}) \qquad (2.16)$$

sendo U_{3j} um número aleatório com distribuição uniforme na faixa [0,1], c_3 o coeficiente de congregação passiva e sel_{ij} uma partícula escolhida aleatoriamente.

A principal desvantagem de usar partículas aleatórias para o cálculo do componente de congregação passiva é que partículas com um baixo desempenho poderiam atrair as outras, impedindo a melhoria contínua na posição das mesmas. Neste contexto, Voit [97] propôs o uso da segunda melhor partícula para o cálculo da velocidade (vide equação 2.17. Assim a congregação passiva pela seleção da segunda melhor partícula permite manter a filosofia de fácil implementação e melhoria continua do algoritmo PSO.

$$v_{ij}^{(t+1)} = v_{ij}^{(t)} + c_1 U_{1j}(y_{ij}^{(t)} - x_{ij}^{(t)}) + c_2 U_{2j}(y_{sj}^{(t)} - x_{ij}^{(t)}) + c_3 U_{3j}(sb_{ij}^{(t)} - x_{ij}^{(t)}) \qquad (2.17)$$

sendo sb_{ij} a partícula com o segundo melhor valor de aptidão.

2.7.3 O método de aprendizado em oposição

O método de aprendizado em oposição (OBL - *Opposition-based Learning*) foi proposto por Tizhoosh [98] como uma técnica de fácil implementação que permite ao algoritmos baseados em populações fazer a busca do ponto ótimo na direção oposta da busca

atual. A ideia básica desta técnica é que quando uma busca local está sendo realizada em uma direção é importante considerar também a direção oposta [99]. Desta forma o método OBL permite não só melhorar a qualidade da solução senão também preservar a diversidade do enxame.

A abordagem OBL está baseada no conceito do número oposto, definido pela equação 2.18

$$\breve{x} = a + b - x \tag{2.18}$$

sendo x um número real definido na faixa $[a, b]$ e $\breve{x}$ o número oposto de x. Esta definição também é válida para pontos N-dimensionais x_i definidos na faixa $[a_i, b_i]$, $i = 1, 2, ..., N$.

A técnica OBL foi inicialmente aplicada em algoritmos genéticos no qual o conceito de anticromossoma permite que o processo de busca seja acelerado. O OBL também foi aplicado no treinamento de redes neurais artificiais em que o conceito de peso oposto e rede oposta permitiram melhorar os resultados do treinamento [98].

Rahnamayan [100] realizou um estudo comparativo entre a aplicação da técnica OBL e o uso de números aleatórios demonstrando formalmente que, no caso de uma função desconhecida em um espaço N-dimensional, o uso do número oposto tem maiores chances de encontrar o ponto ótimo do que simplesmente usar números aleatórios. Adicionalmente, uma comprovação empírica destas demonstrações matemáticas foram conduzidas montrando que os processos de busca usando os números aleatórios x e seus opostos $\breve{x}$ melhoram o resultado final. Recentemente, a técnica OBL tem sido aplicada nos algoritmos de evolução diferencial (DE) [99], otimização por colônia de formigas [101] e PSO [102].

2.8 IMPLEMENTAÇÕES PARALELAS DOS ALGORITMOS CITADOS

Embora diversas técnicas híbridas e adaptativas tenham sido aplicadas melhorando o desempenho dos algoritmos de otimização, muitos problemas de engenharia continuam sendo impraticáveis devido ao alto custo computacional requerido em termos de recursos de tempo. Os algoritmos de otimização por inteligência de enxames e, em geral, os algoritmos baseados em populações, possuem uma natureza essencialmente paralela, pois são inspirados no comportamento social de espécies naturais. Em muitos

casos, como bactérias, insetos ou alguns grupos de animais, os indivíduos exploram o seu habitat simultaneamente, mediante a realização de labores cooperativas, essencialmente paralelas, intercambiando informação por meio de processos de *agregação* ou *congregação* visando encontrar as condições ótimas para alcançar um determinado objetivo.

2.8.1 O uso de *clusters* de PCs nos algoritmos de inteligência de enxames

Alguns aportes têm sido realizados visando explorar abordagens paralelas dos algoritmos de otimização baseados em populações. Neste sentido, as primeiras propostas foram implementadas usando *clusters* PCs conectados em uma rede de comunicação. Shutte *et al.* [103] realizaram uma implementação paralela síncrona do PSO (PSPSO) a qual foi aplicada em problemas de identificação de sistemas biomecânicos, demonstrando a sua viabilidade para resolver problemas de otimização de grande escala. Parsopoulos *et al.* [104] implementaram um algoritmo PSO paralelo para problemas de otimização multiobjetivo utilizando vetores de avaliação (VEPSO). Em tal trabalho, enxames paralelos são avaliados utilizando somente uma função objetivo para cada enxame e a informação obtida pelo enxame sobre a função é comunicada para os outros enxames por meio do intercambio da melhor partícula. Posteriormente, uma versão paralela assíncrona do PSO (PAPSO) foi proposta para problemas de otimização estrutural de asa de avião, destacando a eficiência paralela computacional se comparado com a versão síncrona (PSPSO) [105]. Koh *et al.* [106] demonstraram que o PAPSO é aproximadamente 3.5 vezes mais rápido do que o PSPSO para resolver problemas de otimização biomecânica executados em um *cluster* de 20 processadores. Abordagens paralelas do PSO utilizando diferentes estratégias de comunicação entre processadores foram aplicadas para problemas complexos, tais como projeto neutrônico de reatores nucleares e otimização de recarga de combustível nuclear [107].

Embora a comunidade científica reconheça o paralelismo intrínseco dos algoritmos de otimização por inteligência de enxames, a maioria dos trabalhos continuam sendo implementados de forma sequencial utilizando processadores de propósito geral. Entretanto, a principal desvantagem das abordagens paralelas anteriormente mencionadas é o alto custo de implementação, considerando que os algoritmos de otimização por inteligência de enxames simplificam a solução dos problemas e, portanto, requerem apenas operações aritméticas básicas. Desta forma, o maior custo computacional em termos de recursos de tempo é devido à avaliação da função objetivo. Adicionalmente,

as soluções por *clusters* de PCs, impossibilitam a sua implementação em plataformas embarcadas ou portáveis.

2.8.2 O uso de GPUs nos algoritmos de inteligência de enxames

Recentemente, unidades de processamento gráfico GPUs (*Graphic Processor Units*) têm sido utilizados para acelerar a execução de algoritmos complexos, que podem ser desenvolvidos aproveitando a alta capacidade computacional baseada em arquiteturas massivamente paralelas com suporte de cálculo aritmético de ponto flutuante [108]. Embora os dispositivos GPUs foram inicialmente desenvolvidos para processamento gráfico, eles também podem ser projetados para aplicações de propósito geral (GPGPUs *General Purpose Computation on GPUs*) mediante a utilização do ambiente de programação CUDA (*Compute Unified Device Architecture*) [109], [110].

No campo dos algoritmos de otimização, as GPGPUs têm oferecido soluções viáveis para resolver problemas de otimização de grande escala com um tempo de execução consideravelmente baixo se comparado com uma implementação em processadores convencionais. Zhou e Tan [111] e Veronese *et al.* [110] apresentaram uma implementação do algoritmo PSO em GPGPUs, demonstrando uma alta capacidade de aceleramento do algoritmo para resolver problemas de otimização de alta dimensionalidade em tempo real. Mussi *et al.* [112] apresenta uma implementação em GPU do algoritmo PSO aplicado ao problema de reconhecimento de sinais de transito em rodovias. Contudo, a solução baseada em GPUs ainda impossibilita a aplicação dos algoritmos em sistemas embarcados ou em sistemas computacionais de pequeno porte.

2.8.3 O uso de FPGAs nos algoritmos de inteligência de enxames

Abordagens paralelas dos algoritmos de otimização por inteligência de enxames utilizando dispositivos lógicos programáveis têm sido propostas nos últimos anos. Dispositivos FPGAs podem ser úteis para a implementação de abordagens paralelas dos algoritmos por inteligência de enxames visando diminuir o tempo de execução. Tais implementações podem oferecer ganhos no desempenho não só pela implementação de indivíduos e funções custo em paralelo, senão também por meio de cálculos simultâneos durante a atualização das posições das partículas e a avaliação da função objetivo.

Existem na literatura alguns estudos que abordam as capacidades de paralelização do algoritmo PSO usando FPGAs.

Reynolds *et al.* [113] implementaram o algoritmo PSO em FPGAs para inversão de redes neurais artificiais demonstrando que o tempo de execução do PSO em FPGAs é aproximadamente 6 vezes mais rápido do que em computadores convencionais. Uma arquitetura totalmente paralela do PSO, baseada em uma estrutura matricial para sincronizar as partículas com um módulo de avaliação da função custo, foi desenvolvida para otimização dinâmica de arranjos de antenas [114]. Entretanto, esses dois primeiros trabalhos não reportam detalhes de custo em área e desempenho das implementações em *hardware*.

Peña *et al.* [115] apresentaram uma arquitetura em *hardware* do PSO com recombinação discreta na qual um módulo externo de geração de número aleatórios proporciona o caráter estocástico na atualização da velocidade das partículas. Nesse trabalho a arquitetura foi avaliada usando 16 partículas em paralelo para duas funções de teste tipo *benchmark* de 32 dimensões (*Esfera* e *Rastrigin*), sendo o algoritmo executado 8600 vezes mais rápido do que uma implementação no Matlab. Maeda e Matsushita [116] apresentaram uma implementação do algoritmo de otimização híbrido PSO com perturbação simultânea em FPGAs. Palangpour *et al.* [117] apresentaram uma implementação em *hardware* do algoritmo PSO, a qual foi validada para as funções de teste *Esfera* e *Rosenbrock*, reportando resultados de síntese e de tempo de execução para problemas de 1, 5 e 10 dimensões. Os resultados reportados nos trabalhos prévios demonstraram que a implementação em *hardware* do PSO reduz em algumas ordens de magnitude o tempo de execução de uma implementação no Matlab.

Farahani *et al.* [118] implementou em FPGAs uma arquitetura assíncrona do algoritmo PSO discreto utilizando técnicas de partição *hardware software*. As partículas são atualizadas em uma abordagem paralela enquanto a função objetivo é implementada utilizando um microprocessador embarcado NIOS II da Altera. A referida arquitetura flexibiliza a aplicabilidade das FPGAs para o algoritmo PSO devido a que a função custo, que depende diretamente do problema a ser resolvido, é facilmente implementada em *software*, enquanto as outras operações do PSO se mantêm inalteradas e são executadas em *hardware*. Contudo, uma perda de desempenho deve ser considerada devido ao caráter sequencial durante a avaliação da função custo, assim como na transferência de dados entre o processador embarcado e as unidades paralelas de atualização das partículas. No intuito de resolver esse problema, uma arquitetura usando quatro

microprocessadores NIOS II em paralelo foi posteriormente desenvolvida e aplicada ao problema de inversão de redes neurais artificiais [119].

Entre as aplicações das implementações em FPGAs do algoritmo PSO podem-se mencionar as seguintes: (a) uma arquitetura em FPGA para uma rede neural artificial tipo *wavelet* com aprendizado por PSO foi proposta por Lin e Tsai [120]. (b) Filtros IIR adaptativos utilizando o PSO e sua respectiva implementação em FPGAs foram propostos por Gao *et al.* [121]. (c) Mehmood *et al.* [122] descreveram uma implementação paralela do PSO, chamada de HPSO, na qual uma abordagem modular é apresentada separando os blocos de enxame dos blocos de avaliação da função objetivo. A referida arquitetura foi aplicada em problema de detecção de objetos mostrando que o algoritmo pode ser facilmente configurado para diferentes tarefas de detecção, bem pela mudança de parâmetros das funções custo ou por meio da definição de novas funções custo. (d) Uma versão do PSO adaptativo foi implementada em FPGAs para otimização dos pesos de parâmetros patológicos e fisiológicos em um sistema de diagnose médico para pacientes com disfunções renais, permitindo assim realizar diagnoses mais precisas de doenças renais em zonas rurais em que há ausência de pessoal médico [123].

Como comentários a serem feitos sobre os trabalhos acima citados pode-se dizer que todos eles utilizam uma aritmética de ponto fixo. Em geral, uma implementação em FPGA utilizando ponto fixo permite alcançar frequências de operação maiores e um consumo de recursos de *hardware* menor se comparado com uma implementação em ponto flutuante. Todavia, existem muitas aplicações em que os processos de otimização requerem realizar cálculos com alta precisão utilizando números muito grandes e muito pequenos. Este fato sugere a utilização de uma faixa dinâmica para a representação numérica durante a execução dos algoritmos de otimização, especificamente durante a avaliação das funções custo. Porém, uma das abordagens mais utilizadas nos dispositivos FPGAs é a representação aritmética de inteiros ou em ponto fixo, enquanto que implementações em *software*, comumente baseadas em PCs, operam com uma aritmética de ponto flutuante com as suas respectivas vantagens de alta precisão e faixa dinâmica.

Existem poucos trabalhos reportando implementações em FPGAs de algoritmos de otimização por inteligência de enxames utilizando aritmética de ponto flutuante. Tewolde *et al.* [124], [125] [126], [127] apresentaram uma implementação em ponto flutuante de precisão simples para uma arquitetura serial do PSO, na qual são utilizadas memórias para armazenar informação de cada partícula no enxame, uma unidade de *swarm* em

que as partículas são atualizadas em paralelo e um módulo de avaliação da função objetivo. Nesta arquitetura é imperativo o uso de um protocolo para controle de informação, devido a que apenas um único módulo de avaliação da função objetivo é implementado [126]. Em [126], a arquitetura serial do PSO proposta por Tewolde foi validada para as funções de teste *Esfera* e *Rosenbrock* e os tempos de execução foram comparados com uma implementação em um microprocessador *freescale* de 16 bits operando a 25MHz, mostrando que a solução em FPGAs é 359 e 653 mais rápida para as funções *Esfera* e *Rosenbrock*, respectivamente, do que a solução microprocessada.

Com relação aos outros algoritmos de inteligência de enxames citados neste capítulo é importante ressaltar que poucos estudos tem sido realizados. Avci [128] realizou uma primeira implementação em FPGAs do algoritmo ABC. No entanto não foi encontrada uma publicação sobre esse trabalho acessível à comunidade acadêmica. Por outro lado, não foram encontrados trabalhos reportados na literatura científica que apresentem implementações paralelas em FPGAs dos algoritmos FA e SFLA.

2.9 CONCLUSÕES DO CAPÍTULO

Neste capítulo foram abordados os conceitos básicos sobre otimização, considerados relevantes para o desenvolvimento do presente trabalho. O estudo do estado da arte das técnicas de otimização evidenciou uma tendência ao uso de metaheurísticas na solução de problemas complexos em que os métodos exatos, baseadas no cálculo gradiente, são de difícil implementação e conduzem a tempos de execução elevados.

Metaheurísticas baseadas em populações, dentre as quais se destacam os algoritmos evolutivos e os algoritmos de inteligência de enxames, utilizam técnicas computacionais bioinspiradas nos princípios biológicos evolutivos encontrados na natureza, tais como a seleção natural, herança genética, mutação e comportamentos coletivos para intercâmbio de informação. Neste trabalho foram escolhidos os algoritmos de otimização por inteligência de enxames devido à sua simplicidade de implementação e capacidades de paralelização, as quais podem ser exploradas visando ganhos de desempenho e melhoras na qualidade das soluções.

Neste capítulo também foi apresentada a tecnologia de *hardware* reconfigurável baseada em FPGAs, examinando a sua potencialidade para acelerar algoritmos, incrementando a taxa de processamento de dados mediante a execução de processos paralelos. Os

dispositivos FPGAs possibilitam a implementação de soluções *hardware* de alto desempenho e baixo consumo de potência se comparado com soluções convencionais em *software*, sendo assim bons candidatos para aplicações embarcadas.

Espera-se, com as implementações em arquiteturas reconfiguráveis dos algoritmos estudados ganhar em desempenho, custo, flexibilidade e eficiência de forma que as arquiteturas propostas possam ser uma alternativa de estudo em problemas de otimização embarcados.

Os conceitos apresentados neste capítulo são importantes para o entendimento das implementações de *hardware* e *software* realizadas no contexto deste trabalho. Estas implementações são descritas nos seguintes capítulos.

Capítulo 3 IMPLEMENTAÇÃO DAS UNIDADES DE CÁLCULO ARITMÉTICO E TRIGONOMÉTRICO EM PONTO FLUTUANTE

Este capítulo apresenta as implementações dos operadores de cálculo aritmético e trigonométrico, assim como os resultados de síntese, execução e desempenho. O capítulo está dividido da seguinte forma: Após algumas considerações iniciais, é descrito o padrão IEEE-754 para representação binária da aritmética de ponto flutuante. Posteriormente, apresentam-se as arquiteturas de *hardware* dos operadores implementados. Em seguida os resultados de custo em área lógica, erro associado e tempo de execução dos operadores implementados são apresentados. Finalmente é realizada uma discussão sobre os pontos mais relevantes associados aos resultados obtidos.

3.1 CONSIDERAÇÕES INICIAIS

A motivação do estudo apresentado neste capítulo está relacionado com os estudos pouco aprofundados encontrados na literatura sobre a implementação em FPGAs de operadores aritméticos e trigonométricos usando uma aritmética de ponto flutuante, principalmente quando direcionados para aplicações embarcadas. A escolha da aritmética de ponto flutuante para representação numérica obedece aos requerimentos de precisão e faixa dinâmica presentes nos problemas de otimização em diversos campos da engenharia. Neste contexto, existe uma necessidade crítica de atingir um compromisso entre desempenho, consumo de energia, consumo de recursos e precisão nas operações matemáticas, de forma que sejam atendidos os requisitos específicos para uma aplicação em sistemas embarcados.

Atualmente, as ferramentas de desenvolvimento proporcionam a possibilidade de instanciar blocos de propriedade intelectual (*IPcores*), dentre os quais podem-se encontrar unidades de cálculo aritmético em ponto flutuante. Estas ferramentas são bastante flexíveis, possibilitando a escolha do tamanho de palavra, latência, entre outras características como otimização para aplicações de alto desempenho ou baixo consumo de

recursos. Contudo, os *IPcores* fornecidos por estas ferramentas estão limitados a certas famílias de um fabricante específico. Adicionalmente, o usuário geralmente não tem acesso à descrição de *hardware* dos operadores e, portanto, não é possível realizar modificações na lógica visando melhoras em uma ou mais variáveis de projeto digital, tais como, custo em área lógica, latência, frequência de operação, erro associado, consumo de potência, etc.

Tendo em conta as aplicações em sistemas embarcados, tais como robótica, redes de sensores, extração de características, processos de aprendizado de máquinas, síntese de som, planejamento e otimização de trajetória, filtros adaptativos, entre outras; assim como as dificuldades acima descritas no uso de bibliotecas oferecidas por diversos fabricantes, foi decidido desenvolver as bibliotecas de cálculo aritmético e trigonométrico em ponto flutuante, levando em consideração três aspectos importantes: (a) descrição de *hardware* genérica que possibilite a portabilidade de plataformas (Xilinx, Altera ou Microsemi); (b) arquiteturas parametrizáveis que permitam a mudança no tamanho de palavra viabilizando uma análise de erro associado e; (c) escolha de parâmetros próprios de cada algoritmo, tais como número de iterações e número de sementes das aproximações iniciais, viabilizando ajustes de precisão, segundo o tipo de aplicação a que se destinam os operadores.

No contexto deste trabalho, foram utilizados os algoritmos *Goldschmidt's* e *Newton-Raphson* para a implementação dos operadores de divisão e raiz quadrada, assim como as expansões por séries de Taylor e o algoritmo CORDIC para a implementação dos operadores trigonométricos. As arquiteturas desenvolvidas foram validadas para diferentes tamanhos de palavra (24, 28, 32, 43 e 64 bits), obtendo dados de consumo de recursos, desempenho, erro associado, frequência de operação e, em alguns casos, estimação do consumo de potência.

Considerando a grande quantidade de resultados coletados, neste capítulo serão apresentadas unicamente as arquiteturas consideradas necessárias para o entendimento das implementações de *hardware* dos algoritmos de otimização, os quais são descritos no seguinte capítulo. No entanto, uma análise completa das implementações dos operadores, dos algoritmos utilizados e dos resultados obtidos pode ser encontrada em [129], [130], [131] e [132].

Com base na análise experimental apresentada em [129] e [132] foi possível estabelecer que a implementação do algoritmo *Goldschmidt's* (GS) para divisão e raiz quadrada

requer mais recursos de *hardware* do que o algoritmo *Newton-Raphson* (NR). Embora o custo em área da arquitetura GS seja levemente maior do que a arquitetura NR, o primeiro algoritmo alcança frequências de operação maiores devido à abordagem paralela na implementação das equações iterativas [129]. Adicionalmente, o algoritmo NR apresentou um erro associado levemente inferior em comparação com o algoritmo GS para os operadores de divisão e raiz quadrada.

Por outro lado, conforme apresentado em [130] e [131], foi possível concluir que a arquitetura de expansão por séries de Taylor (*FPTaylor*) para cálculo das funções trigonométricas é mais econômica em termos de *flip-flops* e LUTs do que as arquiteturas baseadas no algoritmo CORDIC (*FPCordic*). Este fato pode ser explicado devido a que na arquitetura *FPTaylor* apenas uma unidade *FPadd* é utilizada, enquanto a arquitetura *FPCordic* é baseada em unidades *FPadd* em paralelo [129], [132]. As frequências de operação das arquiteturas *FPCordic* são aproximadamente 70MHz para uma implementação de precisão dupla e aproximadamente 85 MHz para uma implementação de precisão simples. Isto demonstra que a representação de 32 bits é 17.6% mais rápida do que a representação de 64 bits, sendo, portanto, relevante para aplicações embarcadas de alto desempenho em que a precisão do cálculo não é um fator crítico.

A tabela 3.1 apresenta, de forma qualitativa, um quadro comparativo entre as arquiteturas GS e NR e as arquiteturas *FPCordic*/*FPTaylor* para as variáveis de projeto de custo em área lógica, erro associado e latência em ciclos de relógio. Este quadro comparativo foi construído considerando diferentes tamanhos de palavra, número de iterações dos algoritmos, entre outros parâmetros de ajuste, segundo mostrado em [129], [130] e [131]. Na tabela é possível observar que a arquitetura *Newton-Raphson* apresenta melhores resultados de custo em área, erro associado e tempo de execução com relação à arquitetura *Goldschmidt's* para o cálculo da divisão e da raiz quadrada. De forma similar, pode-se concluir que a arquitetura *FPTaylor* apresenta um custo em área e uma latência menor se comparado com a arquitetura *FPCordic*, porém o erro associado para o cálculo das funções *sin*, *cos* e *atan* é consideravelmente maior para a arquitetura *FPTaylor*.

Tabela 3.1: Quadro comparativo entre arquiteturas

Operação	Comparação	Custo em área	Erro associado	Latência
Divisão	GS/NR	Maior	Levemente maior	Levemente maior
Raiz quadrada	GS/NR	Maior	Igual	Maior
sin, *cos*, *atan*	*FPCordic*/*FPTaylor*	Maior	Muito menor	Maior

Os resultados de síntese obtidos durante a análise das arquiteturas propostas demonstraram que o consumo de blocos DSPs é um fator crítico no desenvolvimento em FPGAs dos operadores aritméticos e trigonométricos [129], [130]. Com base no anteriormente descrito e, após a análise de *tradeoff* entre consumo de recursos, erro associado e tempo de execução, foram escolhidos os algoritmos *Newton-Raphson* para a implementação dos operadores de divisão e raiz quadrada e o algoritmo CORDIC para a implementação dos operadores trigonométricos.

Neste capítulo apresentam-se arquiteturas melhoradas dos operadores de soma e subtração ($FPadd$) e de multiplicação ($FPmul$) assim como dos algoritmos *Newton-Raphson* e CORDIC, as quais visam um consumo menor de blocos DSPs e uma frequência de operação maior sem detrimento do erro associado. As arquiteturas melhoradas das unidades $FPadd$ e $FPmul$ permitiram um aumento na frequência de operação dos operadores de cálculo trigonométrico. Adicionalmente, se comparado com os resultados apresentados em [129], [132], a nova arquitetura do algoritmo NR para divisão apresenta uma economia de blocos DSPs de 5.5 e 2.7 vezes para tamanhos de palavra de 32 e 64 bits, respectivamente. De forma similar, a nova arquitetura NR para raiz quadrada apresenta uma economia de blocos DSPs de 6.5 e 5.1 vezes para tamanhos de palavra de 32 e 64 bits, respectivamente, se comparado com os resultados apresentados em [129], [132].

É comum encontrar na literatura implementações de *hardware* em ponto flutuante de 32 e 64 bits, sendo a implementação de 32 bits a mais utilizada devido ao baixo consumo de recursos. Entretanto, uma implementação em ponto flutuante de 27 bits (1 bit de sinal, 8bits de expoente e 18 bits de mantissa) constitui uma implementação mais eficiente em termos de consumo de recursos para desenvolvimento em FPGAs. Isto é devido a que os blocos DSPs usados nos dispositivos FPGAs são baseados em multiplicadores de 9x9 nas famílias Cyclone e Stratix da Altera, ou 18x18 e 18x25 nas famílias Spartan e Virtex da Xilinx, respectivamente. Desta forma, implementações de *hardware* em ponto flutuante com mantissas de 18 bits usam menos blocos DSP do que implementações de precisão simples, as quais usam mantissas de 23 bits. Neste contexto, o presente capítulo também apresenta uma análise comparativa dos resultados de síntese para representações de 27, 32 e 64 bits. É importante salientar que a implementação de 27 bits possibilita uma faixa dinâmica similar à implementação de 32 bits, porém com um aumento no erro associado, como será explicado na análise dos resultados.

3.2 O PADRÃO IEEE-754

O padrão IEEE-754 é uma representação numérica em formato binário para aritmética de ponto flutuante [133], caracterizado por três componentes: (a) um bit de sinal S, (b) um expoente E com E_w bits e (c) uma mantissa M com M_w bits, como mostrado na figura 3.1. A mantissa representa a magnitude do número e uma constante chamada de *bias* é adicionada ao expoente no intuito de trabalhar com expoentes não negativos.

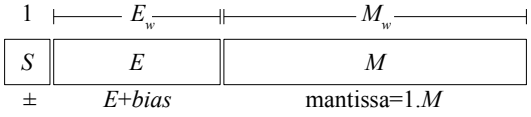


Figura 3.1: O padrão IEEE-754

Este padrão permite ao projetista trabalhar tanto com precisão simples (32 bits) e dupla (64 bits) como também com um formato adaptado aos requerimentos de precisão e de faixa dinâmica de uma aplicação específica, por exemplo, a representação de 27 bits descrita na seção anterior. Um formato de alta precisão indica poucos erros de quantização na implementação final, enquanto que um formato com baixa precisão indica implementações de alta velocidade e reduções em área e consumo de potência [134].

3.3 IMPLEMENTAÇÃO DOS OPERADORES ARITMÉTICOS

Algoritmos de otimização aplicados em engenharia modelam os problemas usando uma função custo, que comumente requerem do cálculo de operações aritméticas. O cálculo de operadores aritméticos em ponto flutuante é um requisito essencial em sistemas computacionais de alto desempenho, especificamente em aplicações de engenharia, em que a aritmética de ponto flutuante fornece uma faixa dinâmica para representar números reais pequenos e grandes com capacidade de manter a sua precisão [40], [41].

3.3.1 Soma/subtração em ponto flutuante (*FPadd*)

Os passos para a execução da soma/subtração em ponto flutuante são os seguintes:

1. Separar o sinal, expoente e mantissa dos operandos de entrada. Conferir se as entradas são zero, infinito ou uma representação inválida no padrão IEEE-754. Se os operandos são válidos adicionar o bit implícito '1' às mantissas.

2. Comparar expoentes e mantissas dos dois operandos. Deslocar à direita o menor dos números. O número de bits a deslocar correspondente ao resultado da subtração dos expoentes, sendo este valor o resultado preliminar do expoente. Somar ou subtrair as mantissas segundo a operação desejada.

3. Deslocar à esquerda o resultado atual da mantissa até encontrar o valor de '1' no bit mais significativo (MSB). Para cada bit deslocado, subtrair '1' no valor atual do expoente. Finalmente, concatenar os resultados do sinal, expoente e mantissa.

A figura 3.2 descreve os blocos funcionais da arquitetura de *hardware* da unidade de soma/subtração que em diante será referenciada como *FPadd*. Os processos sensíveis à borda de subida do relógio estão representados pelos blocos pontilhados. Esta arquitetura utiliza dois ciclos de relógio para realizar o cálculo da soma/subtração em ponto flutuante. No primeiro ciclo de relógio são realizados os primeiros dois passos do algoritmo, isto é avaliar algumas exceções, comparar os expoentes e mantissas, calcula-se o sinal do resultado, o resultado preliminar do expoente e o alinhamento das mantissas segundo a diferença dos expoentes dos valores de entrada. No segundo ciclo de relógio é realizada a normalização da mantissa do resultado, sendo esse valor concatenado com o resultado do expoente e do sinal. Uma *máquina de estados finitos* (FSM) controla o processo de atualização do resultado final.

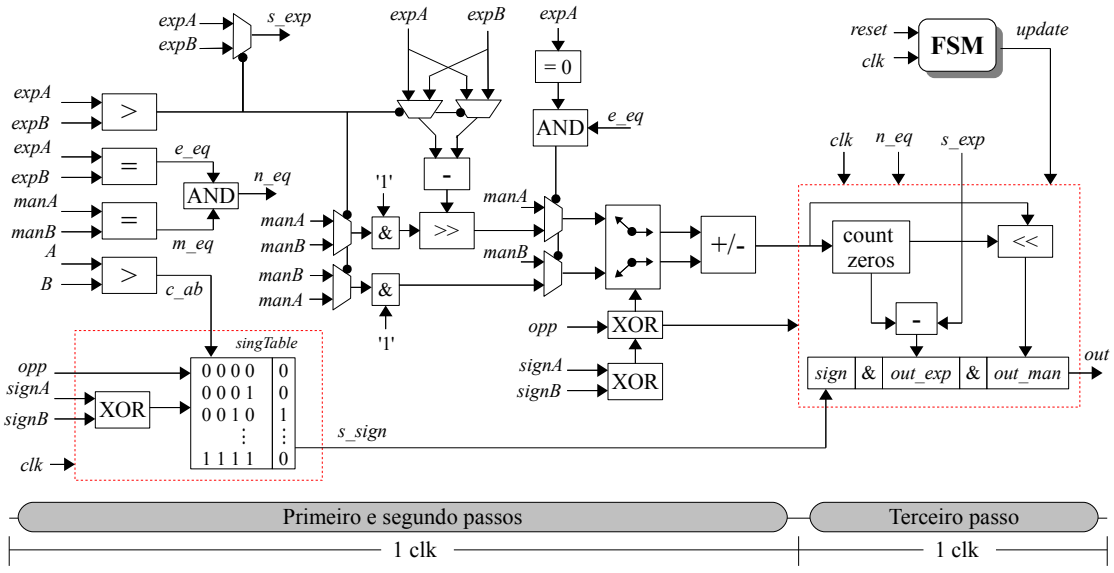


Figura 3.2: Arquitetura da unidade de soma/subtração *FPadd*

3.3.2 Multiplicação em ponto flutuante (*FPmul*)

Os passos para a execução da multiplicação em ponto flutuante são os seguintes:

1. Separar o sinal, expoente e mantissa dos operandos de entrada. Conferir se as entradas são zero, infinito ou uma representação inválida no padrão IEEE-754. Se os operandos são válidos adicionar o bit implícito '1' às mantissas.

2. Multiplicar as mantissas. Somar os expoentes e subtrair o valor de bias ao resultado ($E_A + E_B - bias$). Determinar o sinal do resultado da multiplicação ($S_A \; xor \; S_B$).

3. Se o bit mais significativo do resultado da mantissa é '1' então não é necessário uma etapa de normalização. Caso contrário, deslocar à esquerda o resultado atual da mantissa até encontrar o valor de '1' no bit mais significativo. Para cada bit deslocado, subtrair '1' no valor do expoente. Finalmente, concatenar os resultados do sinal, expoente e mantissa.

A figura 3.3 descreve a arquitetura de *hardware* da unidade de multiplicação (*FPmul*), a qual está divida em duas etapas, sendo necessário um ciclo de relógio para cada uma. Na primeira etapa avalia-se se algum dos operandos é zero, ou infinito. Adicionalmente, calcula-se de forma simultânea o sinal do resultado, o resultado preliminar do expoente e a multiplicação das mantissas. Uma operação lógica *xor* é utilizada para comparar o sinal dos valores de entrada e determinar o sinal do produto. Os expoentes são adicionados com um bit extra indicando *overflow*, resultando em um valor com *bias* duplo, motivo pelo qual o valor do *bias* é subtraído do resultado da soma dos expoentes. Por outro lado, as mantissas de M_w bits junto com o bit implícito são multiplicadas resultando em uma palavra de $2(M_w + 1)$ bits que é truncada nos primeiros $M_w + 2$ bits.

Na segunda etapa avalia-se se o MSB do resultado da multiplicação das mantissas é '1'. Este bit controla um multiplexador que permite selecionar entre os M_w bits mais significativos do valor atual da mantissa (neste caso a mantissa já está normalizada) ou o valor da mantissa deslocado à esquerda em uma posição. Caso o MSB do resultado da multiplicação das mantissas seja igual a '1' o resultado preliminar do expoente deve ser incrementado em '1'. Posteriormente, avalia-se se existe um *overflow* no resultado do expoente ou se algum dos operandos de entrada são infinito na representação IEEE-754. Caso afirmativo então o resultado final do produto é infinito, caso contrário

concatenam-se os valores do sinal, expoente e mantissa. Uma máquina de estados controla o processo de atualização do resultado final.

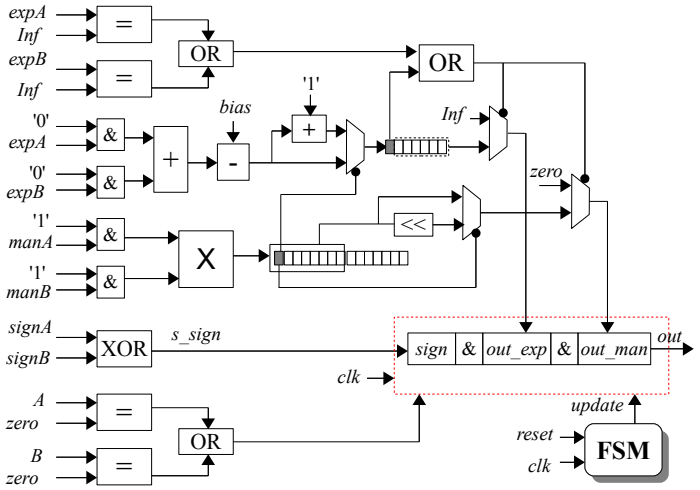


Figura 3.3: Arquitetura da unidade de multiplicação *FPmul*

3.3.3 Divisão em ponto flutuante

Sejam X e Y dois números reais representados no padrão IEEE-754, em que X representa o dividendo e Y o divisor. Um algoritmo genérico para divisão em ponto flutuante é descrito a seguir:

1. Separar o sinal, expoente e mantissa dos operandos X e Y. Conferir se as entradas são zero, infinito ou uma representação inválida no padrão IEEE-754. Se os operandos são válidos adicionar o bit implícito '1' às mantissas.

2. Calcular o resultado da mantissa utilizando um algoritmo de divisão em *ponto fixo*. Calcular o resultado do expoente usando a seguinte expressão ($E_X - E_Y + bias$) e determinar o sinal do resultado ($S_X \; xor \; S_Y$). Finalmente, concatenar os resultados do sinal, expoente e mantissa.

3.3.3.1 Algoritmo *Newton-Raphson* para divisão

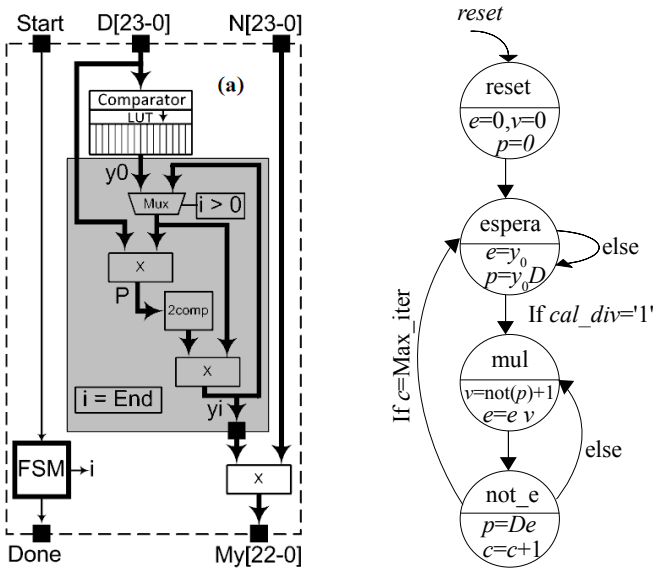
O algoritmo *Newton-Raphson* para divisão considera duas entradas N e D, de n bits cada uma, que satisfazem $N = 1$ e $D < 2$. Começando por uma aproximação inicial

$y_0 = 1/D$, as equações 3.1a e 3.1b devem ser executadas de forma iterativa [135].

$$p = Dy_i \tag{3.1a}$$

$$y_{i+1} = y_i(2 - p) \tag{3.1b}$$

Após da i^{th} iteração, uma aproximação de N/D é encontrada multiplicando $N \cdot y_{i+1}$. O algoritmo *Newton-Raphson* utiliza o valor de D para refinar, a cada iteração, a aproximação inicial de $1/D$ e na última iteração multiplica esta aproximação pelo valor de N [135]. Este fato permite ao algoritmo *Newton-Raphson* utilize menos multiplicações do que algoritmos que aproximam o valor de N/D a cada iteração [129]. A implementação *hardware*, referenciada como *divNR*, e sua respectiva descrição em máquinas de estados é mostrada na figura 3.4.



(a) arquitetura (b) FSM

Figura 3.4: Divisão *Newton-Raphson* (Modificado Sánchez, 2009 [136])

3.3.4 Raiz quadrada em ponto flutuante

Seja X um número real representado no padrão IEEE-754, em que X representa o radicando. Um algoritmo genérico para raiz quadrada em ponto flutuante é descrito a seguir:

1. Separar o sinal, expoente e mantissa da entrada X. Conferir se a entrada é zero, infinito ou uma representação inválida no padrão IEEE-754. Se a entrada é válida

adicionar o bit implícito '1' à mantissa de X. Se o expoente de X é um número par, então multiplicar a mantissa por 2.

2. Calcular o resultado da raiz quadrada da mantissa utilizando um algoritmo de raiz quadrada em *ponto fixo*. Calcular o resultado do expoente usando a expressão $(X + bias)/2$.

3. Finalmente, concatenar os resultados do sinal, expoente e mantissa e remover o bit implícito da mantissa.

De forma similar ao caso da divisão, este algoritmo trabalha principalmente sobre a mantissa, sendo o expoente calculado de forma independente. Isto permite que a mantissa seja processada utilizando operações aritméticas em ponto fixo.

3.3.4.1 Algoritmo *Newton-Raphson* para raiz quadrada

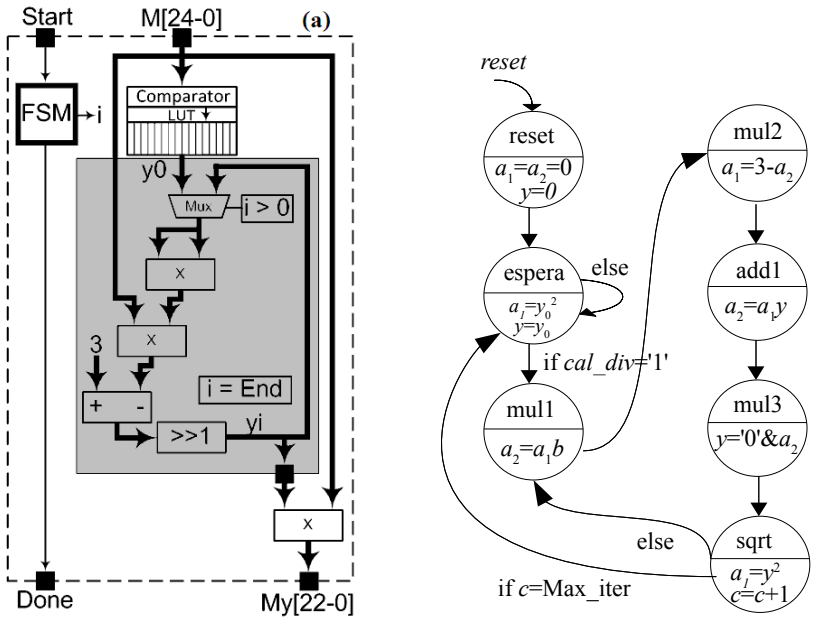
Seja b uma variável de entrada, o algoritmo calcula $sqrt(b)$, começando por uma aproximação inicial $1/sqrt(b)$ e o resultado é refinado pela seguinte equação iterativa.

$$y_{i+1} = 0.5 \cdot y_i(3 - by_i^2) \tag{3.2}$$

A cada iteração a variável y armazena uma aproximação de $1/sqrt(b)$. Finalmente multiplicando $y_{i+1}b$ é calculado o valor de $sqrt(b)$ [137]. A implementação *hardware*, referenciada como *sqrtNR*, e sua respectiva descrição em máquinas de estados é mostrada na figura 3.5

3.4 IMPLEMENTAÇÃO DOS OPERADORES TRIGONOMÉTRICOS

No presente trabalho foi utilizado o algoritmo CORDIC (*Coordinate Rotational Digital Computer*) para as implementações em *hardware* das funções seno, cosseno, arcotangente e exponencial. Estas funções foram escolhidas para serem implementadas em *hardware* devido à sua ampla aplicabilidade em problemas de engenharia. No contexto deste trabalho as funções trigonométricas são usadas tanto na execução de movimento das partículas quanto na avaliação das funções de teste *benchmark* utilizadas para validar os algoritmos de otimização propostos.



(a) arquitetura (b) FSM

Figura 3.5: Raiz quadrada *Newton-Raphson* (Modificado Sánchez, 2009 [136])

3.4.1 Algoritmo CORDIC

O algoritmo CORDIC é um método iterativo baseado em somas e deslocamento de bits utilizado inicialmente para o cálculo de rotação de um vetor bidimensional e para o cálculo do comprimento e ângulo de um vetor [40], [41]. As equações iterativas do algoritmo CORDIC são as seguintes:

$$X_{i+1} = X_i - m\sigma_i 2^{-i} Y_i \tag{3.3}$$

$$Y_{i+1} = Y_i - \sigma_i 2^{-i} X_i \tag{3.4}$$

$$Z_{i+1} = Z_i - \sigma_i \theta_i \tag{3.5}$$

em que i é o índice da iteração, θ_i são microrrotações pré-calculadas, σ_i é a direção da rotação e m indica o tipo de coordenada. O algoritmo CORDIC pode operar em coordenadas circulares (m=1) e hiperbólicas (m=-1). A seguir é feita uma breve descrição de cada caso.

- Coordenadas circulares (m=1)

Neste caso, as microrrotações pré-calculadas são definidas pela expressão $\theta_i = atan(2^{-i})$ e $\sigma_i = -1$ se $Z_i < 0$ ou $+1$, caso contrario. Um fator de escala $K = \prod cos(atan(2^{-i}))$

deve ser aplicado. Para calcular as funções $sin(\theta)$ e $cos(\theta)$ os seguintes valores iniciais devem ser usados: $X_0 = K$, $Y_0 = 0$, e $Z_0 = \theta$. Para calcular a função $atan(y/x)$ os seguintes valores iniciais devem ser usados: $X_0 = 1$, $Y_0 = Y$, e $Z_0 = 0$. Uma redução de argumento quadrática pode ser aplicada visando calcular o seno ou cosseno de grandes argumentos de entrada de forma que $sin(\theta)$ ou $cos(\theta)$ é um dos seguintes valores $\pm\{sin(\alpha), cos(\alpha)\}$, sendo α o argumento reduzido que pode ser calculado pela equação 3.6.

$$\alpha = \delta\pi/2, \delta = (2\theta/\pi) - \kappa \tag{3.6}$$

em que κ e δ são a parte inteira e a parte decimal de $2\theta/\pi$, respectivamente. Adicionalmente, o valor $mod(\kappa, 4)$ determina qual dos operadores $sin(\alpha)$ ou $cos(\alpha)$ deve ser calculado, como explicado por Smith [138]. Portanto, o problema da redução do argumento é resolvido mediante a realização de duas multiplicações, uma subtração e o cálculo de $mod(\kappa, 4)$. Note-se que os dois bits menos significativos da variável κ indicam o valor de $mod(\kappa, 4)$.

- Coordenadas hiperbólicas (m=-1)

Neste caso, as microrrotações pré-calculadas são definidas pela expressão $\theta_i = atanh(2^{-i})$ e $\sigma_i = -1$ se $Z_i < 0$ ou $+1$, caso contrario. Um fator de escala $K' = \prod cos(atanh(2^{-i}))$ deve ser aplicado. Para calcular as funções $sinh(\theta)$ e $cosh(\theta)$ os seguintes valores iniciais devem ser usados: $X_0 = K'$, $Y_0 = 0$, e $Z_0 = \theta$. Para calcular a função exponencial ($exp(\theta)$) pode-se utilizar a identidade $exp(\theta) = sinh(\theta) + cosh(\theta)$ ou as equações 3.3 e 3.4 podem ser somadas, obtendo-se a equação 3.7.

$$W_{i+1} = W_i - \sigma_i 2^{-i} W_i \tag{3.7}$$

com os seguintes valores iniciais: $W_0 = K'$ e $Z_0 = \theta$. Desta forma, o algoritmo CORDIC para cálculo da função *exponencial* é simplificado a duas equações (3.5 e 3.7), permitindo a economia de uma unidade *FPadd*.

Uma das desvantagens do algoritmo CORDIC original para cálculo da função exponencial é que os argumentos de entrada estão restritos à faixa [0,1]. No entanto, esse problema pode ser facilmente resolvido utilizando a identidade mostrada na equação 3.8.

$$\exp(x) = 2^{x_i} \exp(x_f / \log_2(e)) = 2^{x_i} \exp(y) \tag{3.8}$$

sendo x_i e x_f as partes inteira e decimal de $x \log_2(e)$, respectivamente. Perceba-se que o valor de $y = x_f / \log_2(e)$ pertence à faixa [0,1] e, portanto, $\exp(y)$ pode ser calculada

com alta precisão usando o algoritmo CORDIC básico. O fator 2^{x_i} corresponde a uma operação de deslocamento.

O algoritmo CORDIC possui dois modos de operação: o modo rotação e o modo vetorização. No modo rotação o intuito é utilizar as microrrotações para levar a variável Z até um valor próximo do zero. Já no modo vetorização as microrrotações levam a variável Y até um valor próximo do zero. A tabela 3.2 resume as funções que podem ser calculadas pelo algoritmo CORDIC nos modos de rotação ou vetorização, para cada um dos sistemas de coordenadas.

Tabela 3.2: Modos de operação do algoritmo CORDIC

m	Modo rotação $Z_0 \rightarrow 0$	Modo vetorização $Y_0 \rightarrow 0$
1	$x_n = (x\cos(z) - y\sin(z))/K$	$x_n = \sqrt{x^2 + y^2}/K$
	$y_n = (y\cos(z) - x\sin(z))/K$	$z_n = z + \arctan(y/x)$
0	$x_n = x$	$x_n = x$
	$y_n = y + xz$	$z_n = z + y/x$
-1	$x_n = (x\cosh(z) - y\sinh(z))/K'$	$x_n = \sqrt{x^2 + y^2}/K'$
	$y_n = (y\cosh(z) - x\sinh(z))/K'$	$z_n = x + arctanh(y/x)$

3.4.2 Implementação CORDIC das funções trigonométricas

O algoritmo CORDIC para cálculo das funções seno/cosseno ou arco-tangente se diferenciam nas condições iniciais utilizadas para executar os dois modos de operação, como mostrado na tabela 3.2. As equações 3.3, 3.4 e 3.5 estabelecem que cada iteração o algoritmo CORDIC requer calcular três somas/subtrações e uma busca em memória, na qual são armazenadas as microrrotações pré-calculadas.

A figura 3.6 apresenta a arquitetura proposta para a implementação em *hardware* do algoritmo CORDIC para cálculo das funções seno e cosseno, referenciada como *Cordicsincos*. Esta arquitetura está constituída por quatro unidades: (a) *unidade de redução de argumento*, (b) *unidade de microrrotação*, (c) *unidade de seleção* e (d) *unidade FSMCordic*.

A *unidade de redução de argumento*, definida pela equação 3.6, é implementada usando uma unidade *FPmul* e uma unidade *FPfrac*, a qual separa a parte inteira κ e decimal δ de um número com representação em ponto flutuante. Inicialmente o argumento

de entrada A é multiplicado pela constante $2/\pi$ e o resultado é apresentado como entrada da unidade *FPfrac* obtendo as saídas κ e δ. Finalmente, o argumento reduzido (α) é obtido calculando $\delta\pi/2$. A *unidade FSMCordic* seleciona a condição inicial das variáveis X, Y e Z e controla a direção da microrrotação além de realizar o processo de realimentação dos valores da microrrotação atual.

A estrutura básica da *unidade de microrrotação* são três unidades *FPadd* em paralelo. Em uma representação de ponto flutuante, as multiplicações pelos fatores 2^{-i} das equações 3.3 e 3.4 são calculadas subtraindo o valor da iteração atual (i) do valor do expoente das variáveis X e Y. Após um número pré-definido de iterações, as saídas $sin(\alpha)$ e $cos(\alpha)$ apresentam valores válidos. A *unidade de seleção* realiza a seleção do quadrante, o qual direciona às saídas um dos quatro resultados: $\pm\{\sin(\alpha), \cos(\alpha)\}$. Observe-se que o quadrante é calculado usando os dois bits menos significativos do registro κ, os quais correspondem ao valor $mod(\kappa, 4)$.

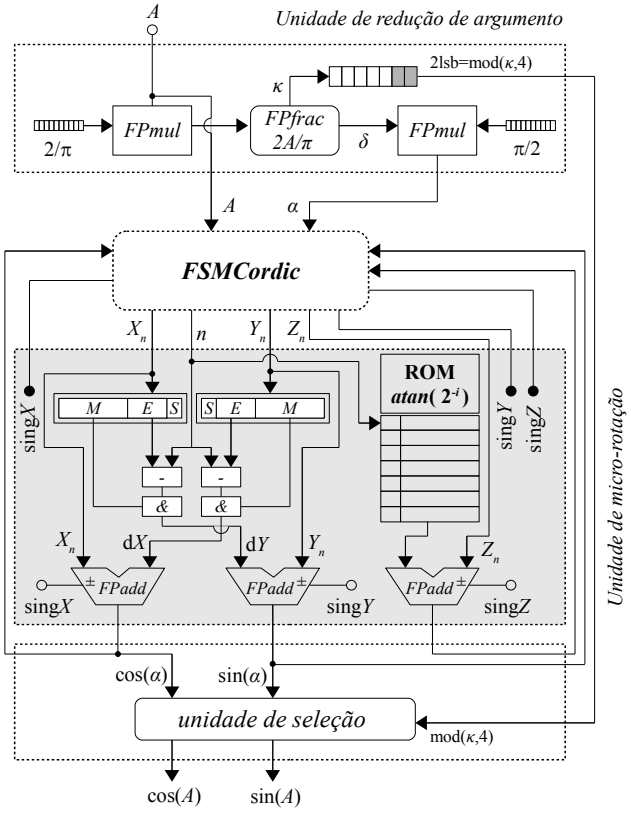


Figura 3.6: Arquitetura iterativa da unidade *Cordicsincos*

A figura 3.7 apresenta a arquitetura proposta para a implementação em *hardware* do algoritmo CORDIC para cálculo da função arco-tangente, referenciada como *Cordicatan*. Esta arquitetura funciona de forma similar à arquitetura CORDIC no modo

rotação para cálculo das funções seno e cosseno com a diferença de que não são mais necessárias as etapas de redução de argumento e seleção do quadrante.

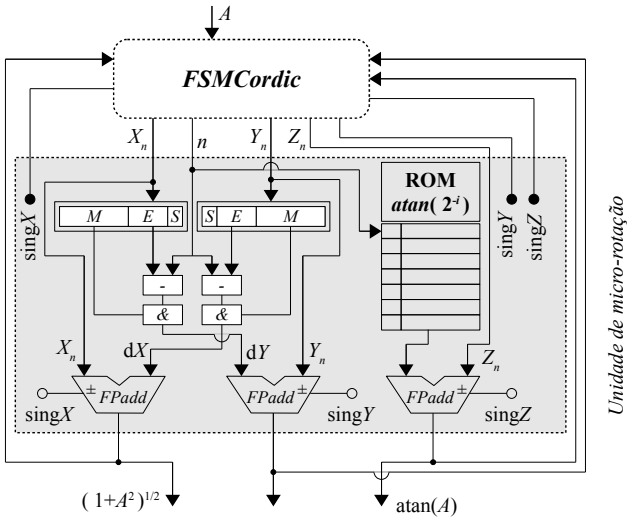


Figura 3.7: Arquitetura iterativa da unidade *Cordicatan*

3.4.3 Implementação CORDIC para a função exponencial

A figura 3.8 apresenta a arquitetura proposta para a implementação em *hardware* da função exponencial. Esta arquitetura, referenciada como *CordicTaylorexp*, faz uso do algoritmo CORDIC em coordenadas hiperbólicas e de uma expansão por séries de Taylor de segunda ordem. A arquitetura está constituída pelos seguintes componentes: (a) *unidade de redução de argumento*, (b) *unidade de microrrotação*, (c) *unidade FSMCordic*, (d) *unidade FSMTaylor* e (e) *unidade de normalização*.

A unidade de redução de argumento utiliza uma unidade *FPmul* para multiplicar o argumento de entrada (Z) pelo valor $2\log_2(e)$, e o resultado desta multiplicação é separado nas suas partes inteira (Z_i) e decimal (Z_f) pelo componente *FPfrac*. Posteriormente, a parte decimal é multiplicada pelo fator $1/\log_2(e)$ no intuito de obter o argumento reduzido (Z_0) do argumento de entrada Z.

Dependendo do valor do argumento reduzido Z_0, duas lógicas são multiplexadas: o algoritmo CORDIC em coordenadas hiperbólicas ou a expansão em séries de Taylor de segunda ordem. Esta separação de abordagens foi realizada no intuito de diminuir o erro de aproximação quando os argumentos reduzidos são pequenos ($|Z_0| < 0.05$).

Estudos numéricos feitos no contexto deste trabalho têm demonstrado que o algoritmo CORDIC em coordenadas hiperbólicas apresenta erros grandes quando o argumento de entrada é pequeno [139], [129]. Portanto, uma expressão em séries de Taylor pode ser útil nesta situação, oferecendo assim uma aproximação mais precisa dentro de uma faixa de argumentos de entrada maior, como apresentado por Muñoz *et al.* [129].

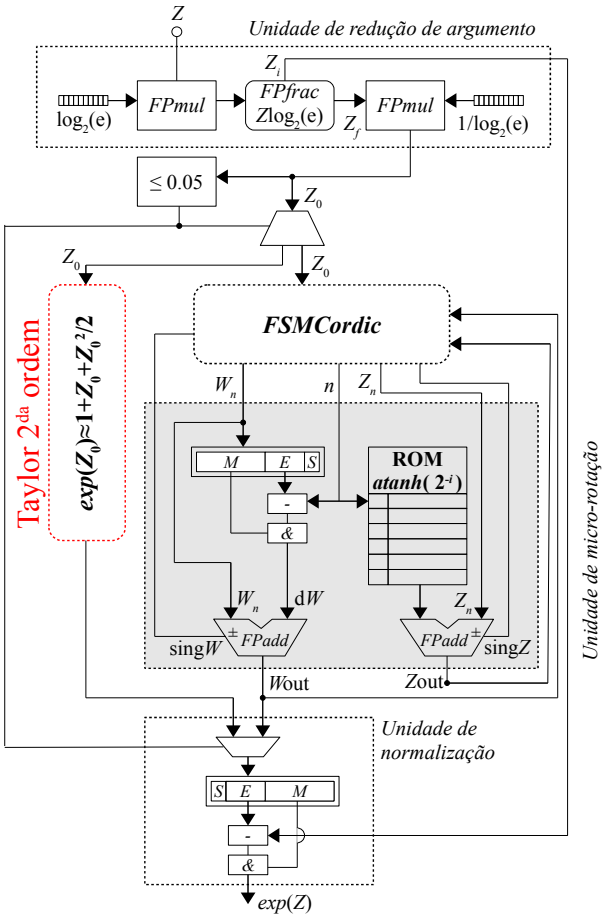


Figura 3.8: Arquitetura iterativa da unidade *CordicTaylorexp*

É importante destacar que a expansão por séries de Taylor de segunda ordem foi implementada utilizando as mesmas unidades *FPmul* e *FPadd* utilizadas na unidade de redução de argumento, permitindo uma economia no consumo de recursos. Por outro lado, a *unidade de microrrotação* em coordenadas hiperbólicas é implementada de forma simular ao caso circular, porém, apenas duas unidades paralelas *FPadd* são requeridas (vide equações 3.5 e 3.7). Finalmente, a *unidade de normalização* realiza a operação $2^{\pm Z_i} \exp(Z_0)$, o qual é feito somando ou subtraindo o valor inteiro (Z_i) do resultado parcial do expoente $\exp(Z_0)$.

3.5 RESULTADOS DE SÍNTESE

Os operadores descritos anteriormente foram mapeados para as representações de 27, 32 e 64 bits no formato IEEE-754. Neste trabalho o erro associado foi tratado como variável de projeto, permitindo assim uma melhor análise da precisão dos operadores com relação às outras três variáveis de projeto: custo em área lógica, tempo de execução e consumo de potência [129] e [130].

Os operadores de ponto flutuante foram descritos na linguagem de descrição de *hardware* VHDL e implementados usando a ferramenta de desenvolvimento ISE10.1 para FPGAs das famílias Virtex5 e Spartan3E (dispositivos xc5vlx110T xc3s1600E) da Xilinx. As arquiteturas desenvolvidas também foram mapeadas em FPGAs CycloneII e CycloneIV da Altera, para o qual utilizou-se a ferramenta de desenvolvimento QuartusII. Maiores detalhes dos resultados de síntese para as implementações dos operadores mapeados na tecnologia da Altera podem ser encontradas em [140] e [141]. Contudo, neste capítulo apresentam-se os resultados obtidos para as implementações na família Virtex5 da Xilinx, sendo esta a plataforma de desenvolvimento e teste para os algoritmos de otimização por inteligência de enxames estudados no seguinte capítulo.

O custo em área lógica em termos de *flip-flops* (FF), *LookUp Tables* (LUTs) e blocos dedicados de processamento digital (DSP48Es), assim como a frequência de operação do circuito, expressada em Mega-Hertz (MHz), é apresentada na tabela 3.3. Uma LUT de 16 palavras foi utilizada para armazenar as sementes iniciais das implementações *Newton-Raphson* para os operadores de divisão e raiz quadrada. Uma LUT de 20 palavras foi utilizada para armazenar os valores pré-calculados das microrrotações do algoritmo CORDIC.

Como esperado, as representações com um tamanho de palavra maior possuem um custo em área lógica maior e um desempenho menor do que as representações com tamanho de palavras menores. Pode-se observar que todos os operadores aritméticos e trigonométricos são satisfatoriamente implementados em termos dos recursos disponíveis no dispositivo FPGA selecionado.

Esta análise experimental do custo em área lógica permite concluir que a unidade *FPadd* possui um consumo de lógica combinacional (consumo de LUTs) alto se comparado com os outros operadores aritméticos, entre os quais o multiplicador *FPmul*

Tabela 3.3: Resultados de síntese dos operadores implementados (dispositivo xc5vlx110t)

Tamanho (Exp,Man)	Operador em ponto flutuante	FF 69120	LUTs 69120	DSP48E 64	Freq. MHz
	FPadd	151	410	0	164.486
	FPmul	30	126	1	130.764
27	*FPdivNR*	129	378	1	121.654
(8,18)	*FPsqrtNR*	158	301	2	119.175
	Cordicsincos	493	1741	1	130.764
	Cordicatan	301	1305	0	164.648
	CordicTaylorexp	368	1376	1	130.764
	FPadd	219	540	0	160.352
32	*FPmul*	35	95	2	137.457
(8,23)	*FPdivNR*	159	437	2	109.878
precisão	*FPsqrtNR*	193	439	2	101.729
simples	*Cordicsincos*	582	2125	2	137.457
	Cordicatan	334	1543	0	130.189
	CordicTaylorexp	433	1678	2	137.457
	FPadd	371	1023	0	136.042
64	*FPmul*	67	395	15	63.118
(11,52)	*FPdivNR*	336	1109	15	60.709
precisão	*FPsqrtNR*	402	862	12	64.708
dupla	*Cordicsincos*	1142	5028	15	65.690
	Cordicatan	699	3749	0	130.572
	CordicTaylorexp	846	3961	15	65.690

requer menos recursos, porém faz uso de blocos DSPs embarcados. Entre os operadores trigonométricos, o operador *Cordicatan* possui o menor consumo de recursos de *hardware*. Isto pode ser explicado dado que esse operador não requer das unidades de redução de argumento. Por outro lado, entre os operadores trigonométricos que requerem de redução de argumento, o operador *Cordicsincos* utiliza três unidades *FPadd* em paralelo e, portanto, possui um consumo de *flip-flops* e de lógica combinacional maior do que o operador *CordicTaylorexp* que utiliza apenas duas unidades *FPadd* em paralelo.

Observa-se que as unidades *FPadd* e *Cordicatan*, as quais não utilizam multiplicadores, atingem as maiores frequências de operação para todos os tamanhos de palavra, atingindo aproximadamente 164MHz no melhor caso (implementação de 27 bits) e 130 MHz para uma implementação de 64 bits (pior caso).

É importante salientar que o dispositivo FPGA utilizado (xc5vlx110t) não é o dispositivo de maior capacidade da família Virtex5-LX. De acordo com os resultados de síntese

dos operadores aritméticos o consumo de recursos, na pior situação (64bits), foi de 1.6% de LUTs e de 23.4% dos blocos DSPs disponíveis. Para os operadores trigonométricos o consumo de recursos foi de 7.3% de LUTs e de 23.4% de blocos DSPs.

3.6 VALIDAÇÃO DAS ARQUITETURAS PROPOSTAS

Nesta seção apresentam-se os resultados de validação das arquiteturas propostas após a execução no dispositivo FPGA. As melhores condições de operação dos algoritmos implementados em *hardware* foram escolhidas com base nas simulações apresentadas em [130], [129], [132], [131]. Os principais resultados destas simulações são descritos a seguir.

Visando analisar o impacto do formato utilizado pelo padrão IEEE-754 (tamanho de palavra) no erro associado, assim como a influência do número de iterações no tempo de execução, os experimentos consideraram os seguintes fatores: (a) tamanhos de palavra, (b) número de sementes dos algoritmos GS e NR e (c) número de iterações ou microrrotações. Para isto, foi usada a ferramenta ModelSim [142] após o processo PAR (*placement and routing*) e um ambiente de simulação construído no Matlab no qual é possível gerar os vetores de teste aleatórios e decodificar os resultados.

Em [130] apresenta-se uma análise dos operadores aritméticos levando em consideração o tamanho de palavra, erro associado, o tempo de execução em ciclos de relógio e o tamanho da memória para armazenar as sementes iniciais dos algoritmos de divisão e raiz quadrada. Os resultados de simulação obtidos apontam que o algoritmo NR atinge as melhores condições de operação para três iterações e 16 palavras para armazenar as sementes iniciais, sendo necessários 11 e 16 ciclos de relógio para a divisão e raiz quadrada, respectivamente.

Com relação aos operadores trigonométricos, em [129] apresenta-se uma análise das condições de funcionamento em função do tamanho de palavra, erro associado e número de microrrotações do algoritmo CORDIC. Nas simulações observou-se que o cálculo da função arco-tangente possui um erro associado maior do que as unidades CORDIC para cálculo das funções seno, cosseno e exponencial. Entretanto, foi observado um incremento do erro no cálculo destas três ultimas funções quando as unidades de redução de argumento são usadas, isto é, quando o argumento de entrada está fora da faixa

$[-\pi/2, \pi/2]$ e [-1,1] para as funções seno/cosseno e exponencial, respectivamente. Na arquitetura proposta em [129] cada microrrotação requer de 4 ciclos de relógio e o cálculo da redução de argumento requer de 12 ciclos de relógio, portanto, cada execução do algoritmo CORDIC requer de $4 * NI + 12$ ciclos de relógio, sendo NI o número total de microrrotações.

A seguir são apresentados os resultados obtidos pelas arquiteturas descritas na seção anterior. É importante salientar que estas arquiteturas diferem das arquiteturas apresentadas em [130], [129] no sentido de que a descrição *hardware* dos circuitos foi aprimorada visando uma economia de recursos, especificamente de blocos DSPs, e um aumento da frequência de operação. Contudo, a lógica dos algoritmos NR e CORDIC foi mantida e, em consequência, não houve um detrimento do erro associado. As condições experimentais são as seguintes:

- Três iterações do algoritmo NR para divisão e raiz quadrada.
- Memória de 16 posições para armazenamento das aproximações iniciais do algoritmo NR para divisão e raiz quadrada.
- 15 microrrotações do algoritmo CORDIC para cálculo das funções seno, cosseno e arco-tangente.
- 25 microrrotações do algoritmo CORDIC para cálculo da função exponencial.
- Validações para tamanhos de palavra de 27, 32 e 64 bits. As representações de 32 e 64 bits foram escolhidas por serem as mais amplamente usadas na literatura além de permitir uma comparação mais eficiente com soluções baseadas em *software*. A representação de 27 bits foi escolhida pelo baixo custo em área lógica (vide resultados de síntese), demonstrando ser mais eficiente em termos de recursos de *hardware* para implementações em FPGA.
- Os vetores de teste usados para validação consideram exceções como divisão por zero, raiz quadrada de número negativos, multiplicação por zero, multiplicação por um, soma/subtração de operandos iguais, arco-tangente de $\pi/2$, etc. Adicionalmente outros valores das mantissas e expoentes considerados importantes para efeitos de validação foram utilizados, por exemplo, combinações das seguintes condições: mantissa igual a zero, mantissa igual ao valor máximo $(2^{FW} - 1)$, sendo FW o número de bits da mantissa, e expoente representando infinito $(2^{EW} - 1)$, sendo EW o número de bits do expoente.

- Os operadores aritméticos foram validados para valores na faixa [-1000,1000]. A raiz quadrada foi validada para valores da faixa [0,1000]. Por outro lado, os operadores seno e cosseno foram validados para argumentos de entrada na faixa $[-100\pi, 100\pi]$ e os operadores para cálculo das funções arco-tangente e exponencial na faixa de valores [-100,100] e [-10,10], respectivamente.

- A comunicação serial RS232 foi usada para comunicação entre o FPGA e o ambiente de validação desenvolvido no Matlab. Este ambiente permite codificar e enviar na representação IEEE-754 os operandos de entrada, assim como receber e decodificar o resultado das operações.

- A quantificação do erro foi feita a partir das medidas do erro quadrático médio (MSE) e erro absoluto médio (MAE) usando 36 amostras para cada tamanho de palavra, sendo o Matlab usado como estimador estatístico (operação de 64 bits).

- Um gerador de código VHDL foi desenvolvido no Matlab. Esta ferramenta, chamada *vFPUgen*, permite selecionar o tamanho de palavra na representação IEEE-754 assim como os parâmetros dos algoritmos NR e CORDIC, fornecendo a descrição de *hardware* dos operadores aritméticos e trigonométricos estudados neste trabalho. Desta forma é reduzido o tempo de desenvolvimento, teste e validação das arquiteturas propostas, além de viabilizar o desenvolvimento de futuras implementações de *hardware*.

A tabela 3.4 apresenta os valores do erro quadrático médio (MSE) e erro absoluto médio (MAE) obtidos após o processo de validação dos operadores aritméticos implementados.

Tabela 3.4: Erro quadrático médio e erro absoluto médio dos operadores aritméticos

Operador	Erro	27(8,18)	32(8,23)	64(11,52)
$FPadd$	MSE	1.54E-5	1.26E-7	2.73E-10
$[-100, 100]$	MAE	2.34E-3	8.90E-5	2.92E-6
$FPmul$	MSE	3.34E-4	1.34E-7	8.82E-13
$[-100, 100]$	MAE	3.17E-3	1.32E-4	1.67E-7
$divNR$	MSE	2.36E-9	6.97E-13	4.97E-14
$[-100, 100]$	MAE	1.52E-5	2.68E-7	6.85E-8
$sqrtNR$	MSE	1.81E-10	2.54E-11	8.80E-19
$[0, 100]$	MAE	8.28E-6	1.44E-6	3.00E-10

Como esperado, o melhor comportamento é encontrado para um formato de 64 bits. O MSE das unidades *FPadd* e *FPmul* é maior do que as unidades de divisão e raiz quadrada. Isto pode ser explicado devido aos refinamentos sucessivos realizados pelo

algoritmo NR. Entre todos os operadores aritméticos, a unidade *FPmul* apresenta o maior valor de MSE devido a erros incorporados no processo de truncamento. No entanto, este problema pode ser facilmente resolvido acrescentando um estado de arredondamento no resultado da multiplicação das mantissas [143].

A tabela 3.5 apresenta os resultados de erro quadrático médio (MSE) e de erro absoluto médio (MAE) para as unidades *Cordicsincos*, *Cordicatan* e *CordicTaylorexp* em função do tamanho de palavra.

Tabela 3.5: Erro quadrático médio e erro absoluto médio dos operadores trigonométricos

Operador	Erro	27(8,18)	32(8,23)	64(11,52)
Cordicsincos(seno)	MSE	3.64E-8	9.10E-10	5.70E-10
$[-100\pi, 100\pi]$	MAE	9.06E-5	2.44E-5	1.84E-5
Cordicsincos(cosseno)	MSE	1.39E-8	5.64E-10	6.00E-10
$[-100\pi, 100\pi]$	MAE	6.82E-5	1.79E-5	1.83E-5
Cordicatan	MSE	4.52E-4	1.21E-9	9.65E-10
$[-100, 100]$	MAE	3.79E-3	2.97E-5	2.68E-5
CordicTaylorexp	MSE	3.07E-10	7.38E-13	6.17E-12
$[0, 1]$	MAE	1.34E-5	6.26E-7	7.55E-7
CordicTaylorexp	MSE	6.30E-5	4.41E-5	4.40E-5
$[1, 5]$	MAE	3.67E-3	1.93E-3	1.92E-3
CordicTaylorexp	MSE	1.09E-1	7.29E-5	1.59E-7
$[5, 10]$	MAE	1.86E-1	4.93E-3	1.97E-4
CordicTaylorexp	MSE	1.75E-10	1.39E-10	1.65E-10
$[-10, 0]$	MAE	7.69E-6	6.40E-6	6.46E-6

Pode-se observar que o valor do erro das unidades *Cordicsincos* e *Cordicatan* varia com o tamanho de palavra. Como esperado, representações com tamanho de palavra menor apresentam erros maiores. No entanto observou-se uma pequena diferença entre as implementações de 32 e 64 bits. Isto pode ser explicado devido ao uso de multiplicadores em ponto flutuante nas *unidades de redução de argumento* e a erros de truncamento nos componentes *FPfrac*, usados durante a etapa de redução do argumento de entrada.

A unidade *CordicTaylorexp* para cálculo da função exponencial apresenta um erro de aproximação pequeno (aproximadamente de 1E-10) para argumentos de entrada pequenos (na faixa $[-1, 1]$) ou negativos. Entretanto, esta unidade apresenta um erro maior para argumentos de entrada positivos. Observe-se que conforme aumenta o argumento de entrada maior é o erro associado. Este problema é evidente no caso da representação de 27 bits, em que o MSE é aproximadamente igual a 1.09E-1 para uma faixa de valores de [5,10], indicando que o tamanho da mantissa não fornece suficiente

precisão para representar números grandes com varias casas decimais. Este problema é resolvido pelo uso da representação de 32 bits (precisão simples), em que o MSE é aproximadamente 7.29E-5 para a mesma faixa de valores.

Embora as arquiteturas CORDIC propostas possuam um erro de aproximação pequeno para o cálculo de funções trigonométricas, três fontes de propagação de erro foram identificadas. A primeira se apresenta por erros de quantização durante o cálculo do argumento reduzido, especificamente pelo uso de truncamentos nas unidades *FPmul* e *FPfrac*. Porém, este problema pode ser resolvido utilizando estados de arredondamento. A segunda fonte de propagação de erro é o uso das técnicas de redução de argumento. As técnicas de redução de argumento apresentadas neste trabalho foram escolhidas pela sua fácil implementação. No entanto, técnicas mais sofisticadas podem ser estudadas visando melhoras na aproximação, como apresentado em [138]. A terceira fonte de erro apresenta-se no cálculo da função *exponencial* para argumentos reduzidos pequenos [139]. Neste trabalho, o impacto da terceira fonte de erro foi minimizado usando uma unidade de comparação e uma expansão em séries de Taylor de segunda ordem para argumentos reduzidos pequenos (na faixa de [-0.05,0.05]), como mostrado na figura 3.8.

A figura 3.9 apresenta uma validação da metodologia híbrida *CordicTaylor* proposta para o cálculo da função *exponencial*. Nesta figura foi calculado o erro quadrático médio (MSE) para 1000 argumentos de entrada aleatórios na faixa [-1,1] usando uma implementação de precisão dupla. Pode-se observar que a arquitetura original *FPCordic*, que usa exclusivamente o algoritmo CORDIC em coordenadas hiperbólicas para o cálculo da função *exponencial*, apresenta um MSE aproximado de $1E^{-2}$ quando o valor do argumento reduzido está na faixa [-0.05, 0.05]. Entretanto, a arquitetura *CordicTaylorexp* proposta, a qual considera a expansão por séries de Taylor, apresenta um MSE máximo de $1E^{-8}$ para valores de argumento reduzido na mesma faixa.

3.7 COMPARAÇÃO DO TEMPO DE EXECUÇÃO

No intuito de realizar uma comparação de desempenho entre as arquiteturas de *hardware* e uma solução baseada em *software*, o tempo de execução em ciclos de relógio foi estimado para ambas as soluções. O processador de *software* MicroBlaze [144] foi embarcado no mesmo dispositivo FPGA e programado para executar os operadores

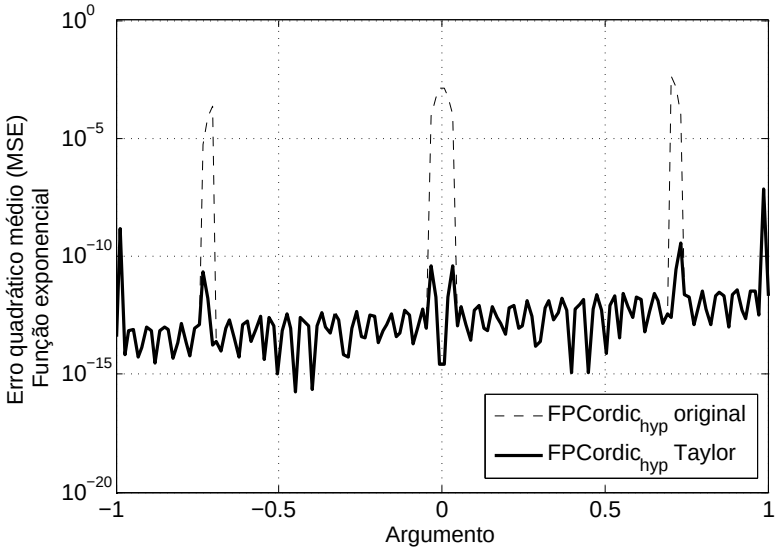


Figura 3.9: Validação da arquitetura *CordicTaylorexp* para argumentos pequenos

aritméticos e trigonométricos implementados em *hardware*. O MicroBlaze é um processador RISC de 32 bits que permite operações aritméticas de ponto flutuante de precisão simples usando o padrão IEEE-754. Desta forma, as soluções de *hardware* e *software* operam à mesma frequência de relógio, permitindo assim uma comparação de desempenho mais eficiente.

O tempo de execução das arquiteturas de *hardware* foi obtido primeiramente em simulação e, posteriormente, o valor foi confirmado no osciloscópio após o mapeamento das arquiteturas no dispositivo FPGA. Por outro lado, para calcular o número de ciclos de relógio da implementação de *software* um *timer* de 32 bits foi instanciado e conectado ao MicroBlaze pelo barramento PLB (*Processor Local Bus*). Um módulo de comunicação serial, também conectado ao PLB, permite enviar os resultados das operações e o valor do *timer*. O MicroBlaze instanciado utiliza uma memória cache de 64 KBytes (32 KBytes de instruções e 32KBytes de dados) baseada em blocos BRAM embarcados, usando um total de 22% dos blocos BRAM disponíveis no dispositivo FPGA. Adicionalmente, segundo os dados de custo em área lógica, após o processo PAR, o processador de *software* utiliza 3% de registradores, 5% de LUTs e 7% de blocos DSP.

A tabela 3.6 apresenta os valores obtidos para os operadores aritméticos implementados usando as condições experimentais mencionadas na seção anterior.

Tabela 3.6: Comparação do tempo de execução dos operadores aritméticos

Operador implementado	Tempo de execução		Fator de aceleração
	hardware	MicroBlaze	
FPadd	2 (20 ns)	17.08 (0.171 μs)	8.55
FPmul	2 (20 ns)	17.08 (0.171 μs)	8.55
divNR	10 (100 ns)	41.58 (0.416 μs)	4.16
sqrtNR	15 (150 ns)	617.28 (6.17 μs)	41.13

As unidades de cálculo aritmético, comparação e conversão de dados em ponto flutuante do MicroBlaze não são arquiteturas *pipeline*, isto é, um único cálculo pode ser feito de cada vez [144]. Segundo o conjunto de instruções do MicroBlaze, as latências das unidades de cálculo aritméticos são respectivamente 2, 4, 28 e 27 ciclos de relógio para as unidades de soma/subtração, multiplicação, divisão e raiz quadrada. Contudo, o uso da representação aritmética de ponto flutuante em processadores de *software*, comumente demanda o uso de funções adicionais associadas à manipulação de números em ponto flutuante que, adicionadas ao tempo de transferência entre a ULA e as memórias de dados e instruções, aumentam o tempo de execução das implementações de *software* em ponto flutuante [145].

A tabela 3.7 apresenta os valores obtidos para os operadores trigonométricos implementados usando as condições experimentais mencionadas na seção anterior.

Tabela 3.7: Comparação do tempo de execução dos operadores trigonométricos

Operador implementado	Tempo de execução		Fator de aceleração
	hardware	MicroBlaze	
Cordicsin	55 (550 ns)	32258.4 (0.323 ms)	587.27
Cordiccos	55 (550 ns)	32405.0 (0.324 ms)	589.10
Cordicatan	47 (470 ns)	31851.1 (0.318 ms)	676.60
CordicTaylorexp	83 (830 ns)	28541.7 (0.285 ms)	343.37

Com referência nas tabelas 3.6 e 3.7 é possível concluir que as arquiteturas propostas permitem uma aceleração considerável no tempo de execução. Observa-se um fator de aceleração de 8.55 vezes para as unidades de soma e multiplicação, 4.16 vezes para a divisão e 41.1 vezes para o cálculo da raiz quadrada. Por outro lado, o cálculo dos operadores trigonométricos apresenta o maior fator de aceleração, sendo o melhor caso observado para o cálculo da função *atan* (676.6 vezes) e o pior caso para o cálculo da função exponencial (343.3 vezes).

3.8 DISCUSÕES FINAIS DO CAPÍTULO E CONTRIBUIÇÕES

Neste capítulo foram apresentadas as implementações de *hardware* dos operadores de cálculo aritmético e trigonométrico usando a representação aritmética de ponto flutuante.

Os dados de custo em área lógica demonstram que os dispositivos FPGA são uma solução factível para a implementação dos operadores estudados. Em geral o consumo de recursos de *hardware* é satisfatório, havendo suficiente área lógica para implementações futuras. Entretanto, o número de blocos DSP dedicados é um parâmetro crítico, especificamente para representações de alta precisão (64 bits). Entre os operadores aritméticos, a unidade de soma/subtração requer de mais recursos de *hardware*, embora não faça uso de blocos DSP embarcados.

Os resultados de síntese permitiram concluir que a representação de 27 bits (1 bit de sinal, 8 bits de mantissa e 18 bits de mantissa) permite realizar implementações mais eficiente em termos de consumo de recursos se comparado com a representação de precisão simples (32 bits). Especificamente, foi observado que a implementação de 27 bits usa 50% menos blocos DSPs do que a implementação de 32 bits. Isto é devido a que os blocos DSPs usados nos dispositivos FPGAs da Xilinx são baseados em multiplicadores de 18x18 nas famílias Spartan e de 18x25 nas famílias Virtex e Kintex da Xilinx. Desta forma, para a implementação de uma operação de multiplicação usando blocos DSPs de 18x25 bits, a representação de 27 bits não realiza produtos parciais em contraste com a implementação de 32 bits em que são necessários dois produtos parciais para multiplicar as mantissas de 23x23 bits.

Adicionalmente, a implementação de 27 bits apresenta uma faixa dinâmica similar à implementação de 32 bits (expoentes do mesmo tamanho), porém com um detrimento no erro associado. Este último fato é derivado do decremento de 5 bits no tamanho da mantissa e afeta, principalmente, os algoritmos que fazem uso de multiplicações intermediárias. Contudo, para as implementações almejadas no contexto deste trabalho, a economia de recursos de *hardware*, especificamente de blocos DSPs, é o fator que mais limita a escalabilidade dos algoritmos de inteligência de enxames, particularmente do número de partículas paralelas implementadas. Portanto, a representação aritmética de 27 bits foi escolhida para mapear os algoritmos paralelos por inteligência de enxames.

Uma das principais contribuições deste trabalho está relacionada com a implementação *hardware* de bibliotecas parametrizáveis dos operadores em ponto flutuante. Diversos trabalhos encontrados na literatura têm realizado propostas para a implementação em FPGAs de operadores aritméticos e trigonométricos em aritmética de ponto fixo e de ponto flutuante. Enquanto ao desenvolvimento de bibliotecas parametrizáveis em FPGAs usando aritmética de ponto flutuante podem-se destacar as propostas de Wang [134] e Govindu [146] para operadores aritméticos. Por outro lado, poucos trabalhos são reportados na literatura sobre a implementação de bibliotecas parametrizáveis de funções trigonométricas usando ponto flutuante. Contudo, nos trabalhos prévios não se considera a precisão como critério de projeto das arquiteturas de *hardware*. Neste trabalho a precisão foi tratada como uma variável de projeto na implementação das arquiteturas propostas. Isto permitiu realizar uma análise de *tradeoff* entre o erro associado e outros três parâmetros de projeto de circuitos: custo em área lógica, tempo de execução e consumo de potência. Detalhes sobre esta análise de *tradeoff* para os operadores desenvolvidos podem ser encontrados em [130] e [131].

Uma segunda contribuição do trabalho está relacionada com a implementação da arquitetura *CordicTaylorexp* para o cálculo da função exponencial (vide figura 3.8), a qual é produto da abordagem de desenvolvimento explicada anteriormente. Esta arquitetura foi desenvolvida inicialmente implementando o algoritmo CORDIC em coordenadas hiperbólicas. No entanto, durante a fase de análise do erro associado foi possível constatar que o algoritmo apresenta erros de aproximação quando o argumento de entrada é pequeno. Visando contornar este problema, a arquitetura proposta utiliza os mesmos recursos de *hardware* para implementar a expansão por séries de Taylor de segunda ordem da função exponencial. Desta forma, é possível multiplexar entre as duas lógicas implementadas após a comparação do valor do argumento de entrada. Uma nova fase de análise do erro associado mostrou a eficiência do resultado apresentado pela arquitetura híbrida, vide figura 3.9. Embora possam ser usadas algumas técnicas para melhorar a precisão do algoritmo CORDIC para o cálculo da função exponencial, é importante ressaltar que a abordagem *CordicTaylor*, apresentada neste trabalho, mostrou-se eficiente em termos de desempenho, erro associado e tempo de execução.

Uma contribuição final dos tópicos abordados neste capítulo é a ferramenta *vFPUgen*. Esta ferramenta permite a geração automática de código VHDL dos operadores aritméticos e trigonométricos usando ponto flutuante, de forma que o tamanho de palavra e os parâmetros dos algoritmos são configurados *offline* segundo os requerimentos de precisão, custo em área lógica e tempo de execução de uma aplicação específica. É

importante destacar que esta ferramenta facilita o uso de FPGAs em variadas aplicações em engenharia e permite diminuir o tempo de desenvolvimento.

Como trabalhos futuros sobre a implementação dos operadores aritméticos espera-se um melhoramento no erro de aproximação da unidade de multiplicação mediante a substituição dos processos de truncamento por estados de arredondamento. Por outro lado, os operadores desenvolvidos poderão ser usados como co-processadores de *hardware* e integrados no MicroBlaze, sendo acessados por meio de instruções customizadas, permitindo acelerar a execução de algoritmos baseados em *software*. A estimação do consumo de potência dos operadores implementados é uma tarefa importante para a complementação dos resultados obtidos, visando futuras aplicações em sistemas embarcados. Finalmente, uma implementação em Java da ferramenta *vFPUgen* e a construção de uma interface de usuário permitirá a disseminação da ferramenta entre a comunidade científica além de viabilizar a independência de plataforma.

Capítulo 4 IMPLEMENTAÇÕES E RESULTADOS DOS ALGORITMOS DE OTIMIZAÇÃO POR INTELIGÊNCIA DE ENXAMES

Este capítulo apresenta as arquiteturas paralelas propostas para os algoritmos de otimização por inteligência de enxames estudados neste trabalho. As arquiteturas propostas foram mapeadas em um dispositivo FPGA da família Virtex5 da Xilinx e o custo em área lógica foi estimado usando os resultados de síntese. As implementações foram validadas por meio de experimentos computacionais usando funções de teste *benchmark*, permitindo realizar comparações numéricas do desempenho dos algoritmos implementados em *hardware* com a sua respectiva realização em *software*. Por fim, uma comparação do tempo de execução entre as soluções de *hardware* e *software* é apresentada. O capítulo finaliza com uma discussão dos resultados obtidos.

4.1 CONSIDERAÇÕES INICIAIS

Os algoritmos de inteligência de enxames descritos a seguir fazem uso dos operadores aritméticos e trigonométricos de aritmética de ponto flutuante apresentados no capítulo anterior. Esta escolha é justificada dado que problemas de otimização geralmente requerem de uma representação numérica de alta precisão com uma alta faixa dinâmica.

As primeiras implementações de *hardware* das arquiteturas paralelas dos algoritmos PSO, SFLA e ABC foram baseadas em uma representação numérica de 32 bits. Detalhes destas implementações e seus respectivos resultados podem ser encontradas nas publicações realizadas no contexto deste trabalho [147], [148], [149], [150], [151], [152]. Os resultados de síntese obtidos para uma representação de 32 bits apontaram um alto consumo de recursos de *hardware*, especificamente de blocos DSP embarcados. Contudo, como demonstrado no capítulo anterior, dado que dispositivos FPGAs usam blocos DSPs de 9x9 bits ou 18x18 bits, uma representação de ponto flutuante de 27 bits (8 bits de expoente e 18 bits de mantissa) permite realizar implementações de *hardware* eficientes em termos de consumo de recursos mantendo a precisão e faixa dinâmica dos

operadores aritméticos.

Visando acelerar o tempo de desenvolvimento, uma ferramenta de *software* para geração automática de código VHDL dos algoritmos de inteligência de enxames foi desenvolvida no Matlab. Esta ferramenta, chamada de *Swarm Generator*, permite selecionar o número de partículas, dimensionalidade do problema de otimização, número de iterações, entre outros parâmetros de cada algoritmo. Adicionalmente, esta ferramenta permite escolher o tamanho de palavra da representação em ponto flutuante segundo os requerimentos de custo em área lógica, precisão e faixa dinâmica. As implementações apresentadas neste trabalho usam uma representação numérica de 27 bits.

Conforme explicado na Seção 2.7, três métodos de adição de diversidade artificial foram estudados e implementados em *hardware*. Inicialmente estes métodos foram aplicados para o algoritmo PSO. Em Muñoz *et al.* [148] apresenta-se uma descrição detalhada da implementação em *hardware* da metodologia atrativa-repulsiva. Em Muñoz *et al.* [151] é descrita a implementação *hardware* do método de congregação passiva seletiva, a qual usa a segunda melhor partícula para evitar o problema de convergência prematura. Em Muñoz et al. [153] é apresentada uma implementação da metodologia de aprendizado em oposição a qual foi aplicada no treinamento de uma rede neural artificial.

No Apêndice B deste trabalho apresenta-se um quadro comparativo dos resultados obtidos de uma implementação *software* das três técnicas de diversidade artificial aplicadas no algoritmo PSO para problemas *becnhmark*. Entretanto, devido à facilidade de implementação e baixo consumo de recursos de *hardware*, a técnica de aprendizado em oposição (OBL) foi escolhida para ser aplicada em todos os algoritmos de inteligência de enxames estudados neste trabalho. A abordagem OBL usa o conceito de número oposto (vide equação 2.18), possibilitando que as partículas sejam *teletransportadas* em regiões opostas no espaço de busca. Observe-se que em espaços de busca simétricos a implementação da abordagem OBL pode ser facilmente aplicada mediante uma operação de negação.

4.2 UNIDADE DE GERAÇÃO DE NÚMEROS ALEATÓRIOS

Como descrito no Capítulo 2, os algoritmos estudados no presente trabalho utilizam geradores de números aleatórios (GNA) com distribuição uniforme. Assim, foi necessário implementar em *hardware* unidades de geração de números aleatórios, de forma que

os circuitos implementados possam simular o comportamento estocástico dos algoritmos de otimização. Diversas técnicas têm sido propostas para a implementação de GNAs em ponto flutuante, entre elas as mais usadas são os métodos baseados na conversão de fixo para flutuante de registros binários de deslocamento com realimentação linear (LFSR: *Linear Feedback Shift Register*) e as técnicas de separação do expoente usando variáveis aleatórias com distribuição geométrica [154]. Outra técnica amplamente usada para gerar um número aleatório em ponto flutuante com distribuição uniforme na faixa $[A, B)$ é calcular $U(B - A) + A$, sendo U um número em ponto flutuante uniformemente distribuído na faixa $[0, 1)$.

Neste trabalho as unidades GNA foram descritas utilizando LFSRs devido à facilidade de implementação em *hardware*, requerendo poucas portas lógicas. O LFSR é uma técnica síncrona onde um registrador é deslocado produzindo um novo estado e o bit de entrada é determinado por meio de uma operação linear do estado prévio. Geralmente utiliza-se uma função *ou exclusiva* (*xor*), facilmente implementada em *hardware*. No LFSR alguns bits, chamados de *taps*, são utilizados na função de realimentação obtendo o novo estado do registrador. Neste trabalho as unidades GNA foram implementadas utilizando LFSRs de 20bits, como mostra a figura 4.1, de forma que selecionando os *taps* apropriados, a sequência de bits pseudoaleatórios se repete a cada 2^{20} estados [155].

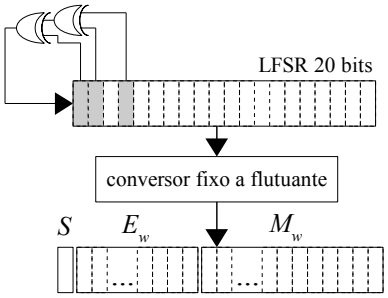


Figura 4.1: Unidade de geração de números aleatórios em ponto flutuante

Dado que o LFSR opera em ponto fixo, um conversor de ponto fixo para flutuante é utilizado no intuito de representar os números em aritmética de ponto flutuante. Tal metodologia adiciona uma perda de eficiência do gerador devido a que números muito pequenos são representados com menor resolução após a conversão de ponto fixo para flutuante. Isto pode ser explicado levando em consideração que a *função de massa de probabilidade* (FMP) de um número em ponto flutuante uniformemente distribuído incrementa a sua resolução para números próximos do zero, porém, após

a conversão de fixo para flutuante, a FMP deste número conserva os valores com a mesma probabilidade e resolução da representação em ponto fixo [154]. Contudo, nas aplicações de otimização embarcada os números aleatórios com distribuição uniforme são usados para criar o movimento aleatório de partículas e, portanto, a perda de uniformidade, que é percebida para números pequenos (inferiores a 2^{-3}), não representa um problema significativo. É importante ressaltar que o restante dos números na faixa de valores $[0, 1]$ continuarão a ser amostrados a partir de uma distribuição bastante próxima à distribuição uniforme.

4.3 IMPLEMENTAÇÃO *HARDWARE* DO NÚMERO OPOSTO

Conforme explicado nas considerações iniciais, a técnica de aprendizado em oposição (OBL) foi escolhida para ser aplicada em todos os algoritmos estudados neste trabalho. A abordagem OBL usa o conceito de número oposto (vide equação 2.18), possibilitando que as partículas sejam *teletransportadas* em regiões opostas no espaço de busca.

A figura 4.2 apresenta a implementação *hardware* do cálculo do número oposto. Esta arquitetura utiliza uma unidade GNA para sortear se o operador *not* é aplicado no bit mais significativo da posição atual $x(t)$, calculando assim o número oposto; caso contrário se mantem o valor da posição atual. Adicionalmente, alguns bits da mantissa do número aleatório gerado são transferidos para a mantissa da nova posição, modificando levemente a posição da partícula. Observe-se que esta arquitetura está limitada ao uso de espaços de busca simétricos com relação ao zero visando simplificar a implementação *hardware*.

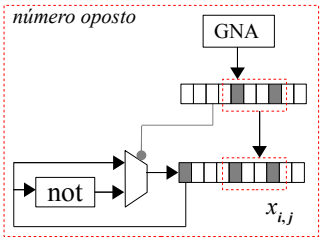


Figura 4.2: Arquitetura de *hardware* do cálculo do número oposto para espaços de busca simétricos

4.4 IMPLEMENTAÇÃO *HARDWARE* DO ALGORITMO PSO

Segundo as equações 2.2 e 2.3, o processo de atualização da posição de uma partícula em uma dimensão requer as seguintes operações: (a) cinco operações de soma/subtração, (b) cinco multiplicações e (c) a geração de dois números aleatórios com distribuição uniforme. A figura 4.3 apresenta a arquitetura proposta para a implementação de uma partícula, a qual atualiza a posição de uma partícula em uma dimensão. A arquitetura utiliza uma unidade de multiplicação (*FPmul*), uma unidade de soma/subtração (*FPadd*) e uma unidade GNA.

A partícula recebe como entradas o peso de inércia w, a velocidade atual $\boldsymbol{v}$, a posição atual $\boldsymbol{x}$, a melhor posição individual $\boldsymbol{y}_i$ e a melhor posição global $\boldsymbol{y}_s$. O processo de atualização é executado em cinco estados e todas as operações em cada estado são executadas simultaneamente. As saídas, velocidade atual ($\boldsymbol{v}(t+1)$) e posição atual ($\boldsymbol{x}(t+1)$) são armazenadas em registradores. Observe-se que duas multiplicações foram economizadas modificando as unidades GNA para gerar números aleatórios com distribuição uniforme em uma faixa aproximada de $[0, c_1]$ e $[0, c_2]$. Uma *máquina de estados finitos* (FSM) permite sincronizar e compartilhar os componentes de *hardware*.

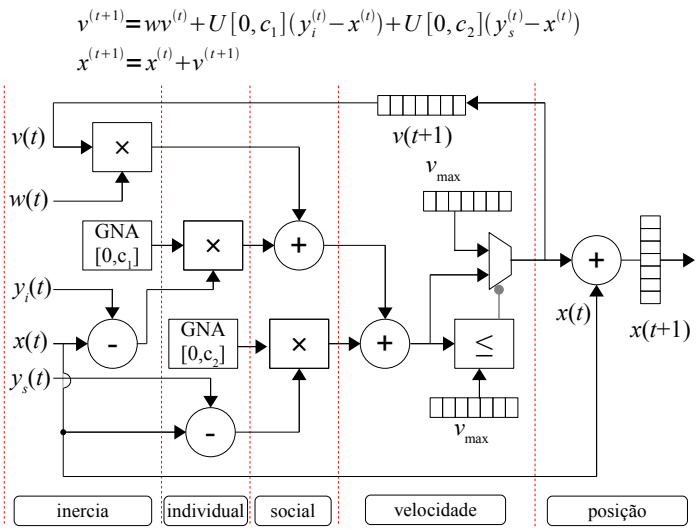


Figura 4.3: Arquitetura de uma partícula no algoritmo PSO

A arquitetura de *hardware* do algoritmo PSO está baseada em uma abordagem síncrona, isto é, cada partícula possui uma implementação em *hardware* da função objetivo para avaliar os resultados das novas posições após o processo de atualização das posições.

Esta arquitetura, chamada de HPPSO (*Hardware Parallel Particle Swarm Optimization*), é apresentada na figura 4.4. A mesma consiste de um enxame de S partículas atualizando as suas posições simultaneamente para otimizar um *cluster* de S funções objetivo. Cada função objetivo representa o problema de otimização N-dimensional. Dado que a atualização das S partículas e a avaliação das S funções objetivo são realizadas em instantes diferentes, os mesmos recursos de *hardware* são compartilhados para ambos os processos.

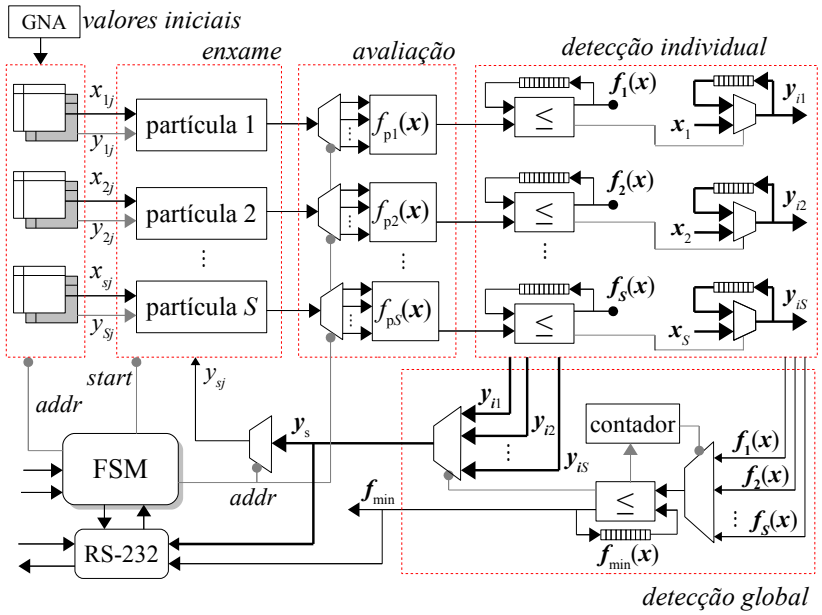


Figura 4.4: Arquitetura de *hardware* do algoritmo PSO (HPPSO)

A arquitetura HPPSO consiste em cinco componentes principais: (a) *unidade de enxame*, (b) *unidade de avaliação*, (c) *unidade de detecção individual*, (d) *unidade de detecção global* e (e) *unidade de controle* (FSM). A *unidade de enxame* contém S partículas em paralelo, sendo cada partícula implementada como descrito na figura 4.3. A *unidade de avaliação* contém S funções objetivo em paralelo. Note-se que cada função objetivo requer a atualização da posição da partícula em todas as dimensões. Portanto, as S partículas da *unidade de enxame* são reutilizadas N vezes, atualizando assim a posição das partículas em todas as dimensões. A *unidade de detecção individual* compara, de forma simultânea, os valores atuais de aptidão de cada partícula com seu respectivo melhor valor de aptidão previamente encontrado. Se o valor de aptidão atual da partícula i ($f(\boldsymbol{x}_i)$) for menor que o seu melhor valor de aptidão ($f(\boldsymbol{y}_i)$), então a melhor posição individual ($\boldsymbol{y}_i$) é substituída pela posição atual da partícula i ($\boldsymbol{x}_i$). A *unidade de detecção global* calcula o valor mínimo entre os S valores de aptidão (lembrar que todos os problemas de otimização considerados neste trabalho são de minimização).

A execução do algoritmo começa com a inicialização aleatória das posições das partículas em cada dimensão. A melhor posição individual de cada partícula ($\boldsymbol{y}_i$) e a melhor posição global ($\boldsymbol{y}_s$) são inicializadas no valor limite do espaço de busca. Cada um dos componentes de *hardware* possui um sinal de *clock*, *reset*, *start* e *ready*. Estes dois últimos sinais são controlados pela FSM visando sincronizar os componentes em função do estado atual.

Uma das principais características da arquitetura HPPSO é a sua estrutura regular. Este fato permite que o projetista faça modificações *offline* apenas em uma instancia da arquitetura. Por exemplo, uma nova função objetivo pode ser implementada modificando apenas a unidade de avaliação, enquanto as outras unidades permanecem inalteradas.

Adicionalmente, é importante salientar que nesta arquitetura não foram usados blocos de memória RAM para armazenar a posição atual e a melhor posição individual de cada partícula. Implementações de *hardware* do algoritmo PSO usando blocos RAM internos foram realizadas e estudadas previamente em Muñoz *et al.* [149]. No entanto, nas arquiteturas apresentadas neste trabalho foram usados flip-flops para o armazenamento da informação do enxame, generalizando assim a descrição de *hardware* das arquiteturas e viabilizando a portabilidade para outras famílias de FPGAs da Xilinx ou para outras tecnologias como Altera e Microsemi.

4.5 IMPLEMENTAÇÃO *HARDWARE* DO ALGORITMO O-PSO

O Algoritmo 5 apresenta o pseudocódigo do método de aprendizado em oposição (OBL) aplicado no algoritmo PSO. Este algoritmo é similar ao algoritmo PSO canônico (vide Algoritmo 1 na Seção 2.3), porém, após a detecção global, um contador é incrementado quando o valor de aptidão mínimo não melhora o valor da iteração anterior. Após um número máximo de tentativas ($maxFNC$) sem melhora no valor de aptidão, a posição das partículas será atualizada usando o conceito de número oposto (vide equação 2.18).

A arquitetura proposta para o cálculo da posição oposta (ou antipartícula) é apresentada na figura 4.5. Esta arquitetura recebe um bit de entrada *opp* que indica se a nova posição da partícula é atualizada usando a equação canônica do PSO ou se o número oposto deve ser calculado. No ultimo caso, a FSM controla uma unidade GNA para

Algoritmo 5: Pseudocódigo do algoritmo O-PSO

Entrada: S, N, c_1, c_2, x_{max}, v_{max}, Max_{iter}, $MaxFNC$, $threshold$
Saída: posição da melhor partícula: $\boldsymbol{x}$ e o seu melhor valor de aptidão: $f(\boldsymbol{x})$
início
  para $k = 1 : S$ **faça**
    para $j = 1 : N$ **faça**
      $v_{kj} = -v_{max} + 2U[0,1]v_{max}$; //Inicializa enxame
      $x_{kj} = -x_{max} + 2U[0,1]x_{max}$
    fim para
  fim para
  repita
    para $k = 1 : S$ **faça**
      se $f(\boldsymbol{x}_k) \leq f(\boldsymbol{y}_{ik})$ **então**
        $\boldsymbol{y}_{ik}$=$\boldsymbol{x}_k$; //Avaliação e detecção individual
        $fmin_k = f(\mathbf{x}_k)$
      fim se
      se $fmin_k \leq fmin$ **então**
        $\boldsymbol{y}_s$=$\boldsymbol{y}_{ik}$; //Avaliação e detecção social
        $fmin = fmin_k$
      fim se
    fim para
    se $fmin \leq lastfmin$ **então** $lastfmin = fmin_k$ $FNC = 0$; //verifica melhora
    senão $FNC++$;
    para $k = 1 : S$ **faça**
      para $j = 1 : N$ **faça**
        se $FNC = MaxFNC$ **então**
          $FNC = 0$ **se** $LFSR(j) =' 1'$ **então** $x_{kj} = -x_{kj} + U[-1,1]/2$; //aplica OBL
        senão
          $v_{kj} = v_{kj} + U_1[0,c_1](y_{ikj} - x_{kj}) + U_2[0,c_2](y_{sj} - x_{kj})$; //movimenta partículas
          $x_{kj} = x_{kj} + v_{kj}$
        fim se
      fim para
    fim para
  até $f(\boldsymbol{y}_s) \leq threshold$;
fim

sortear se é aplicado o operador *not* no bit mais significativo da posição atual $x(t)$, calculando assim o número oposto; caso contrário se mantem o valor da posição atual.

A arquitetura de *hardware* do algoritmo PSO com aprendizado em oposição é mostrada na figura 4.6. Esta arquitetura, chamada de HPOPSO, está baseada na implementação da partícula oposta descrita anteriormente. Observe-se que após a unidade de detecção global o contador FNC é incrementado quando o valor de aptidão atual (f_{min}) não melhora o valor na iteração anterior ($lastf_{min}$). A cada iteração é avaliado se se o contador FNC é igual ao número máximo de tentativas sem melhoria no valor de aptidão ($maxFNC$). Caso afirmativo o sinal *opp* recebe o valor de '1', indicando que todas as partículas do enxame irão atualizar as suas posições usando o número oposto.

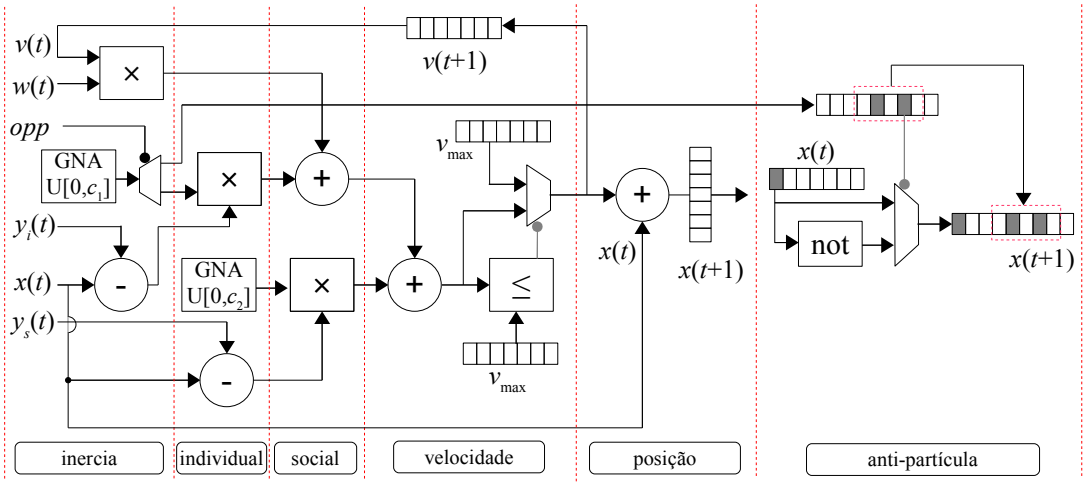


Figura 4.5: Arquitetura da partícula com aprendizado em oposição (antipartícula)

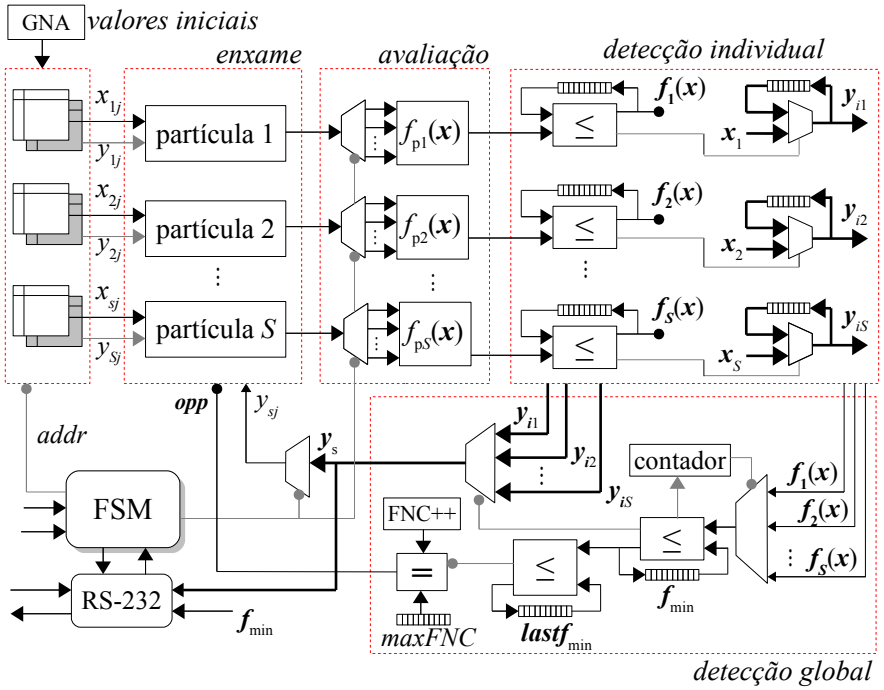


Figura 4.6: Arquitetura *hardware* do algoritmo O-PSO (HPOPSO)

4.6 IMPLEMENTAÇÃO *HARDWARE* DO ALGORITMO ABC

O Algoritmo 2 na Seção 2.4 apresenta o pseudocódigo o algoritmo ABC. Observe-se que o processo de atualização da posição das abelhas operarias e/ou seguidoras requer

do uso de uma unidade GNA na faixa [-1,1], vide equação 2.7. Para isto, a mesma arquitetura da unidade GNA descrita na figura 4.1 foi utilizada, porém o bit mais significativo da sequência pseudoaleatória é direcionado ao bit mais significativo do número aleatório em ponto flutuante, modificando aleatoriamente o bit de sinal para gerar números positivos e negativos.

No algoritmo ABC original, vide Algoritmo 2, as abelhas operarias com fontes de alimento com baixo desempenho ($p(i) \approx 1.0$) são ignoradas usando a comparação $rand_i > p(i)$, de forma que as seguidoras ociosas escolham mais de uma vez as melhores fontes de alimento (valores de probabilidade baixa). Entretanto, na arquitetura HPABC proposta, explicada a continuação, as abelhas com baixo desempenho são utilizadas para seguir exclusivamente a melhor fonte de alimento. Esta modificação no algoritmo ABC original foi realizada visando simplificar a implementação *hardware*. O Algoritmo 6 mostra o pseudocódigo do algoritmo *GlobalBeeABC* (GBABC) proposto para a implementação de *hardware*.

Algoritmo 6: Pseudocódigo do algoritmo *GlobalBeeABC* (GBABC)

Entrada: S, N, $[x_{min}, x_{max}]$, $maxTrial$, $maxIter$
Saída: posição da melhor fonte de alimento: $\boldsymbol{x}_g$ e o seu melhor valor de aptidão: $f(\boldsymbol{x}_g)$
início
  Gerar as posições aleatórias para as S fontes de alimento e determinar aptidões $f(\boldsymbol{x})$
  repita
    // Enviar operarias para explorar as fontes de alimento:
    Para cada solução i determinar um vizinho k e dimensão j
    Criar uma nova solução usando a equação 2.7
    Calcular os valores de aptidão $f(\boldsymbol{x})$
    Atualizar posições se $f(\boldsymbol{x}_i)$ melhora o valor anterior
    Calcular o vetor de probabilidades p_i usando equação 2.6
    // Enviar seguidoras para explorar as fontes de alimento segundo p_i:
    para $i = 1 : S$ **faça**
      se $rand() > p_i$ **então**
        Determinar um vizinho k e dimensão j
        Criar uma nova solução $\boldsymbol{x}_i$ usando a equação 2.7
      senão
        Vizinho é determinado pela melhor solução global $\boldsymbol{x}_g$
        Determinar dimensão j
        Criar uma nova solução $\boldsymbol{x}_i$ usando a equação 2.7
      fim se
      Calcular o valor de aptidão $f(\boldsymbol{x}_i)$
      Atualizar a posição se $f(\boldsymbol{x}_i)$ melhora o valor anterior
    fim para
    // Enviar escoteiras para buscar novas fontes de alimento
    Determine as soluções abandonadas e envie as abelhas escoteiras para buscar novas fontes de alimento
    Atualize a melhor solução $\boldsymbol{x}_g$ segundo as aptidões
    $iter = iter + 1$
  até $iter \leq maxIter$;
fim

A figura 4.7 mostra a implementação de *hardware* da equação 2.7 para a atualização

da posição de uma abelha operária e/ou seguidora em uma dimensão. Esta arquitetura está baseada em uma FSM de quatro estados. O primeiro estado calcula a diferença entre os indivíduos i e k na dimensão j. No segundo estado o resultado da diferença é multiplicado pelo fator aleatório ϕ_{ij} (calculado externamente) e no terceiro estado o resultado da multiplicação é adicionado à posição atual do indivíduo i. Finalmente, a nova posição é comparada com os limites máximo e mínimo permitidos. A mesma arquitetura pode ser utilizada para o movimento das abelhas operárias e as seguidoras.

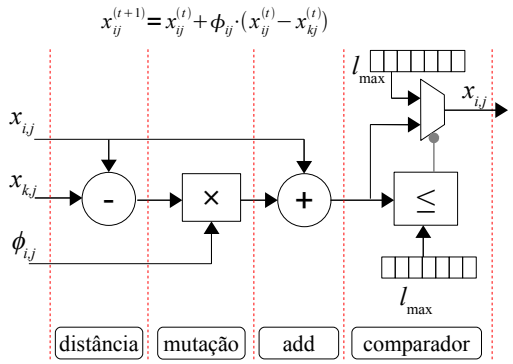


Figura 4.7: Arquitetura de movimento de uma abelha operária ou seguidora

A figura 4.8 ilustra a implementação em *hardware* da arquitetura paralela do algoritmo ABC, chamada de HPABC. A mesma está constituída de estados sequenciais sincronizados por uma FSM. Cada estado está representado pelos quadros pontilhados. Os processos dentro de cada estado são executados em paralelo e o inicio de um processo (sinal de *start*) é ativado pelo sinal de *ready* do processo anterior (similar a um arranjo sistólico linear unidirecional de S rotas), o que simplifica a descrição da FSM.

No primeiro estado inicializa-se a posição de todas as fontes de alimento em todas as dimensões usando para isto uma unidade GNA cujo estado inicial é determinado por um registrador de entrada (semente inicial). A posição de cada fonte de alimento é armazenada em uma memória.

No segundo estado S unidades GNA em paralelo são utilizadas para calcular os valores aleatórios ϕ_{ij}. Os bits 7 até 4 e 11 até 8 de cada GNA são utilizados para selecionar o vizinho k e o parâmetro j, respectivamente, os quais permitem escolher, por meio de multiplexadores, os valores $x_{i,j}$ e $x_{k,j}$. No terceiro estado são executados em paralelo os S movimentos das abelhas operárias no intuito de atualizar a posição da fonte de alimento na dimensão j. Para isto é utilizada a arquitetura de movimento de uma

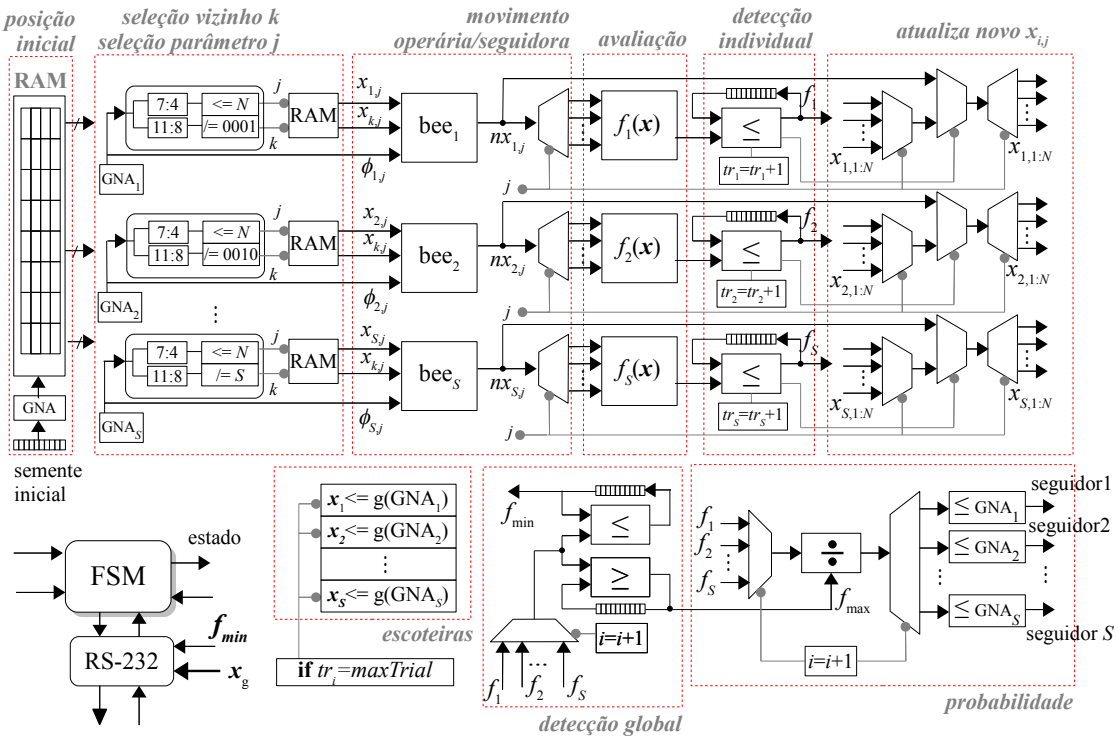


Figura 4.8: Arquitetura *hardware* do algoritmo ABC (HPABC)

abelha (figura 4.7).

No quarto estado, as fontes de alimento são avaliadas em paralelo usando as respectivas funções objetivo. No quinto estado, realiza-se a detecção individual (caso de minimização). Se o valor de aptidão melhora o valor anterior então se atualiza a nova posição, caso contrário incrementa-se o contador tr_i e a posição da fonte de alimento mantém o valor original. No próximo estado, realiza-se a detecção global, identificando a fonte de alimento com melhor aptidão e a fonte de alimento com pior desempenho. Simultaneamente, caso um dos contadores tr_i alcance o valor máximo permitido $maxTrial$ então os valores da unidade GNA_i (valores entre [-1,1]) são utilizados para criar novas posições no espaço de busca $[x_{min}, x_{max}]$. Este processo corresponde ao envio das abelhas escoteiras.

A fase das abelhas seguidoras começa com a implementação da equação para o cálculo da probabilidade da escolha de uma abelha operária por parte de uma abelha seguidora (equação 2.6). Este cálculo é realizado de forma sequencial usando uma unidade de divisão $FPdivNR$ (vide figura 3.4), um contador e um multiplexador. Posteriormente cada valor de probabilidade é comparado em paralelo com os respectivos valores das unidades GNA. Caso o valor GNA seja maior que o valor da probabilidade então a

abelha seguidora escolhe a operária, caso contrário, a fonte de alimento da operária é abandonada e a seguidora escolhe a melhor fonte de alimento global. No próximo passo, o algoritmo retorna ao segundo estado começando o movimento das abelhas seguidoras.

4.7 IMPLEMENTAÇÃO *HARDWARE* DO ALGORITMO O-ABC

O Algoritmo 7 mostra o pseudocódigo do algoritmo *GlobalBeeOABC* (GBOABC) proposto para a implementação da arquitetura HPOABC. Observe-se que durante o processo de envio das abelhas escoteiras as fontes de alimento com baixo desempenho não são abandonadas, senão que o número oposto da posição destas fontes de alimento é calculado sorteando algumas dimensões.

A figura 4.9 ilustra a implementação de *hardware* da técnica de aprendizado em oposição aplicada no algoritmo ABC. Esta arquitetura, chamada de HPOABC, é similar à arquitetura HPABC, porém a implementação das abelhas escoteiras é baseada no cálculo no número oposto e não na criação de uma nova fonte de alimento em posição aleatória.

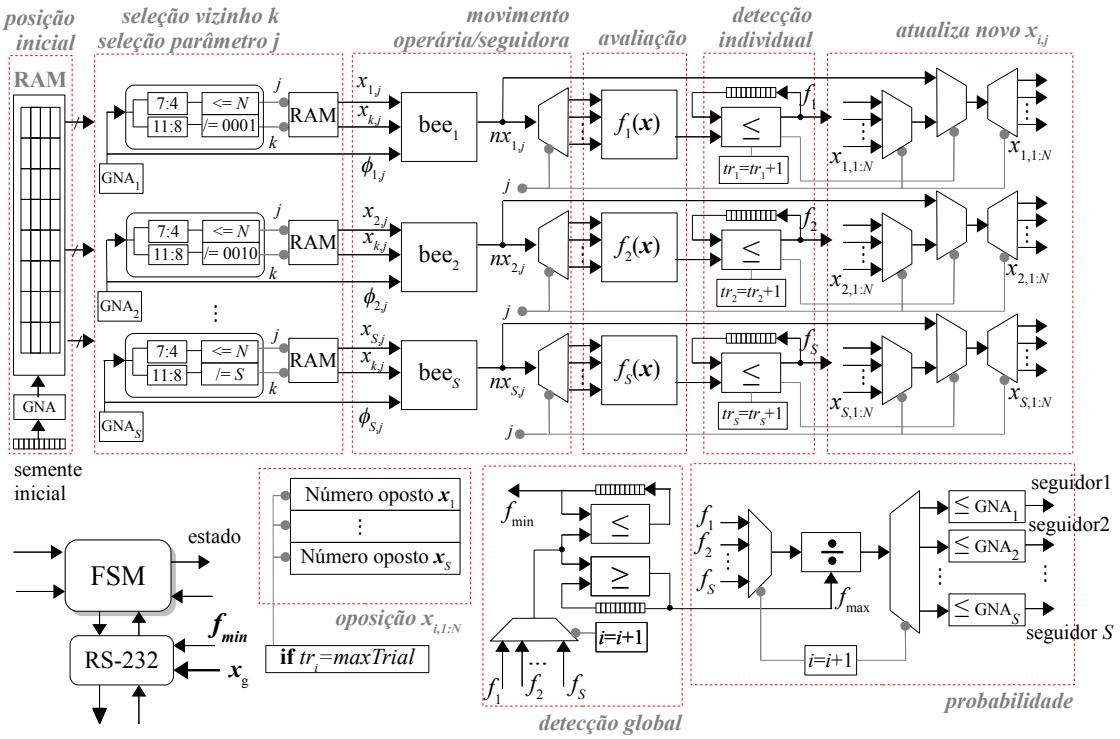


Figura 4.9: Arquitetura *hardware* do algoritmo GBOABC (HPOABC)

Algoritmo 7: Pseudocódigo do algoritmo *GlobalBeeOABC* (GBOABC)

Entrada: S, N, $[x_{min}, x_{max}]$, $maxTrial$, $maxIter$
Saída: posição da melhor fonte de alimento: $\boldsymbol{x}_g$ e o seu melhor valor de aptidão: $f(\boldsymbol{x}_g)$
início
  Gerar as posições aleatórias para as S fontes de alimento e determinar aptidões $f(\boldsymbol{x})$
  repita
    // Enviar operárias para buscar as fontes de alimento:
    Para cada solução i determinar um vizinho k e dimensão j
    Criar uma nova solução usando a equação 2.7
    Calcular os valores de aptidão $f(\boldsymbol{x})$
    Atualizar posições se $f(\boldsymbol{x}_i)$ melhora o valor anterior
    Calcular o vetor de probabilidades p_i usando equação 2.6
    // Enviar seguidoras para explorar as fontes de alimento segundo p_i:
    para $i = 1 : S$ **faça**
      se $rand() > p_i$ **então**
        Determinar um vizinho k e dimensão j
        Criar uma nova solução $\boldsymbol{x}_i$ usando a equação 2.7
      senão
        Vizinho é determinado pela melhor solução global $\boldsymbol{x}_g$
        Determinar dimensão j
        Criar uma nova solução $\boldsymbol{x}_i$ usando a equação 2.7
      fim se
      Calcular o valor de aptidão $f(\boldsymbol{x}_i)$
      Atualizar a posição se $f(\boldsymbol{x}_i)$ melhora o valor anterior
    fim para
    // Enviar escoteiras para buscar novas fontes de alimento
    Aplique a técnica OBL nas soluções de baixo desempenho, sorteando as dimensões:
    para $i = 1 : S$ **faça**
      se $tr_i == maxTrial$ **então**
        $tr_i = 0$
        para $j = 1 : N$ **faça**
          se $LFSR(j) = 1$ **então** $x_{ij} = -x_{ij} + U[-1, 1]/2$
        fim para
      fim se
    fim para
    $iter = iter + 1$
  até $iter \leq maxIter$;
fim

4.8 IMPLEMENTAÇÃO *HARDWARE* DO ALGORITMO FA

Como explicado na equação 2.9, o processo de atualização da posição de um vaga-lume requer do uso de uma unidade GNA com distribuição uniforme na faixa [-0.5,0.5]. Para isto, a mesma arquitetura da unidade GNA descrita na figura 4.1 foi utilizada, porém o bit mais significativo da sequência pseudoaleatória é direcionado ao bit mais significativo do número aleatório em ponto flutuante, permitindo gerar números positivos e negativos.

A figura 4.10 mostra a arquitetura em *hardware* da equação 2.9 de atualização da posição do vaga-lume i sendo atraído pelo vaga-lume vizinho k. Para o controle da

arquitetura é utilizada uma FSM que sincroniza os seguintes operadores: uma unidade GNA, uma unidade $FPadd$ e uma unidade $FPmul$. O cálculo da nova posição em uma dimensão é realizado em cinco estados.

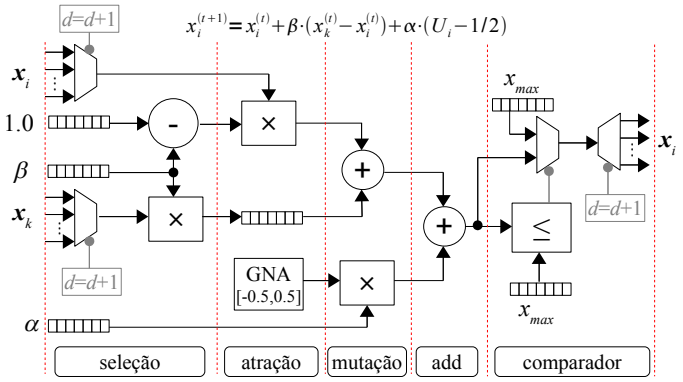


Figura 4.10: Arquitetura de um vaga-lume

4.8.1 Arquitetura do cálculo da atração

Conforme explicado na Seção 2.5 o algoritmo FA baseia-se na atração luminosa entre indivíduos, sendo que a atração pode ser calculada com uma função monótona decrescente. Na arquitetura de *hardware* do algoritmo FA utilizou-se uma função monótona decrescente de segundo grau ($m = 2$ na equação 2.8), visando aumentar a região de atração entre os vaga-lumes. Adicionalmente, a escolha de $m = 2$ também obedece à economia de recursos de *hardware* e à redução do tempo de execução, pois desta forma é evitado o cálculo da raiz quadrada durante o processo de determinação da distância entre vaga-lumes.

A figura 4.11 mostra a arquitetura utilizada para o cálculo da atração. É importante destacar que os mesmos recursos de *hardware* para o cálculo da posição dos vaga-lumes são utilizados para o cálculo da atração. Isto é possível dado o caráter sequencial destes processos. A função exponencial foi implementada usando uma unidade $CordicTaylorexp$ (vide figura 3.8) de 16 iterações.

Visando simplificar a implementação *hardware* em uma abordagem paralela do algoritmo FA, algumas modificações foram realizadas. A principal diferença entre a arquitetura de *hardware* proposta para o algoritmo FA e o algoritmo FA original, proposto por Yang [74], é no processo de atualização das posições dos vaga-lumes. No algoritmo

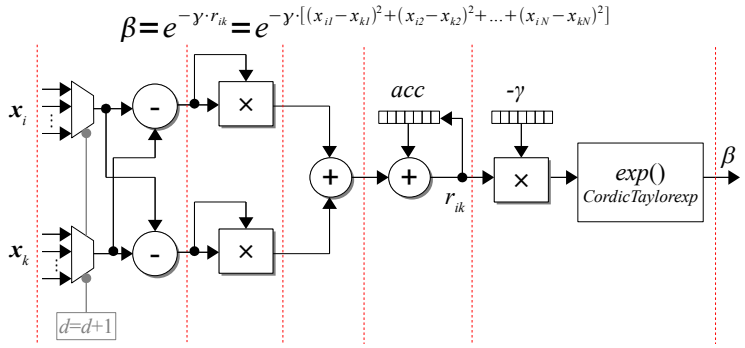


Figura 4.11: Arquitetura do cálculo de atração

original, vide Algoritmo 3, Seção 2.5, os vaga-lumes são atraídos pelos vaga-lumes com maior intensidade de lampejo. Entretanto, na arquitetura proposta neste trabalho todos os vaga-lumes são atraídos exclusivamente pelo vaga-lume com melhor desempenho global (menor valor de aptidão). O pseudocódigo do algoritmo *GlobalFirefly* (GFA) proposto é apresentado no Algoritmo 8. Neste algoritmo a posição do melhor vaga-lume $\boldsymbol{x}_g$ e seu valor de aptidão I_{best} são usados como referência no movimento dos outros vaga-lumes.

Algoritmo 8: Pseudocódigo para o algoritmo *Global Firefly* (GFA)

Entrada: S, N, $[x_{min}, x_{max}]$, $\alpha^{(1)}$, δ, $maxIter$
Saída: posição do melhor vaga-lume: $\boldsymbol{x}_g$ e o seu melhor valor de aptidão: $f(\boldsymbol{x}_g)$
início
  Gerar as posições aleatórias $\boldsymbol{x}_i$ para os S vaga-lumes
  Calcular as intensidades I_i avaliando a função custo $f(\boldsymbol{x}_i)$
  Determine a melhor solução $\boldsymbol{x}_g$ e intensidade I_{best}
  Determinar o coeficiente de absorção γ
  repita
    para $i = 1 : S$ **faça**
      se $I_{best} > I_i$ **então**
        Calcula a distância r_{ik} e a atração β usando equação 2.8
        Cria uma nova solução $\boldsymbol{x}_i$ usando a equação 2.9
        Avalia a nova solução $f(\boldsymbol{x}_i)$ e atualiza intensidade I_i
      fim se
    fim para
    Determine a melhor solução $\boldsymbol{x}_g$ e intensidade I_{best}
    Calcula novo valor de α usando equação 2.10
    $iter = iter + 1$
  até $iter \leq maxIter$;
fim

A arquitetura geral da implementação em *hardware* do algoritmo FA é mostrada na figura 4.12. Esta arquitetura, chamada de HPFA, é baseada em uma FSM que percorre cinco estados (espera, inicialização, aptidão, ordenamento e movimento). No primeiro estado o algoritmo HPFA espera o sinal de *start*. No segundo estado as posições

iniciais dos vaga-lumes são inicializadas aleatoriamente. No terceiro estado é avaliada em paralelo a aptidão da posição atual de cada vaga-lume. No quarto estado a melhor solução é identificada de forma que todos os vaga-lumes possam seguir o vaga-lume com o melhor valor de aptidão (f_{min}). No quinto estado as novas posições dos vaga-lumes são calculadas em paralelo. Para isto, é necessário calcular a atração entre vaga-lume i e o melhor vaga-lume no enxame, usando a arquitetura da figura 4.11. Após o cálculo da atração, os vaga-lumes atualizam a sua posição usando a arquitetura de movimento dos vaga-lumes (vide figura 4.10). Finalmente, o algoritmo retorna ao estado três, começando uma nova iteração.

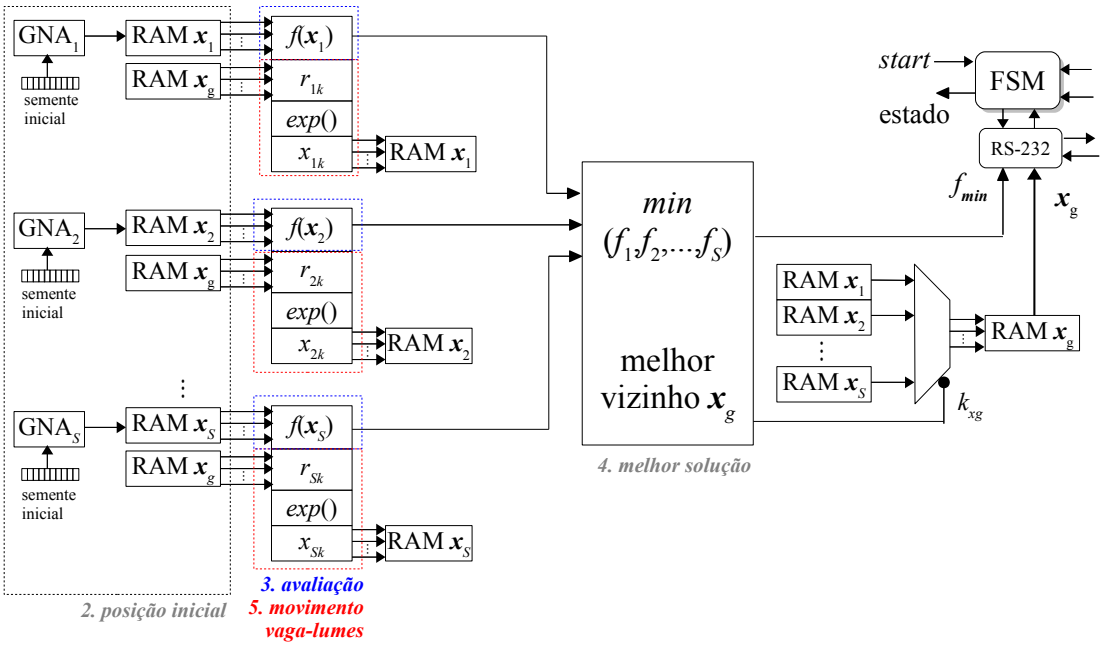


Figura 4.12: Arquitetura *hardware* do algoritmo GFA (HPFA)

4.9 IMPLEMENTAÇÃO *HARDWARE* DO ALGORITMO O-FA

No intuito de aplicar a técnica de aprendizado em oposição na arquitetura HPFA alguns recursos de *hardware* como comparadores e contadores foram acrescentados. O pseudocódigo do algoritmo proposto, chamado de *GlobalOFA* (GOFA), é apresentado no Algoritmo 9. Observe-se que no processo de atualização da posição dos vaga-lumes um contador $trial_i$ é incrementado quando a aptidão I_i na nova posição $\boldsymbol{x}_i$ não melhora o seu valor de aptidão anterior (f_{ind-i}). Após um número pré-estabelecido de tentativas sem melhoria no valor de aptidão, o número oposto da posição $\boldsymbol{x}_i$ é calculado, visando direcionar a busca do vaga-lume i na posição oposta no espaço N-dimensional.

Algoritmo 9: Pseudocódigo para o algoritmo *GlobalOFA* (GOFA)

Entrada: S, N, $[x_{min}, x_{max}]$, $\alpha^{(1)}$, δ, $maxFNC$, $maxIter$
Saída: posição do melhor vaga-lume: $\boldsymbol{x}_g$ e o seu melhor valor de aptidão: $f(\boldsymbol{x}_g)$
início
  Gerar as posições aleatórias $\boldsymbol{x}_i$ para os S vaga-lumes
  Calcular as intensidades I_i avaliando a função custo $f(\boldsymbol{x}_i)$
  Determine a melhor solução $\boldsymbol{x}_g$ e intensidade I_{best}
  Determinar o coeficiente de absorção γ
  repita
    para $i = 1 : S$ **faça**
      se $I_{best} > I_i$ **então**
        se $trial_i = maxFNC$ **então**
          $trial_i = 0$
          para $j = 1 : N$ **faça**
            se $LFSR(j) = 1$ **então** $x_{ij} = -x_{ij} + U[-1, 1]/2$; `//aplica OBL`
          fim para
        senão
          Calcula a distância r_{ik} e a atração β usando equação 2.8
          Cria uma nova solução $\boldsymbol{x}_i$ usando a equação 2.9
        fim se
        Avalia a nova solução $f(\boldsymbol{x}_i)$ e atualiza intensidade I_i
        se $I_i \geq f_{ind-i}$ **então** $trial_i++$
        senão
          $trial_i = 0$
          $f_{ind-i} = I_i$
        fim se
      fim se
    fim para
    Determine a melhor solução $\boldsymbol{x}_g$ e intensidade I_{best}
    Calcula novo valor de α usando equação 2.10
    $iter = iter + 1$
  até $iter \leq maxIter$;
fim

A arquitetura geral da implementação em *hardware* do algoritmo GOFA é mostrada na figura 4.13. Esta arquitetura, chamada de HPOFA, é similar à arquitetura HPFA com a principal diferença de que comparadores e contadores são utilizados após o processo de avaliação das funções objetivo.

4.10 IMPLEMENTAÇÃO *HARDWARE* DO ALGORITMO SFLA

Como explicado na Seção 2.6, o algoritmo SFLA atualiza a posição do pior sapo em cada *memeplex* utilizando as equações 2.11 e 2.12. Pode-se observar que o processo de atualização requer de uma operação de multiplicação, duas unidades de soma/subtração, uma comparação e uma unidade GNA. A implementação *hardware* da arquitetura do sapo foi descrita usando uma FSM, sendo a atualização da posição em cada dimensão realizada em quatro estados como mostrado na figura 4.14.

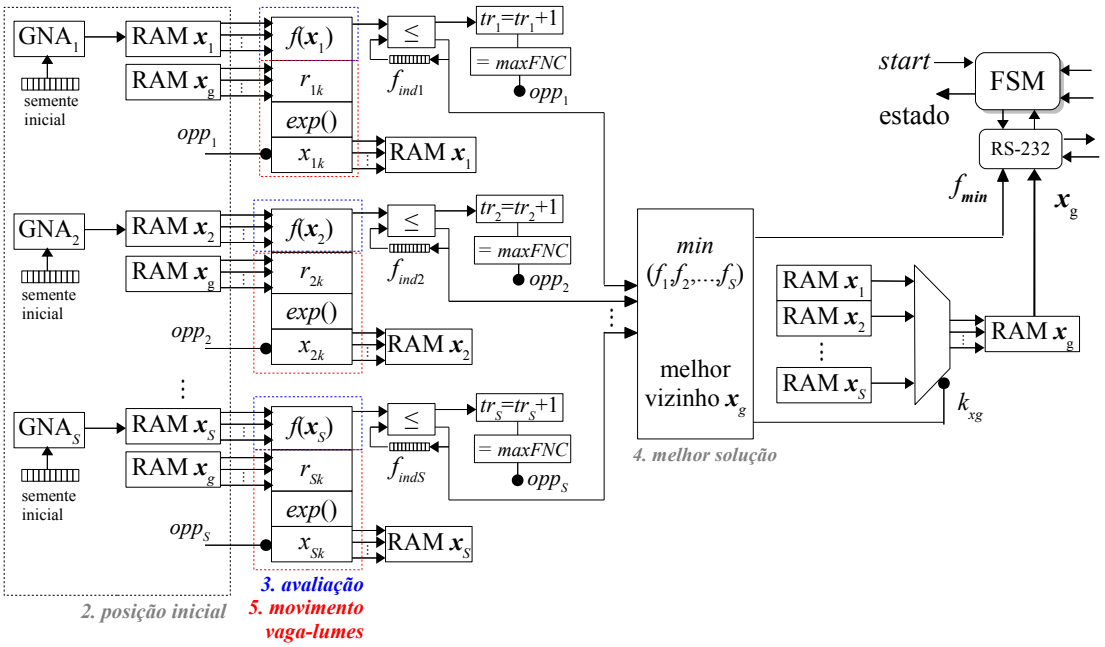


Figura 4.13: Arquitetura geral do algoritmo GOFA (HPOFA)

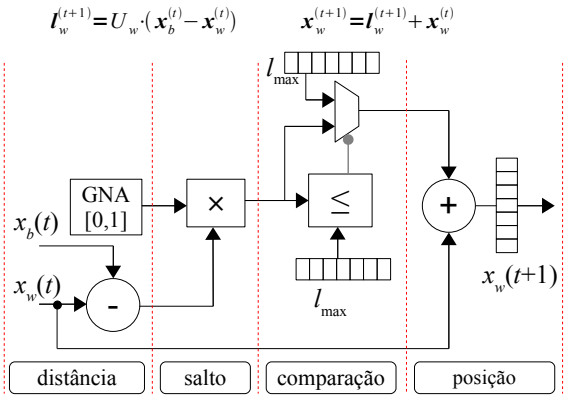


Figura 4.14: Arquitetura de atualização do sapo

As operações em cada estado são executadas simultaneamente. No primeiro estado, a unidade GNA fornece um número aleatório com distribuição uniforme na faixa [0,1]. Simultaneamente, a distância entre a posição atual do melhor sapo e o pior sapo no *memeplex* é calculada usando uma unidade *FPadd*. No segundo estado o salto aleatório para o pior sapo é calculado usando uma unidade *FPmul*. O terceiro estado compara o salto aleatório com o valor máximo permitido (l_{max}) e, finalmente, no quarto estado atualiza-se a posição do pior sapo do *memeplex* reutilizando a unidade *FPadd* para somar a posição atual e o salto aleatório, sendo a nova posição armazenada em um registro.

4.10.1 Arquitetura da unidade de embaralhamento

No algoritmo SFLA o embaralhamento é o processo que permite aos *memes* intercambiar informação (*memotipo* referente à posição dos sapos). No caso específico do algoritmo SFLA são permutadas as posições dos S sapos no pântano em cada dimensão. Em *software* uma possível solução para este processo é a implementação de uma função de permutação aleatória (função *randperm*), a qual requer a execução de S operações de geração de números aleatórios com distribuição uniforme.

No presente trabalho o processo de embaralhamento foi simplificado visando facilitar a implementação de *hardware*. A figura 4.15 ilustra a arquitetura desenvolvida para o processo de embaralhamento dos *memes*, em que S representa o número total de sapos no pântano, M indica o número de *memeplexes*, F representa o número de sapos por *memeplex* e N é a dimensionalidade do problema.

Duas unidades LFSR de geração de números aleatórios são utilizadas durante cada processo de permutação. Estas unidades LFSR fornecem uma sequência binária pseudoaleatória com distribuição uniforme de 6 bits, as quais endereçam o processo de leitura e escrita da memória de armazenamento das posições dos sapos. Os 3 bits menos significativos de cada LFSR representam o índice m do conjunto de *memeplexes* e os 3 bits mais significativos representam o índice f do conjunto de sapos de cada *memeplex*. Uma máquina de estados controla as unidades LFSR, um contador para seleção da dimensão na qual está sendo realizada a permutação, assim como o processo de leitura e escrita.

Mediante a implementação desta arquitetura, os *memeplexes* evoluídos são eficientemente embaralhados. Dois ciclos de relógio são necessários para executar o processo de leitura ou escrita (um ciclo de relógio para a geração do endereçamento aleatório por meio do LFSR e outro ciclo de relógio para o endereçamento da memória). Desta forma, $2(S_F \times N)$ ciclos de relógio são necessários para realizar o processo de embaralhamento, sendo o parâmetro S_F o número de sapos a serem embaralhados em cada dimensão e N o número de dimensões do problema de otimização. Perceba-se que esta arquitetura de embaralhamento limita a arquitetura proposta a utilizar como máximo 8 *memeplexes*, cada um com máximo 8 sapos. Portanto, o enxame está limitado a 8x8 sapos em total, o que é suficiente para aplicações práticas segundo referências dos trabalhos prévios encontrados na literatura [75], [81].

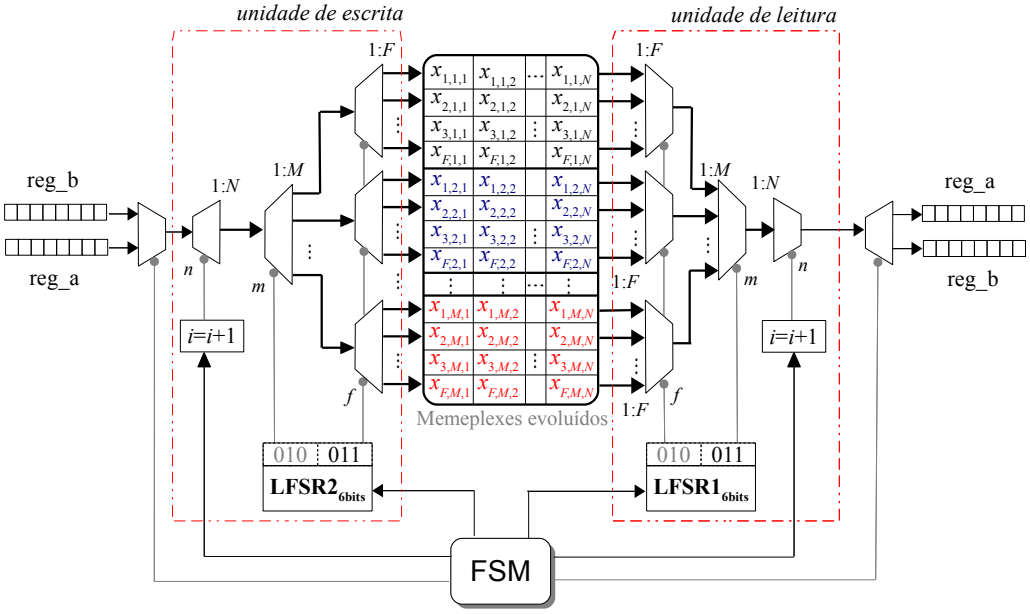


Figura 4.15: Arquitetura da unidade de embaralhamento

A figura 4.16 ilustra a implementação *hardware* do algoritmo SFLA. Nesta arquitetura, chamada de HPSFLA, são implementados M *memeplexes* paralelos, sendo que cada *memeplex* realiza o processo de busca local por meio da evolução de F sapos. Os mesmos recursos de *hardware* da implementação do movimento dos sapos são compartilhados para implementar a função objetivo. Como explicado no Algoritmo 4 (vide Seção 2.6), cada vez que um *memeplex* modifica a posição de um sapo, a função objetivo é avaliada visando verificar o melhoramento da aptidão com relação à posição anterior.

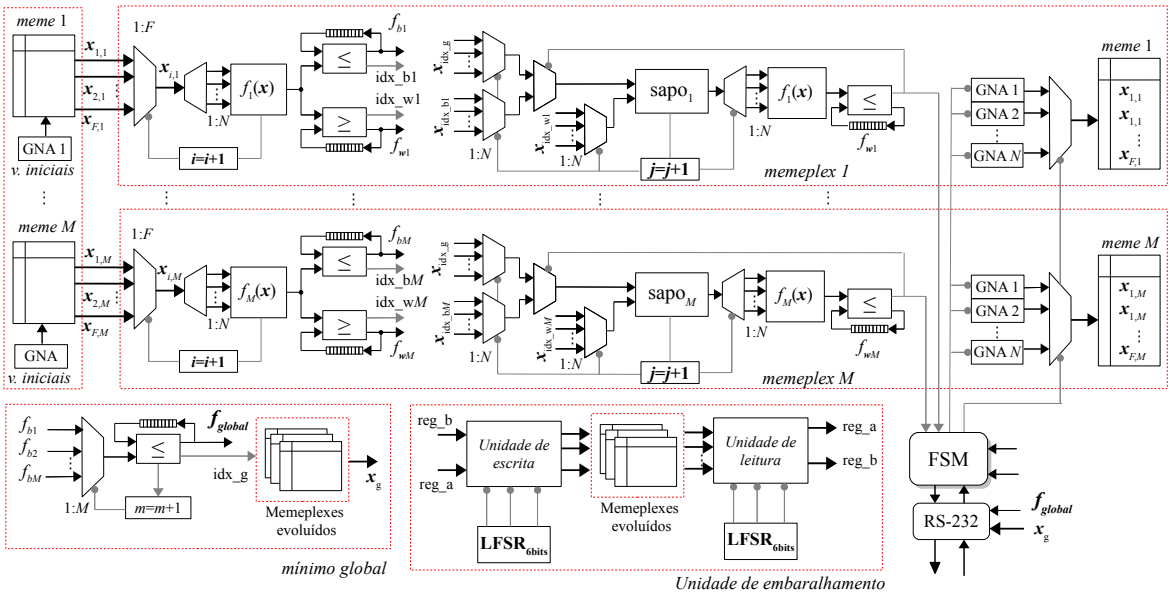


Figura 4.16: Arquitetura de *hardware* do algoritmo SFLA (HPSFLA)

Na figura, os processos contornados por quadros pontilhados são executados em paralelo. Estes processos são controlados por uma FSM. Uma descrição detalhada do funcionamento da arquitetura durante as iterações de busca local é apresentada a seguir.

Na primeira iteração, uma unidade GNA inicializa as posições dos sapos em cada *memeplex*. Logo em seguida, a aptidão da posição de cada sapo em cada *memeplex* é calculada usando as M implementações paralelas da função objetivo. Uma vez que o valor de aptidão tem sido calculado, é feita a comparação com os valores de aptidão anteriores (na primeira iteração são inicializados em infinito). Desta forma, são identificados o melhor e o pior sapo de cada *memeplex* e os respectivos índices são registrados nos sinais idx_b e idx_w, respectivamente.

Após a identificação do melhor sapo de cada *memeplex*, a detecção global é realizada visando calcular o valor de aptidão mínimo ($fmin$) e a sua respectiva solução, isto é, a posição global do melhor sapo ($\boldsymbol{x}_g$). Os registros idx_b e idx_w também são utilizados para endereçar as memórias de armazenamento das posições dos sapos, permitindo que a posição do pior sapo seja melhorada usando a melhor solução local (posição do melhor sapo de cada *memeplex* $\boldsymbol{x}_b$). Posteriormente, a nova posição do pior sapo é avaliada usando a implementação da função custo. Caso a nova posição melhore o valor de aptidão anterior, então uma nova iteração é iniciada. Caso contrário, a posição do pior sapo é atualizada usando desta vez a melhor solução global ($\boldsymbol{x}_g$). Caso a nova solução do pior sapo não melhore o seu valor de aptidão então o pior sapo é eliminado e a máquina de estados controla N unidades GNA paralelas, criando um novo sapo em uma posição aleatória.

Após um número pré-estabelecido de iterações de busca local, a FSM controla a execução da unidade de embaralhamento. A arquitetura HPSFLA proposta permite a aceleração do algoritmo SFLA pela implementação paralela dos *memeplexes* e das funções custo.

4.11 IMPLEMENTAÇÃO *HARDWARE* DO ALGORITMO O-SFLA

A figura 4.17 ilustra a implementação *hardware* do algoritmo O-SFLA. Esta arquitetura, chamada de HPOSFLA, funciona de forma similar à arquitetura de *hardware*

HPSFLA, anteriormente descrita. A principal diferença entre as arquiteturas HPOSFLA e HPSFLA é que a primeira implementa a técnica de aprendizado em oposição (OBL) em vez de eliminar o pior sapo quando o mesmo não melhora sua aptidão.

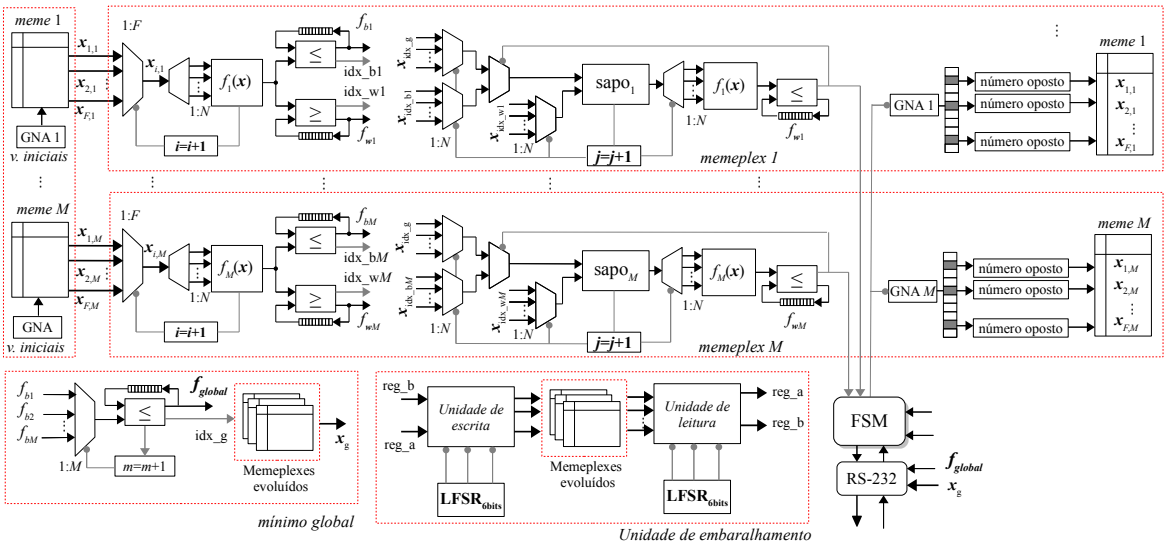


Figura 4.17: Arquitetura de *hardware* do algoritmo O-SFLA (HPOSFLA)

Observe-se que em vez de eliminar o sapo e criar um novo em uma posição aleatória no espaço de busca, a arquitetura HPOSFLA aplica o conceito de número oposto para enviar o sapo na direção oposta da busca atual. O número oposto é calculado usando uma unidade GNA para sortear se é aplicado o operador *not* no bit mais significativo da posição atual $x(t)$ em uma determinada dimensão. Adicionalmente, alguns bits da mantissa do número aleatório gerado são transferidos para a mantissa da nova posição, modificando levemente a posição da partícula. O Algoritmo 10 mostra o pseudocódigo do algoritmo O-SFLA.

4.12 CONJUNTO DE *BENCHMARKS* USADOS PARA VALIDAÇÃO

Neste trabalho, as distintas arquiteturas propostas para os algoritmos de inteligência de enxames foram validadas usando problemas de otimização global considerados *benchmark* pela comunidade científica. Esta metodologia permitiu utilizar funções de teste predefinidas pela comunidade científica que possuem características diferentes em termos de complexidade, decomposição, interação entre as variáveis de decisão, entre outras. A validação com funções *benchmark* permite uma análise do comportamento

Algoritmo 10: Pseudocódigo para o algoritmo O-SFLA

Entrada: S, M, F, N, l_{max}, x_{max}, Max_{LS}, $threshold$
Saída: posição do melhor sapo: $\boldsymbol{x}$ e o seu melhor valor de aptidão: $f(\boldsymbol{x})$
início
  Gerar as posições aleatórias para os S sapos;
  Dividir os S sapos em M *memeplexes*
  repita
    Avaliação das S aptidões $f(\boldsymbol{x})$
    Identificar a melhor posição global $\boldsymbol{x}_g$ e respectiva aptidão $fmin$
    `//O seguinte trecho se aplica para cada memeplex`
    para $i = 1 : Max_{LS}$ **faça**
      Identificar a melhor e a pior posição ($\boldsymbol{x}_b$, $\boldsymbol{x}_w$) de acordo com a aptidão
      Atualizar $\boldsymbol{x}_w^{(t+1)}$ usando as equações 2.11 e 2.12
      se $f(\boldsymbol{x}_w^{(t+1)}) \geq f(\boldsymbol{x}_w^{(t)})$ **então**
        Atualizar $\boldsymbol{x}_w^{(t+1)}$ com 2.11 e 2.12 usando $\boldsymbol{x}_g$ em lugar de $\boldsymbol{x}_b$
        se $f(\boldsymbol{x}_w^{(t+1)}) \geq f(\boldsymbol{x}_w^{(t)})$ **então**
          $\boldsymbol{x}_w^{(t+1)} = -\boldsymbol{x}_w^{(t)} + U[-1, 1]/2$; `//aplica OBL`
        fim se
      fim se
    fim para
    Embaralhar os M *memeplexes* evoluídos
  até $fmin \leq threshold$;
fim

dos algoritmos propostos em diferentes cenários, assim como uma comparação apropriada das soluções obtidas e dos resultados de desempenho.

Duas funções de teste unimodais e duas funções de teste multimodais foram utilizadas no presente trabalho. Segundo Tang *et al.* [156] a avaliação da função objetivo demanda o maior custo computacional. Portanto, explorar a capacidade de paralelização das funções objetivo de um problema de otimização é de vital importância para sua implementação em FPGAs. Uma análise do compromisso entre o custo em área lógica, tempo de execução e número de funções custo a serem executadas em paralelo deve ser realizada de forma prévia à implementação. Desta forma, o desempenho de um algoritmo de otimização pode ser melhorado pela avaliação paralela das funções custo e pela execução de operações simultâneas durante a avaliação de cada uma delas.

4.12.1 Função *Esfera*

Esta é uma função de teste contínua unimodal como indicado pela equação 4.1 O mínimo global é $f(\boldsymbol{x}) = 0$, localizado no ponto $\boldsymbol{x}(i) = 0$, $i = 1, \ldots, N$.

$$f(\vec{x}) = \sum_{i=1}^{N} x_i^2 \tag{4.1}$$

A figura 4.18 apresenta a implementação *hardware* da função *Esfera* para um problema N-dimensional e a respectiva visualização em um caso bidimensional. A implementação foi realizada de forma semiparalela, em que as componentes pares e impares x_{2i} e x_{2i-1} são separadas e calculadas simultaneamente. Dois multiplexadores de entrada em paralelo direcionam os valores de entrada para duas unidades paralelas *FPmul*, calculando os valores x_{2i}^2 e x_{2i-1}^2 que posteriormente são somados pela unidade *FPadd*. A mesma unidade *FPadd* é utilizada para acumular os valores quadráticos atuais com as componentes pares e impares previamente calculadas. Cada vez que o acumulador (*acc*) é utilizado, incrementa-se um contador para direcionar as seguintes componentes e, então, o mesmo procedimento é executado. Quando o contador é igual a $N/2$, o acumulador contém o valor da função objetivo.

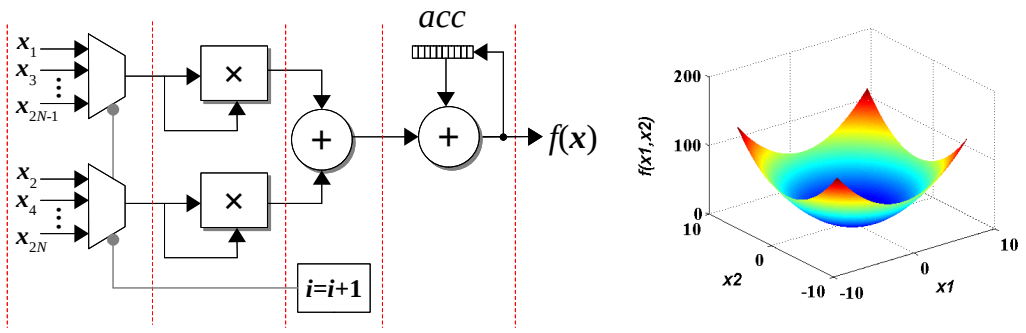


Figura 4.18: Arquitetura da função custo *Esfera*. Exemplo para duas variáveis de decisão

4.12.2 Função *Quadric*

Esta é uma função continua unimodal que produz um hiperboloide rotado, como definido pela equação 4.2. O mínimo global é $f(\boldsymbol{x}) = 0$, localizado no ponto $\boldsymbol{x}(i) = 0$, $i = 1, \ldots, N$.

$$f(\vec{x}) = \sum_{i=1}^{N} \left(\sum_{j=1}^{i} x_j \right)^2 \tag{4.2}$$

A figura 4.19 apresenta a arquitetura utilizada para a implementação da função *Quadric* N-dimensional e sua respectiva visualização em um caso bidimensional. A implementação foi realizada de uma forma sequencial, em que um multiplexador direciona as componentes do vetor de entrada para a unidade *FPadd* que acumula as valores $x_1, x_2, \ldots, x_N$ a cada execução. O resultado do acumulador (*acc1*) é utilizado como argumento de entrada da unidade *FPmul* e um segundo acumulador (*acc2*) é utilizado para calcular o resultado final. Os valores iniciais dos acumuladores são configura-

dos em zero. Uma FSM controla os operadores aritméticos e usa um contador para controlar o multiplexador de entrada.

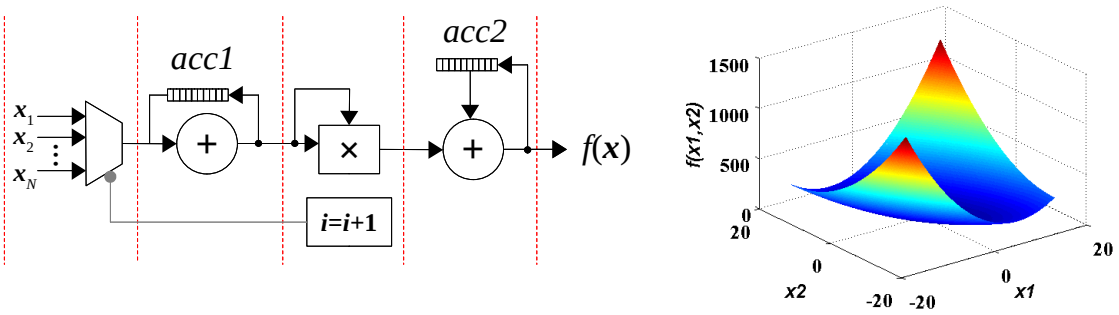


Figura 4.19: Arquitetura da função custo *Quadric*. Exemplo para duas variáveis de decisão

4.12.3 Função *Rosenbrock*

Esta é uma função multimodal para $N > 2$, como indicado na equação 4.3. O mínimo global é $f(\boldsymbol{x}) = 0$, localizado no ponto $\boldsymbol{x}(i) = 1,\ i = 1,\ldots,N$. A figura 4.20 apresenta a implementação em *hardware* da função *Rosenbrock* para um caso N-dimensional. A implementação foi realizada em uma abordagem semiparalela, em que componentes pares e impares subsequentes (x_1 e x_2, x_3 e $x_4,\ldots,x_{N-1}$ e x_N) podem ser agrupadas para calcular de forma paralela uma porção da função. Desta forma, a arquitetura mostrada na figura 4.20 permite calcular as componentes $100(x_2 - x_1^2)^2 + (1 - x_1)^2, 100(x_4 - x_3^2)^2 + (1 - x_3)^2, \ldots, 100(x_N - x_{N-1}^2)^2 + (1 - x_{N-1})^2$. Os mesmos recursos de *hardware* são compartilhados mediante o direcionamento dos argumentos de entrada x_{i-1} e x_i através de multiplexadores de entrada.

$$f(\vec{x}) = \sum_{i=1}^{N/2} 100\left(x_{2i} - x_{2i-1}^2\right)^2 + (1 - x_{2i-1})^2 \tag{4.3}$$

4.12.4 Função *Rastrigin*

Esta é uma função altamente multimodal na qual os mínimos locais estão regularmente distribuídos, como definido na equação 4.4. O mínimo global é $f(\boldsymbol{x}) = 0$, localizado no ponto $\boldsymbol{x}(i) = 0,\ i = 1,\ldots,N$. A figura 4.21 apresenta a arquitetura utilizada na

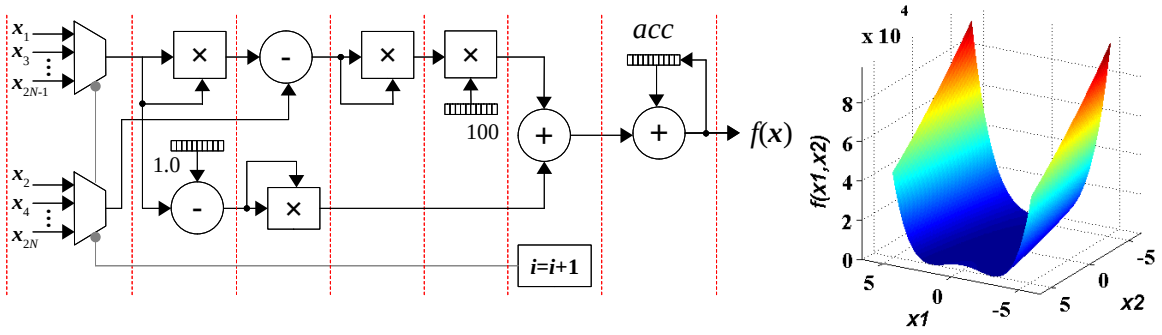


Figura 4.20: Arquitetura da função custo *Rosenbrock*. Exemplo para duas variáveis de decisão

implementação da função *Rastrigin* em um problema N-dimensional. De forma similar à função *Esfera*, esta arquitetura permite separar as componentes pares e impares para serem calculadas em paralelo. A arquitetura consiste de duas unidades paralelas *FPmul*, uma unidade *FPadd*, uma unidade de redução de argumento e duas unidades paralelas *FPTaylor* para calcular as funções *cosseno* [131]. A unidade *FPTaylor* foi configurada para operar com cinco potências não nulas da expansão em séries.

$$f(\vec{x}) = \sum_{i=1}^{N} \left(x_i^2 - 10cos(2\pi x_i) + 10 \right) \tag{4.4}$$

O cálculo da função *Rastrigin* é realizado em nove estados, como mostrado na figura 4.21. Note-se que $\cos(2\pi x_1) = \cos(2\pi dec(x_1))$, em que a função $dec(x_1)$ é a parte decimal da entrada x_1. Desta forma, o primeiro estado calcula simultaneamente a parte decimal das entradas x_{2i-1} e x_{2i}. No segundo estado, duas unidades *FPmul* são utilizadas para calcular os valores $2\pi dec(x_{2i-1})$ e $2\pi dec(x_{2i})$, obtendo um número na faixa $[0, 2\pi]$. Posteriormente, os terceiro e quarto estados efetuam uma redução de argumento no intuito de obter o valor equivalente dos argumentos de entrada da função *cosseno* na faixa $[-\pi/2, \pi/2]$. O quinto estado utiliza as unidades *FPTaylor* para calcular os *cossenos*. Finalmente, uma unidade *FPadd* é utilizada para acumular os resultados parciais. Uma FSM incrementa um contador linear para endereçar os novos argumentos de entrada. O mesmo procedimento se repete para os novos valores de entrada.

4.13 DESENHO DOS EXPERIMENTOS

Visando analisar a escalabilidade das arquiteturas propostas, diferentes experimentos variando o tamanho de enxame e a dimensionalidade dos problemas foram sintetizados, implementados e validados. As tabelas 4.1 e 4.2 mostram os experimentos realizados,

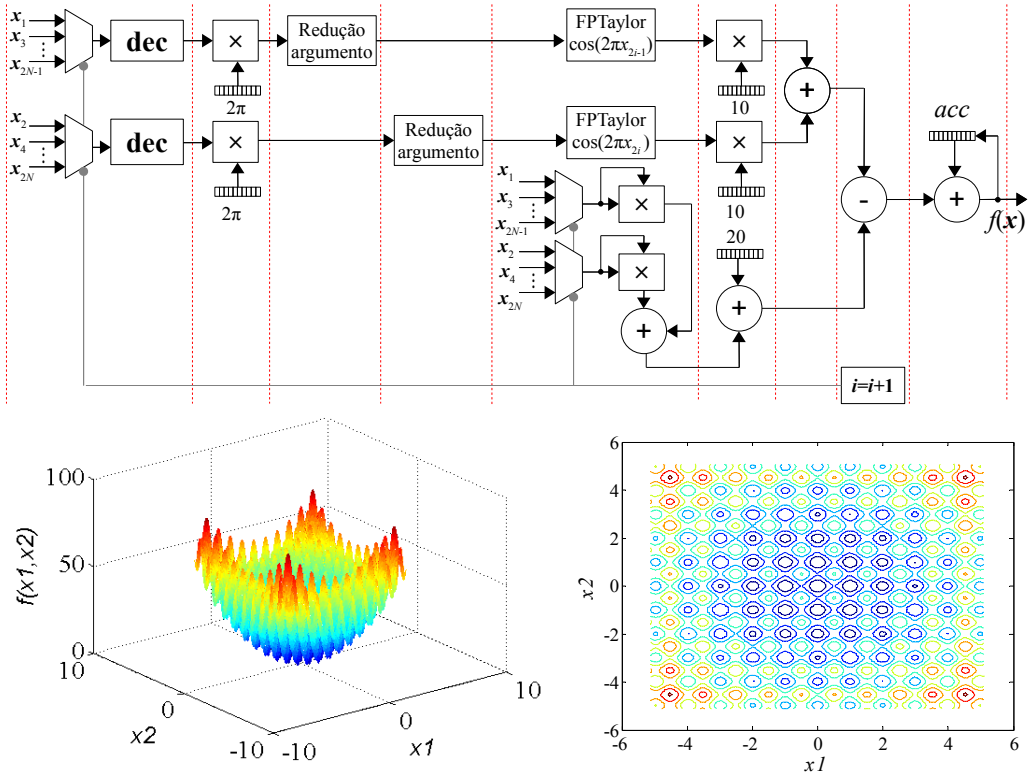


Figura 4.21: Arquitetura da função custo *Rastrigin*. Exemplo para duas variáveis de decisão

em que a parte sombreada corresponde a cinco casos específicos que permitem analisar o comportamento dos algoritmos nas seguintes situações: (a) variação da dimensionalidade com número de partículas paralelas constante, (b) variação do número de partículas com dimensionalidade constante. As partículas são chamadas de fontes de alimento e de vaga-lumes para os algoritmos ABC e FA, respectivamente. No caso do algoritmo SFLA cada *memeplex* é formado por uma colônia de 4 indivíduos, totalizando 16, 24 e 32 sapos.

Tabela 4.1: Experimentos realizados para os algoritmos PSO, ABC e FA

Partículas	Dimensões		
	4	6	10
8			
10			
12			

É importante destacar que os resultados de síntese, convergência dos algoritmos e tempo de execução serão apresentados usando estes cinco casos específicos.

Tabela 4.2: Experimentos realizados para o algoritmo SFLA

Memeplexes	Dimensões		
(4 sapos)	4	6	10
4			
6			
8			

4.14 RESULTADOS DE SÍNTESE

Os algoritmos descritos nas seções anteriores foram implementados para os casos indicados nas tabelas 4.1 e 4.2. As arquiteturas foram implementadas usando a ferramenta de desenvolvimento ISE10.1 para um dispositivo FPGA Virtex5 (dispositivo xc5vlx110T).

Os resultados de síntese apresentam o custo em área lógica em função do consumo de registradores *flip-flops* (FF), *LookUp Tables* (LUTs) e blocos dedicados de processamento digital (DSP48Es). A frequência de operação dos circuitos é mostrada em *Mega-Hertz* (MHz).

O conjunto de tabelas e gráficas a seguir apresentam o custo em área lógica das arquiteturas implementadas para cada função de teste *benchmark*. A tabela 4.3 e a figura 4.22 resumem os dados de síntese para a função *Esfera*. A tabela 4.4 e a figura 4.23 resumem os dados de síntese para a função *Quadric*. A tabela 4.5 e a figura 4.24 resumem os dados de síntese para a função *Rosenbrock*. Por fim, a tabela 4.6 e a figura 4.25 resumem os dados de síntese para a função *Rastrigin*.

De forma geral a arquitetura HPOABC usa menos flip-flops que as outras arquiteturas. Isto pode ser explicado analisando os requerimentos de memória dos algoritmos implementados.

O algoritmo PSO requer uma memória duplicada para armazenar a posição atual e a melhor posição individual das partículas. Entretanto, no algoritmo SFLA cada *memeplex* requer armazenar a posição de um conjunto de indivíduos ou sapos (4 sapos por *memeplex* nas implementações realizadas), provocando um aumento do consumo de registradores para o mesmo número de partículas paralelas. No caso do algoritmo FA os requerimentos de memória são similares aos do algoritmo ABC. Contudo, independentemente da função custo implementada, as arquiteturas HPFA e HPOFA

Tabela 4.3: Resultados de síntese. Função *Esfera* (dispositivo xc5vlx110t)

Número de dimensões	Partículas paralelas	Implementação *hardware*	FF 69120	LUTs 69120	DSP48E 64	Freq. MHz
4	8 partículas	HPOPSO	6955	12626	16	130.576
	8 fontes alimento	HPOABC	5787	16160	17	121.654
	8 vaga-lumes	HPOFA	6720	20957	17	130.100
	6 *memeplexes*	HPOSFLA	7438	16312	13	130.287
6	8 partículas	HPOPSO	8251	14673	16	130.576
	8 fontes alimento	HPOABC	6381	19640	17	121.654
	8 vaga-lumes	HPOFA	7152	22239	17	130.100
	6 *memeplexes*	HPOSFLA	9100	18404	13	130.287
10	8 partículas	HPOPSO	10852	17519	16	130.576
	8 fontes alimento	HPOABC	7578	25514	17	121.654
	8 vaga-lumes	HPOFA	8024	24866	17	130.100
	6 *memeplexes*	HPOSFLA	12449	26737	13	130.287
6	10 partículas	HPOPSO	10213	18549	20	130.576
	10 fontes alimento	HPOABC	7729	26489	21	121.654
	10 vaga-lumes	HPOFA	8876	27956	21	129.914
	4 *memeplexes*	HPOSFLA	6232	12326	9	130.474
6	12 partículas	HPOPSO	12173	21493	24	130.576
	12 fontes alimento	HPOABC	9073	32313	25	108.637
	12 vaga-lumes	HPOFA	10598	33855	25	129.729
	8 *memeplexes*	HPOSFLA	11979	26427	17	130.100

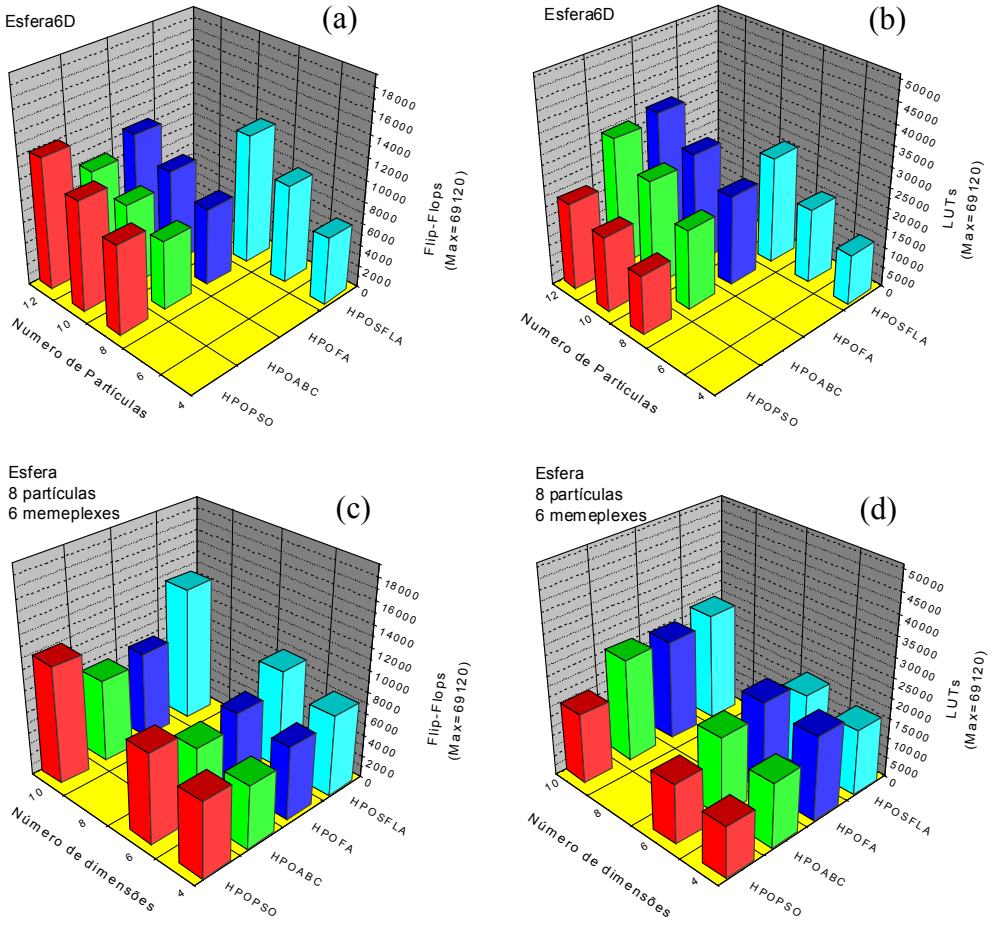


Figura 4.22: Comparação do consumo de recursos para a função *Esfera*

Tabela 4.4: Resultados de síntese. Função *Quadric* (dispositivo xc5vlx110t)

Número de dimensões	Partículas paralelas	Implementação *hardware*	FF 69120	LUTs 69120	DSP48E 64	Freq. MHz
4	8 partículas	HPOPSO	6707	11829	8	142.167
	8 fontes alimento	HPOABC	5547	15120	9	121.654
	8 vaga-lumes	HPOFA	6760	21098	17	130.100
	6 *memeplexes*	HPOSFLA	7258	15559	7	130.287
6	8 partículas	HPOPSO	8003	14194	8	142.167
	8 fontes alimento	HPOABC	6141	18581	9	121.654
	8 vaga-lumes	HPOFA	7192	22673	17	130.100
	6 *memeplexes*	HPOSFLA	8932	17716	7	130.287
10	8 partículas	HPOPSO	10604	17271	8	142.167
	8 fontes alimento	HPOABC	7338	24326	9	121.654
	8 vaga-lumes	HPOFA	8064	25388	17	130.100
	6 *memeplexes*	HPOSFLA	12281	26472	7	130.287
6	10 partículas	HPOPSO	9903	17848	10	142.167
	10 fontes alimento	HPOABC	7429	25175	11	121.654
	10 vaga-lumes	HPOFA	8926	28353	21	130.100
	4 *memeplexes*	HPOSFLA	6112	11981	5	130.474
6	12 partículas	HPOPSO	11801	20680	12	142.167
	12 fontes alimento	HPOABC	8713	31092	13	108.637
	12 vaga-lumes	HPOFA	10658	34395	25	130.100
	8 *memeplexes*	HPOSFLA	11742	26827	9	130.100

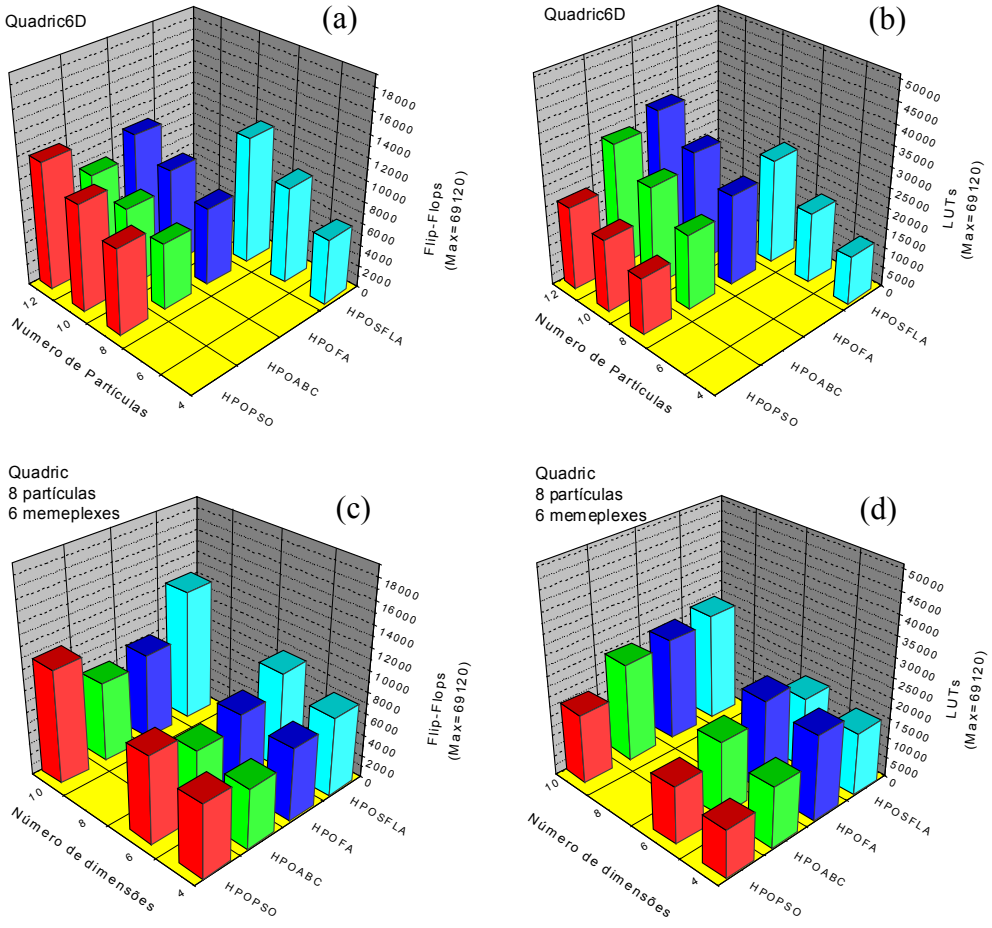


Figura 4.23: Comparação do consumo de recursos para a função *Quadric*

Tabela 4.5: Resultados de síntese. Função *Rosenbrock* (dispositivo xc5vlx110t)

Número de dimensões	Partículas paralelas	Implementação *hardware*	FF 69120	LUTs 69120	DSP48E 64	Freq. MHz
4	8 partículas	HPOPSO	6707	12278	8	142.167
	8 fontes alimento	HPOABC	5555	15678	9	121.654
	8 vaga-lumes	HPOFA	6952	21091	17	130.100
	6 *memeplexes*	HPOSFLA	7251	15855	7	130.389
6	8 partículas	HPOPSO	8003	14187	8	142.167
	8 fontes alimento	HPOABC	6149	19125	9	121.654
	8 vaga-lumes	HPOFA	7065	21334	17	130.100
	6 *memeplexes*	HPOSFLA	8925	17751	7	130.389
10	8 partículas	HPOPSO	10604	16813	8	142.167
	8 fontes alimento	HPOABC	7346	24593	9	121.654
	8 vaga-lumes	HPOFA	8256	25015	17	130.100
	6 *memeplexes*	HPOSFLA	12274	26154	7	130.389
6	10 partículas	HPOPSO	9903	17818	10	142.167
	10 fontes alimento	HPOABC	7439	25907	11	121.654
	10 vaga-lumes	HPOFA	9166	28196	21	129.914
	4 *memeplexes*	HPOSFLA	6104	11962	5	130.576
6	12 partículas	HPOPSO	11801	20601	12	142.167
	12 fontes alimento	HPOABC	8725	32286	13	108.637
	12 vaga-lumes	HPOFA	10946	33956	25	129.729
	8 *memeplexes*	HPOSFLA	11746	26075	9	130.202

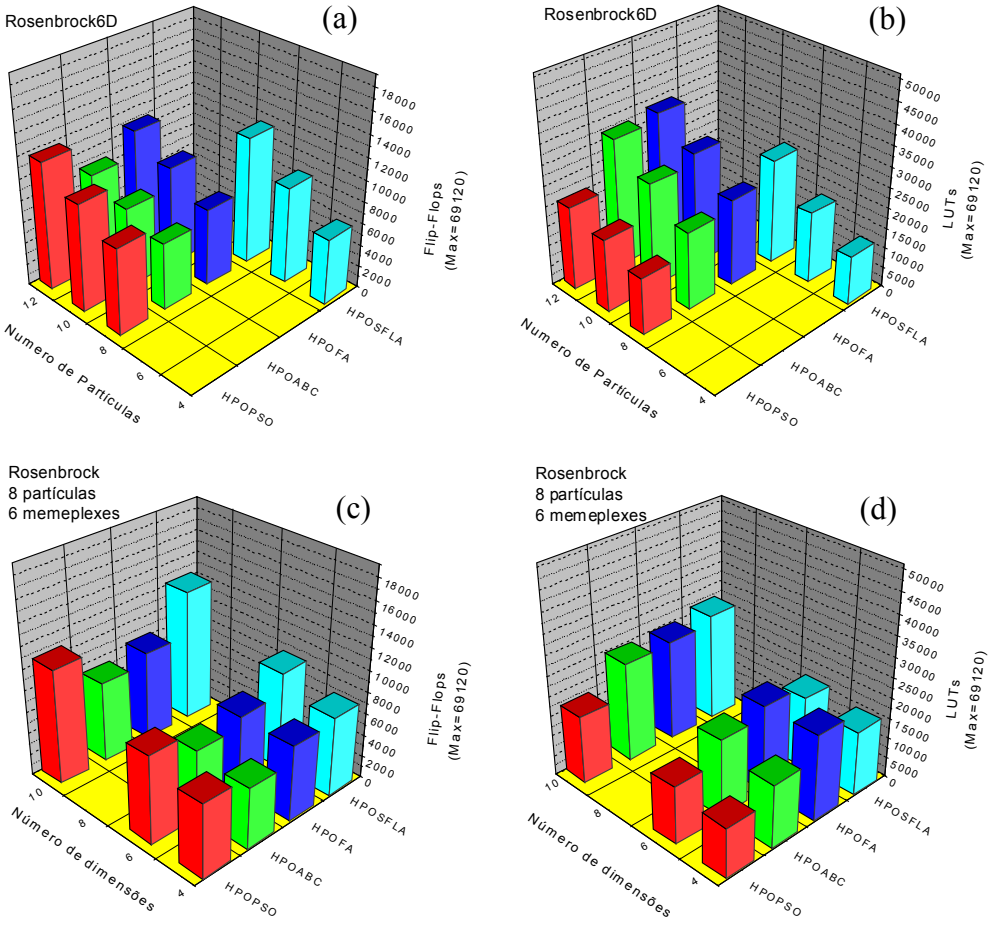


Figura 4.24: Comparação do consumo de recursos para a função *Rosenbrock*

Tabela 4.6: Resultados de síntese. Função *Rastrigin* (dispositivo xc5vlx110t)

Número de dimensões	Partículas paralelas	Implementação *hardware*	FF 69120	LUTs 69120	DSP48E 64	Freq. MHz
4	8 partículas	HPOPSO	11059	24205	16	130.758
	8 fontes alimento	HPOABC	9899	27621	17	121.654
	8 vaga-lumes	HPOFA	9656	28314	17	120.902
	6 *memeplexes*	HPOSFLA	10516	25404	13	130.287
6	8 partículas	HPOPSO	12355	26219	16	129.122
	8 fontes alimento	HPOABC	10493	31014	17	121.654
	8 vaga-lumes	HPOFA	10120	30380	17	126.211
	6 *memeplexes*	HPOSFLA	12203	27339	13	130.287
10	8 partículas	HPOPSO	14978	29171	16	130.382
	8 fontes alimento	HPOABC	11690	36991	17	121.654
	8 vaga-lumes	HPOFA	11016	33121	17	127.857
	6 *memeplexes*	HPOSFLA	15551	35613	13	130.287
6	10 partículas	HPOPSO	15343	33420	20	129.312
	10 fontes alimento	HPOABC	12869	41104	21	121.654
	10 vaga-lumes	HPOFA	12576	37629	21	120.744
	4 *memeplexes*	HPOSFLA	8280	18269	9	130.474
6	12 partículas	HPOPSO	18329	39339	24	129.128
	12 fontes alimento	HPOABC	15242	50272	25	106.826
	12 vaga-lumes	HPOFA	15062	45306	25	125.688
	8 *memeplexes*	HPOSFLA	16099	38364	17	129.222

propostas requerem duas unidades $FPadd$ e duas unidades $FPmul$ para o movimento dos vaga-lumes, especificamente durante o cálculo da atração, incrementando o uso de registradores em comparação com as arquiteturas HPABC e HPOABC, que requerem uma unidade $FPadd$ e uma unidade $FPmul$ para o movimento das abelhas. Isto pode ser confirmado observando que o consumo de flip-flops para a função custo *Rastrigin*, a qual requer duas unidades $FPadd$ e duas unidades $FPmul$, é similar para as arquiteturas HPOABC e HPOFA. Já no caso das funções custo $Esfera$, *Quadric* e *Rosenbrock* o consumo de flip-flops é consideravelmente maior para a arquitetura HPOFA.

Por outro lado, o consumo de lógica combinacional (LUTs) é menor para a arquitetura HPOPSO do que para as outras arquiteturas. Isto pode ser explicado considerando o tipo de operações necessárias para o funcionamento do algoritmo.

O algoritmo ABC requer de realizar divisões para o cálculo da probabilidade de uma abelha operária ser escolhida por uma abelha seguidora. No caso do algoritmo FA, o processo de atração entre vaga-lumes requer do cálculo da função exponencial. Neste ponto é importante salientar que o algoritmo ABC requer da geração de mais números aleatórios se comparado com os algoritmos PSO, FA ou SFLA. Observe-se que para o caso das funções custo $Esfera$, *Quadric* e *Rosenbrock* a arquitetura HPOFA requer mais LUTs do que a arquitetura HPOABC, pois a arquitetura HPOFA usa mais

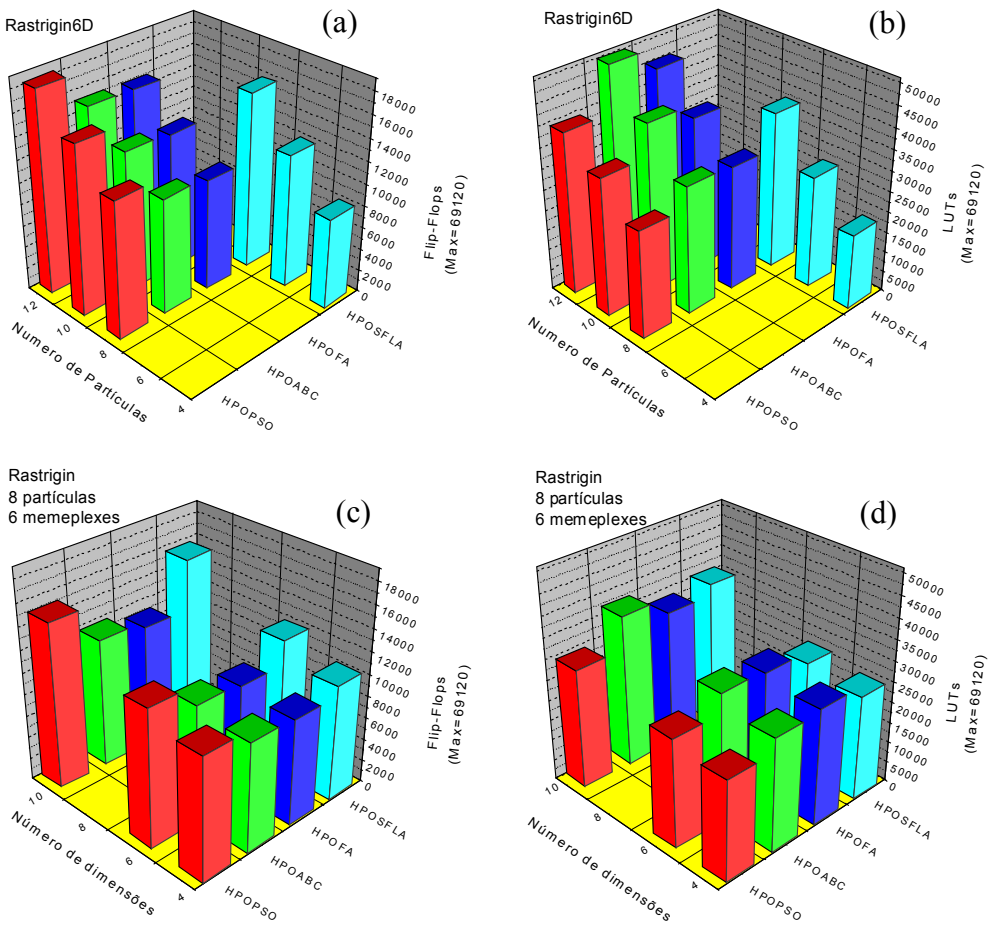


Figura 4.25: Comparação do consumo de recursos para a função *Rastrigin*

operadores aritméticos para o movimento dos vaga-lumes. Entretanto, no caso da função custo *Rastrigin*, a arquitetura HPOABC consome mais LUTs (aproximadamente 50200 LUTs no pior caso) do que as outras arquiteturas.

De forma geral, quanto mais recursos de *hardware* estejam disponíveis melhores serão os resultados obtidos pelas ferramentas de síntese e roteamento e, consequentemente, melhor será o desempenho dos circuitos implementados.

No capítulo anterior foi demonstrado que o tamanho da representação aritmética afeta consideravelmente o consumo de blocos DSPs embarcados. Implementações das arquiteturas paralelas dos algoritmos HPPSO, HPABC e HPSFLA [151], [149], [152], [150], baseadas em uma representação de 32 bits, demonstram que o consumo de blocos DSPs embarcados é um fator de extrema importância que afeta tanto o desempenho dos circuitos como a escalabilidade dos algoritmos. Nestas primeiras versões evidenciou-se um alto consumo de blocos DSPs, em alguns casos excedendo os recursos disponíveis. To-

davia, a implementação das arquiteturas usando uma representação de 27 bits permitiu diminuir em um 50% o consumo de blocos DSPs. Segundo os dados de síntese obtidos, as arquiteturas HPOFA e HPOABC requerem um maior número de multiplicadores, sendo a pior situação encontrada para a implementação de 8 fontes de alimento ou vaga-lumes paralelas otimizando a função *Rastrigin* de 10 dimensões, para a qual foram necessários 25 blocos DSPs (aproximadamente 39% dos blocos DSPs disponíveis).

De forma geral, a frequência de operação dos circuitos oscila em torno de 130 MHz. Como esperado a frequência de operação dos circuitos diminui conforme aumenta o número de partículas paralelas. A arquitetura HPOPSO apresenta o melhor desempenho em termos da frequência de operação, alcançando 142 MHz para as funções $Esfera$, *Quadric* e *Rosenbrock* e 130 MHz para a função *Rastrigin*. Entretanto, a arquitetura HPOABC apresenta a frequência de operação mais baixa devido ao uso da unidade de divisão em ponto flutuante, que segundo os dados reportados no capítulo anterior possui uma frequência menor (aproximadamente 121 MHz) em comparação com as unidades de soma (164 MHz) e multiplicação (130 MHz).

Com base nos dados obtidos para a variação de dimensionalidade, mantendo constante o número de partículas paralelas, o aumento no consumo de flip-flops foi menor para as arquiteturas HPOABC e HPOFA enquanto o aumento no consumo de LUTs foi menor para a arquitetura HPOPSO. A arquitetura HPOSFLA apresentou, em geral, o maior aumento no consumo de recursos de *hardware* (flip-flops e LUTs). Conforme explicado anteriormente, os requerimentos de memória para o armazenamento da posição das partículas das arquiteturas HPSFLA e HPOSFLA é maior inclusive quando poucos *memeplexes* paralelos são usados. Nas arquiteturas HPOABC e HPOFA, para uma representação numérica de 27 bits e um aumento de ND dimensões são necessários $27 \times ND$ bits a mais para armazenar a posição de cada indivíduo. No caso da arquitetura HPOPSO o número de bits adicionais é o dobro da situação anterior devido à memória individual de cada partícula. Entretanto, no caso da arquitetura HPOSFLA, um aumento de ND dimensões em uma implementação de M *memeplexes* com F indivíduos cada um, são necessário $27 \times ND \times M \times F$ bits adicionais para representar a posição dos indivíduos.

Os dados de síntese reportam que para o aumento do número de partículas, mantendo constante a dimensionalidade dos problemas, a arquitetura HPOABC apresenta um aumento menor no consumo de flip-flops enquanto a arquitetura HPOPSO apresenta um aumento menor no consumo de LUTs. Nesta situação, novamente a arquitetura

HPOSFLA apresentou o maior aumento no consumo de recursos de *hardware* (flip-flops e LUTs).

É importante salientar que uma variação no tamanho do enxame tem uma influencia maior no consumo de recursos de *hardware* do que uma variação na dimensionalidade dos problemas. O aumento de uma partícula requer do uso de lógica sequencial e combinacional para a implementação dos operadores aritméticos e trigonométricos tanto na implementação das equações de atualização da posição (movimento das partículas) como na implementação de uma função custo em paralelo.

4.15 TESTES PARA VERIFICAÇÃO DAS IMPLEMENTAÇÕES

No intuito de verificar o correto funcionamento das implementações de *hardware*, dois tipos de teste foram realizados.

No primeiro teste, os resultados de simulação comportamental da implementação *hardware* com representação de 64 bits foram comparados passo a passo com os resultados da implementação *software* no Matlab, o qual utiliza uma representação numérica de 64 bits em ponto flutuante. Para isto, foi implementado um algoritmo PSO de 4 partículas resolvendo a função *Rastrigin* de duas dimensões. As implementações *hardware* e *software* usaram os mesmos parâmetros de configuração do algoritmo e a mesma posição inicial das partículas. Entretanto, tendo em conta as diferenças dos métodos de geração de números aleatórios entre as duas implementações, uma primeira validação foi realizada sem os componentes aleatórios no cálculo da velocidade das partículas. Posteriormente, foi implementado no Matlab o algoritmo de geração de número aleatórios utilizado na implementação *hardware* (vide figura 4.1), permitindo realizar comparações mais eficientes. Os resultados destas comparações foram positivos, verificando assim a corretude da arquitetura HPPSO proposta.

O segundo teste para verificação foi aplicado para todas as arquiteturas de *hardware* dos algoritmos de inteligência de enxames, propostas neste trabalho. Neste teste, comparou-se o resultado final (valor final da função custo e da posição final da melhor solução) da execução dos algoritmos no dispositivo FPGA com os respectivos resultados obtidos pelo processo de simulação comportamental (usando o ModelSim). Os resultados foram idênticos para todas as arquiteturas aplicadas nos problemas *bench-*

mark, mostrando que as implementações de *hardware* dos algoritmos por inteligência de enxames estão garantidamente *corretas*.

4.16 RESULTADOS DE CONVERGÊNCIA

As arquiteturas de *hardware* dos algoritmos de otimização por inteligência de enxames foram validadas após o respectivo mapeamento e implementação no dispositivo FPGA escolhido. A figura 4.26 mostra o ambiente de validação utilizado. Uma comunicação serial RS-232 foi utilizada para enviar os comandos de configuração dos algoritmos e decodificar os resultados obtidos. Um ambiente de validação foi desenvolvido no Matlab para realizar as referidas tarefas. Adicionalmente um osciloscópio digital foi utilizado para medir o tempo de execução dos algoritmos.

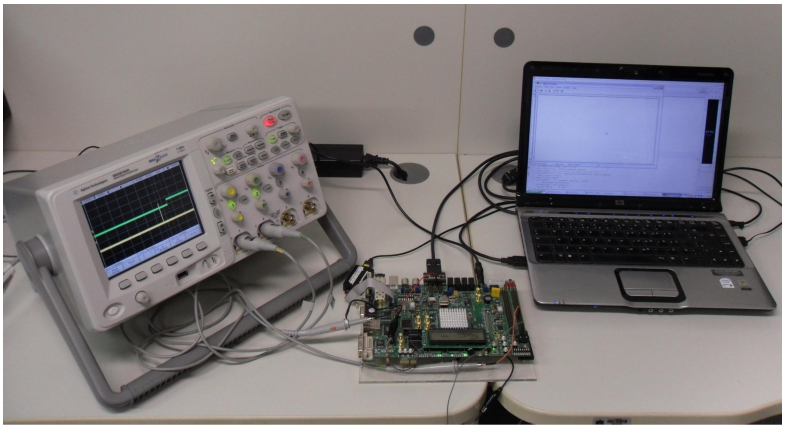


Figura 4.26: Ambiente de validação das arquiteturas de *hardware* desenvolvidas

As arquiteturas propostas foram validadas para o caso de minimização das quatro funções de teste *Esfera*, *Quadric*, *Rosenbrock* e *Rastrigin*, para as quais se conhece *a priori* o valor ótimo (vide seção 4.12). Para cada função de teste foram usados 16 experimentos, cada um com diferentes posições iniciais das partículas. Isto foi feito modificando manualmente os *dip switches* da placa de desenvolvimento, os quais estão conectados diretamente a um registro que contém a semente inicial da unidade de geração de números aleatórios encarregada de criar as posições iniciais das partículas.

Em posse da melhor posição global e da melhor aptidão obtida para cada experimento, foi calculado o valor médio, mediana, desvio padrão e o valor de aptidão mínimo entre todos os experimentos. As condições experimentais para cada algoritmo são

apresentadas na tabela 4.7. O espaço de busca foi limitado à faixa [-8.0,8.0]. O valor de aptidão máximo admissível (*threshold*) foi configurado em 0.01 para as funções unimodais e para a função *Rastrigin*, enquanto foi configurado em um valor de 1.0 para a função *Rosenbrock*.

Tabela 4.7: Condições experimentais

Implementação	Parâmetro	Valor
	Tamanho do enxame	8,10,12
	Dimensionalidade	4,6,10
HPPSO	Número de iterações	15000
HPOPSO	Max. iterações sem mudança de aptidão	100
	Peso de inercia	[0.8,0.1]
	Coeficientes cognitivo e social	c_1=c_2=2.5
	Velocidade máxima	[-6.0,6.0]
	Fontes de alimento	8,10,12
	Dimensionalidade	4,6,10
HPABC	Número de iterações	15000
HPOABC	Max. iterações para enviar escoteiras	20
	Max. iterações para aplicar OBL	20
	Max. iterações sem mudança de aptidão	100
	Número de vaga-lumes	8,10,12
	Dimensionalidade	4,6,10
HPFA	Número de iterações	10000
HPOFA	Max. iterações para aplicar OBL	40
	Coeficiente de absorção γ	0.8
	Coeficiente de atração inicial β_0	0.8
	Coeficiente de atração minimo β_{min}	0.2
	Parâmetro m	2.0
	Coeficientes de ajuste de aleatoriedade α	[1.0,0.001]
	Número de *memeplexes*	4,6,8
	Número de sapos por *memeplex*	4
HPSFLA	Dimensionalidade	4,6,10
HPOSFLA	Número de iterações	2000
	Máx. número de iterações locais	20
	Embaralhamento máximo	10
	Salto máximo	[-5.0,5.0]

No intuito de realizar uma análise de desempenho em função da qualidade da solução obtida entre as implementações em FPGAs dos algoritmos básicos e as implementações em FPGAs dos algoritmos baseados no aprendizado em oposição, algumas comparações numéricas foram realizadas usando os resultados de convergência para a solução dos problemas de otimização multimodal. Portanto, as arquiteturas de *hardware* foram aplicadas unicamente para as funções custo *Rosenbrock* e *Rastrigin*. Adicionalmente, as mesmas condições experimentais apresentadas na tabela anterior foram utilizadas para as implementações de *hardware* e *software* permitindo realizar comparações de desempenho entre as duas abordagens.

As tabelas 4.8, 4.9, 4.10, 4.11 apresentam os dados numéricos das soluções obtidas para as arquiteturas de *hardware* que implementam os algoritmos PSO, ABC, FA e SFLA, respectivamente. Os dados são apresentados de forma a facilitar uma comparação numérica entre as abordagens com e sem aprendizado por oposição.

Tabela 4.8: Comparação de convergência entre as arquiteturas HPPSO e HPOPSO

Problema	Implementação *hardware*	Média	Mediana	Mínimo	Desvio Padrão	Número acertos
Rosenbrock 4D	HPPSO	0.6287	2.58E-4	5.63E-7	1.3462	13/16
	HPOPSO	0.1957	2.18E-5	1.42E-7	0.7811	15/16
Rastrigin 4D	HPPSO	6.06E-5	5.40E-5	1.47E-5	2.71E-5	16/16
	HPOPSO	5.13E-5	4.85E-5	7.00E-6	2.09E-5	16/16
Rosenbrock 6D	HPPSO	3.8298	3.3423	4.67E-3	4.8690	7/16
	HPOPSO	2.6272	1.2557	0.0146	3.7816	8/16
Rastrigin 6D	HPPSO	8.18E-5	8.87E-5	4.46E-5	1.86E-5	16/16
	HPOPSO	8.35E-5	7.77E-5	2.42E-5	2.55E-5	16/16
Rosenbrock 10D	HPPSO	12.1166	12.8623	0.1455	7.3299	2/16
	HPOPSO	8.7331	7.4097	0.2075	7.7184	2/16
Rastrigin 10D	HPPSO	0.0623	1.38E-4	9.49E-5	0.2487	15/16
	HPOPSO	1.42E-4	1.43E-4	5.85E-5	4.82E-5	16/16

Com base no valor médio e número de acertos da tabela 4.8 é possível concluir que as duas arquiteturas apresentam boas soluções para a função multimodal *Rastrigin*, enquanto que a qualidade das soluções obtidas foi inferior para o caso da função *Rosenbrock*. Como esperado o desempenho dos algoritmos diminui conforme aumenta a complexidade dos problemas de otimização (aumento da dimensionalidade). Uma comparação entre o valor médio e a mediana permite concluir que a arquitetura HPOPSO, a qual utiliza a técnica de aprendizado em oposição, melhora os resultados obtidos para a função custo multimodal *Rosenbrock*. No caso da função custo *Rastrigin*, foi observada uma melhoria na solução obtida unicamente para o caso de maior dimensionalidade.

Tabela 4.9: Comparação de convergência entre as arquiteturas HPABC e HPOABC

Problema	Implementação *hardware*	Média	Mediana	Mínimo	Desvio Padrão	Número acertos
Rosenbrock 4D	HPABC	6.74E-4	4.86E-4	9.21E-5	9.30E-4	16/16
	HPOABC	2.76E-3	2.11E-3	3.68E-5	2.60E-3	16/16
Rastrigin 4D	HPABC	4.73E-5	5.12E-5	1.09E-5	2.32E-5	16/16
	HPOABC	5.64E-5	5.24E-5	1.45E-5	2.10E-5	16/16
Rosenbrock 6D	HPABC	0.0124	7.99E-3	9.15E-4	0.0139	16/16
	HPOABC	0.0102	8.91E-3	6.21E-4	0.0108	16/16
Rastrigin 6D	HPABC	5.79E-5	4.84E-5	2.23E-5	2.74E-5	16/16
	HPOABC	7.66E-5	6.54E-5	3.62E-5	3.01E-5	16/16
Rosenbrock 10D	HPABC	0.6382	0.5269	0.3537	0.3507	14/16
	HPOABC	0.6239	0.5446	0.2068	0.3928	13/16
Rastrigin 10D	HPABC	1.14E-4	1.29E-4	1.44E-5	5.30E-5	16/16
	HPOABC	1.22E-4	1.31E-4	3.85E-5	3.11E-5	16/16

Observa-se, a partir dos dados reportados na tabela 4.9, que as arquiteturas HPABC

e HPOABC encontram, na maioria dos casos, o valor ótimo para ambos os problemas multimodais. Conforme aumenta a dimensionalidade dos problemas menor é a qualidade da solução obtida. No pior caso, observado para a função *Rosenbrock*10D existe uma taxa de acerto de 81.25% e 87.5% para as abordagens com e sem aprendizado em oposição, respectivamente. Assim, neste caso é possível concluir que a abordagem de aprendizado em oposição não contribui com o aprimoramento da solução obtida.

A tabela 4.10 mostra que a arquitetura HPFA, a qual implementa o algoritmo FA original proposto por Yang [74] (vide Algoritmo 3, Seção 2.5), possui um desempenho baixo na solução dos problemas de otimização multimodais estudados neste trabalho. Entretanto, a arquitetura de *hardware* HPOFA, que implementa o algoritmo GOFA proposto (vide Algoritmo 9, Seção 4.8), apresenta uma melhoria considerável nas soluções obtidas para a função *Rastrigin*, conseguindo uma taxa de acerto alta para todas as dimensionalidades. Adicionalmente, foi observado um aprimoramento da qualidade da solução obtida para a função *Rosenbrock*, obtendo uma taxa de acerto superior para os casos de baixa dimensionalidade.

Tabela 4.10: Comparação de convergência entre as arquiteturas HPFA e HPOFA

Problema	Implementação *hardware*	Média	Mediana	Mínimo	Desvio Padrão	Número acertos
Rosenbrock 4D	HPFA	9.5905	10.9928	1.7528	4.4379	0/16
	HPOFA	0.6599	0.5996	0.1532	0.5132	13/16
Rastrigin 4D	HPFA	5.1614	4.9748	1.09E-4	3.7794	1/16
	HPOFA	6.73E-5	6.49E-5	2.05E-5	2.95E-5	16/16
Rosenbrock 6D	HPFA	9.4744	10.0992	2.7108	4.2008	0/16
	HPOFA	0.9220	0.9275	0.0381	0.5217	8/16
Rastrigin 6D	HPFA	21.1429	20.3967	3.9799	11.7640	0/16
	HPOFA	8.72E-5	8.89E-5	3.89E-5	1.98E-5	16/16
Rosenbrock 10D	HPFA	11.0203	9.1553	5.0206	5.1038	0/16
	HPOFA	3.6639	3.6879	0.0344	2.2179	2/16
Rastrigin 10D	HPFA	35.9430	35.3212	20.8942	10.1523	0/16
	HPOFA	1.37E-4	1.25E-4	6.53E-5	5.11E-5	16/16

Com base nos valores do valor médio e mediana apresentados na tabela 4.11 é possível concluir que as arquiteturas HPSFLA e HPOSFLA apresentam resultados similares para a solução do problema multimodal *Rosenbrock*. Observa-se que no caso de maior dimensionalidade *Rosenbrock*10D a arquitetura com aprendizado em oposição consegue uma melhoria de aproximadamente 47% da qualidade das soluções obtidas. Por outro lado, no caso da função *Rastrigin*, a arquitetura HPOSFLA obtém resultados mais próximos da solução ótima se comparado com a arquitetura HPSFLA, porém o número de acertos é baixo se comparado com os resultados conseguidos pelos outros algoritmos.

Tabela 4.11: Comparação de convergência entre as arquiteturas HPSFLA e HPOSFLA

Problema	Implementação *hardware*	Média	Mediana	Mínimo	Desvio Padrão	Número acertos
Rosenbrock 4D	HPSFLA	0.7311	0.1057	9.24e-3	1.1693	12/16
	HPOSFLA	0.3140	0.3027	0.0552	0.1964	16/16
Rastrigin 4D	HPSFLA	3.3573	2.6282	0.7709	2.3290	0/16
	HPOSFLA	0.3447	1.39E-3	1.69E-5	1.0841	11/16
Rosenbrock 6D	HPSFLA	1.8297	0.4878	0.0827	2.6600	11/16
	HPOSFLA	1.0015	0.9705	0.4090	0.3164	9/16
Rastrigin 6D	HPSFLA	7.1223	5.3190	2.5969	4.0435	0/16
	HPOSFLA	0.9295	0.0316	2.40E-5	1.8135	7/16
Rosenbrock 10D	HPSFLA	6.8651	5.9619	0.1879	4.4013	1/16
	HPOSFLA	3.3180	2.8040	2.1392	2.0227	0/16
Rastrigin 10D	HPSFLA	28.7984	27.8743	11.3177	10.2359	0/16
	HPOSFLA	1.7903	0.0254	5.16E-5	4.0624	7/16

Uma comparação numérica dos resultados de convergência entre as arquiteturas implementadas pode ser realizada com base nas tabelas 4.12, 4.13, 4.14 e 4.15, as quais apresentam os resultados obtidos para as funções custo *Esfera*, *Quadric*, *Rosenbrock* e *Rastrigin*, respectivamente. Nestas tabelas os resultados de desempenho são apresentados para diferentes tamanhos de enxame e dimensionalidades dos problemas de otimização.

Tabela 4.12: Convergência das implementações em *hardware*. Função *Esfera*

Número de dimensões	Partículas paralelas	Implementação *hardware*	Média	Mediana	Mínimo	Desvio Padrão	Número acertos
4	8 partículas	HPOPSO	1.8E-38	1.7E-38	8.4E-39	5.4E-39	16/16
	8 fontes alimento	HPOABC	1.5E-26	3.4E-32	3.0E-38	6.2E-26	16/16
	8 vaga-lumes	HPOFA	6.50E-9	5.38E-9	2.32E-9	3.30E-9	16/16
	6 *memeplexes*	HPOSFLA	2.11E-9	1.2E-16	1.7E-21	8.41E-9	16/16
6	8 partículas	HPOPSO	2.5E-38	2.4E-38	1.1E-38	1.1E-38	16/16
	8 fontes alimento	HPOABC	8.2E-27	6.2E-38	4.9E-38	3.0E-26	16/16
	8 vaga-lumes	HPOFA	1.13E-8	1.18E-8	4.97E-9	3.93E-9	16/16
	6 *memeplexes*	HPOSFLA	3.3E-10	3.1E-15	5.9E-22	9.9E-10	16/16
10	8 partículas	HPOPSO	4.2E-38	4.1E-38	2.8E-38	8.1E-39	16/16
	8 fontes alimento	HPOABC	2.1E-30	1.1E-37	9.7E-38	8.1E-30	16/16
	8 vaga-lumes	HPOFA	2.71E-8	2.55E-8	1.17E-8	1.10E-8	16/16
	6 *memeplexes*	HPOSFLA	1.2E-10	3.7E-13	5.1E-19	4.1E-10	16/16
6	10 partículas	HPOPSO	2.7E-38	2.8E-38	1.1E-38	7.6E-39	16/16
	10 fontes alimento	HPOABC	4.6E-30	5.9E-38	4.4E-38	1.8E-29	16/16
	10 vaga-lumes	HPOFA	1.04E-8	1.05E-8	1.55E-9	4.30E-9	16/16
	4 *memeplexes*	HPOSFLA	1.8E-13	5.9E-16	3.9E-25	3.3E-13	16/16
6	12 partículas	HPOPSO	2.0E-38	2.1E-38	5.4E-39	8.0E-39	16/16
	12 fontes alimento	HPOABC	1.4E-27	5.9E-38	4.8E-38	5.8E-27	16/16
	12 vaga-lumes	HPOFA	1.34E-8	1.51E-8	3.79E-9	5.25E-9	16/16
	8 *memeplexes*	HPOSFLA	1.7E-12	1.3E-14	1.4E-19	3.1E-12	16/16

Os seguintes comentários podem ser destacados:

- A partir da tabela 4.12 é possível concluir que todas as arquiteturas de *hardware*

Tabela 4.13: Convergência das implementações em *hardware*. Função *Quadric*

Número de dimensões	Partículas paralelas	Implementação *hardware*	Média	Mediana	Mínimo	Desvio Padrão	Número acertos
4	8 partículas	HPOPSO	2.2E-38	2.1E-38	5.2E-39	8.3E-39	16/16
	8 fontes alimento	HPOABC	1.7E-35	3.6E-36	3.0E-37	3.5E-35	16/16
	8 vaga-lumes	HPOFA	6.85E-9	5.48E-9	6.5E-10	7.61E-9	16/16
	6 *memeplexes*	HPOSFLA	6.30E-8	7.9E-11	2.1E-16	1.64E-7	16/16
6	8 partículas	HPOPSO	3.8E-38	3.7E-38	2.1E-38	7.7E-39	16/16
	8 fontes alimento	HPOABC	7.1E-22	1.1E-33	1.9E-35	2.1E-21	16/16
	8 vaga-lumes	HPOFA	2.14E-8	2.09E-8	7.95E-9	9.85E-9	16/16
	6 *memeplexes*	HPOSFLA	6.34E-8	4.2E-10	1.1E-13	1.59E-7	16/16
10	8 partículas	HPOPSO	2.8E-17	3.1E-18	8.3E-23	4.4E-17	16/16
	8 fontes alimento	HPOABC	1.75E-4	3.48E-5	8.70E-7	5.04E-3	16/16
	8 vaga-lumes	HPOFA	6.90E-8	6.08E-8	3.46E-8	3.51E-8	16/16
	6 *memeplexes*	HPOSFLA	2.87E-5	6.87E-7	9.6E-10	8.32E-5	16/16
6	10 partículas	HPOPSO	3.4E-38	3.2E-38	1.6E-38	1.0E-28	16/16
	10 fontes alimento	HPOABC	1.7E-19	6.6E-38	1.9E-38	6.9E-19	16/16
	10 vaga-lumes	HPOFA	1.54E-8	1.21E-8	5.04E-9	1.06E-8	16/16
	4 *memeplexes*	HPOSFLA	5.81E-7	2.0E-10	6.1E-18	1.64E-6	16/16
6	12 partículas	HPOPSO	3.2E-38	3.4E-38	1.5E-38	8.6E-38	16/16
	12 fontes alimento	HPOABC	9.0E-24	2.9E-32	1.7E-35	3.5E-23	16/16
	12 vaga-lumes	HPOFA	1.91E-8	1.61E-8	3.57E-9	1.49E-8	16/16
	8 *memeplexes*	HPOSFLA	6.34E-8	4.2E-10	1.1E-13	1.59E-7	16/16

Tabela 4.14: Convergência das implementações em *hardware*. Função *Rosenbrock*

Número de dimensões	Partículas paralelas	Implementação *hardware*	Média	Mediana	Mínimo	Desvio Padrão	Número acertos
4	8 partículas	HPOPSO	0.1957	2.18E-5	1.42E-7	0.7811	15/16
	8 fontes alimento	HPOABC	2.76E-3	2.11E-3	3.68E-5	2.60E-3	16/16
	8 vaga-lumes	HPOFA	0.6599	0.5996	0.1532	0.5132	13/16
	6 *memeplexes*	HPOSFLA	0.3140	0.3027	0.0552	0.1964	16/16
6	8 partículas	HPOPSO	2.6272	1.2557	0.0146	3.7816	8/16
	8 fontes alimento	HPOABC	0.0102	8.91E-3	6.21E-4	0.0108	16/16
	8 vaga-lumes	HPOFA	0.9220	0.9275	0.0381	0.5217	8/16
	6 *memeplexes*	HPOSFLA	1.0015	0.9705	0.4090	0.3164	9/16
10	8 partículas	HPOPSO	8.7331	7.4097	0.2075	7.7184	2/16
	8 fontes alimento	HPOABC	0.6239	0.5446	0.2068	0.3928	13/16
	8 vaga-lumes	HPOFA	3.6639	3.6879	0.0344	2.2179	2/16
	6 *memeplexes*	HPOSFLA	3.3180	2.8040	2.1392	2.0227	0/16
6	10 partículas	HPOPSO	1.3543	0.7720	1.42E-3	1.4962	9/16
	10 fontes alimento	HPOABC	5.70E-3	5.91E-3	1.59E-3	3.21E-3	16/16
	10 vaga-lumes	HPOFA	0.9006	1.0162	0.0137	0.4870	8/16
	4 *memeplexes*	HPOSFLA	1.3566	1.3404	0.6648	0.4193	2/16
6	12 partículas	HPOPSO	0.4372	0.0135	3.69E-4	0.8927	14/16
	12 fontes alimento	HPOABC	0.0118	6.61E-3	9.46E-4	0.0127	16/16
	12 vaga-lumes	HPOFA	0.7446	0.6916	0.2538	0.3898	12/16
	8 *memeplexes*	HPOSFLA	0.8836	0.9355	0.4353	0.3586	10/16

Tabela 4.15: Convergência das implementações em *hardware*. Função *Rastrigin*

Número de dimensões	Partículas paralelas	Implementação *hardware*	Média	Mediana	Mínimo	Desvio Padrão	Número acertos
4	8 partículas	HPOPSO	5.13E-5	4.85E-5	7.00E-6	2.09E-5	16/16
	8 fontes alimento	HPOABC	5.64E-5	5.24E-5	1.45E-5	2.10E-5	16/16
	8 vaga-lumes	HPOFA	6.73E-5	6.49E-5	2.05E-5	2.95E-5	16/16
	6 *memeplexes*	HPOSFLA	0.3447	1.39E-3	1.69E-5	1.0841	11/16
6	8 partículas	HPOPSO	8.35E-5	7.77E-5	2.42E-5	2.55E-5	16/16
	8 fontes alimento	HPOABC	7.66E-5	6.54E-5	3.62E-5	3.01E-5	16/16
	8 vaga-lumes	HPOFA	8.72E-5	8.89E-5	3.89E-5	1.98E-5	16/16
	6 *memeplexes*	HPOSFLA	0.9295	0.0316	2.40E-5	1.8135	7/16
10	8 partículas	HPOPSO	1.42E-4	1.43E-4	5.85E-5	4.82E-5	16/16
	8 fontes alimento	HPOABC	1.22E-4	1.31E-4	3.85E-5	3.11E-5	16/16
	8 vaga-lumes	HPOFA	1.37E-4	1.25E-4	6.53E-5	5.11E-5	16/16
	6 *memeplexes*	HPOSFLA	1.7903	0.0254	5.16E-5	4.0624	7/16
6	10 partículas	HPOPSO	7.37E-5	7.85E-5	3.32E-5	2.33E-5	16/16
	10 fontes alimento	HPOABC	8.71E-5	9.00E-5	3.20E-5	2.82E-5	16/16
	10 vaga-lumes	HPOFA	7.95E-5	7.50E-5	3.29E-5	3.54E-5	16/16
	4 *memeplexes*	HPOSFLA	0.3470	8.39E-5	2.33E-5	0.8733	12/16
6	12 partículas	HPOPSO	7.73E-5	8.22E-5	3.35E-5	2.94E-5	16/16
	12 fontes alimento	HPOABC	6.74E-5	6.35E-5	3.35E-5	2.65E-5	16/16
	12 vaga-lumes	HPOFA	8.45E-5	8.59E-5	2.58E-5	3.87E-5	16/16
	8 *memeplexes*	HPOSFLA	0.4568	8.88E-3	4.34E-5	0.8013	8/16

alcançam bons resultados para a solução da função *Esfera*, obtendo uma taxa de acerto de 100% para todas as dimensionalidades testadas. Observa-se que a arquitetura HPOPSO atinge soluções com um refinamento maior. Isto pode ser explicado considerando que o peso de inercia da equação de movimento da partícula (equação 2.3) é pequeno durantes as últimas iterações do algoritmo. Desta forma, evita-se que as partículas realizem mudanças consideráveis na posição, refinando assim a solução obtida mediante um processo de busca local.

- Observa-se que no caso da função *Esfera* a arquitetura HPOFA alcança o valor ótimo, porém com um refinamento menor se comparado com as outras arquiteturas. Este fato é devido à ausência de processos de busca local no algoritmo FA. Adicionalmente, nos casos estudados observou-se que o incremento no tamanho do enxame não afetou o resultado da solução obtida pelas arquiteturas implementadas.

- Os resultados de convergência para a solução da função *Quadric*, tabela 4.13, mostram uma taxa de acerto de 100% para todas as arquiteturas de *hardware* implementadas. De forma similar ao caso anterior, a arquitetura HPOPSO consegue as soluções com um valor mais próximo do valor ótimo, enquanto as arquiteturas HPOFA e HPSFLA possuem um refinamento da solução final menos eficiente. Observa-se ainda que o incremento de dimensionalidade para esta função custo

afeta consideravelmente o desempenho da arquitetura HPOABC. O valor médio alcançado por esta arquitetura para a solução do problema de 10 dimensões é aproximadamente 1, 4 e 13 ordens de magnitude maior se comparado com as soluções obtidas pelas arquiteturas HPOSFLA, HPOFA e HPOPSO, respectivamente. Adicionalmente, nos casos estudados observa-se que o incremento no tamanho do enxame não afeta o resultado da solução obtida pelas arquiteturas implementadas.

- No caso do problema de otimização multimodal *Rosenbrock*, observa-se que os melhores resultados são alcançados pela arquitetura HPOABC, que na maioria dos casos atinge uma taxa de acerto de 100%. No entanto, o desempenho desta arquitetura apresenta uma pequena diminuição para o problema de maior dimensionalidade em que as taxas de acerto alcançadas foram de 12.5%, 81.25%, 12.5% e 0% para as implementações HPOPSO, HPOABC, HPOFA e HPOSFLA, respectivamente. Outro ponto a ser destacado é que a arquitetura HPOFA alcança soluções mais próximas do valor ótimo do que a arquitetura HPOPSO para os problemas de maior dimensionalidade. Observa-se também que para a arquitetura HPOPSO, o valor médio e a mediana do valor da função custo são menores quando são usadas mais partículas paralelas. Desta forma, o aumento do número de partículas paralelas tem um impacto maior na arquitetura HPOPSO do que a arquitetura HPOFA.

- A tabela 4.15 apresenta os resultados de convergência para o problema de otimização multimodal *Rastrigin*. Observa-se que a taxa de acerto é de 100% para todas as arquiteturas exceto para a arquitetura HPOSFLA cujo desempenho está comprometido para problemas de 6 e 10 dimensões, casos em que a taxa de acerto é de 43.75%. Com base nestes resultados é possível concluir que para problemas altamente multimodais o desempenho das arquiteturas HPOPSO, HPOABC e HPOFA é similar sem importar o aumento no tamanho do enxame. Observa-se que o valor médio e a mediana para estas três arquiteturas são similares e que o aumento do número de partículas paralelas não representa um aprimoramento significativo da solução final.

4.17 TESTES DE SIGNIFICÂNCIA ESTATÍSTICA

Visando avaliar a capacidade de busca dos algoritmos implementados em *hardware* com significância estatística, a seguinte metodologia foi usada. Primeiramente, o teste

de Kolmogorov-Smirnov foi aplicado no intuito de verificar se os resultados de convergência seguem uma distribuição normal. No caso negativo o teste não paramétrico de Wilcoxon foi usado para comparar as medianas entre dois algoritmos e o teste de Kruskal-Wallis foi aplicado para comparar as medianas entre mais de dois algoritmos.

Foi usado um nível de confiança de 95%, istó é, uma significância de 5% ou valor-p inferior a 0.05. Desta forma, um valor-p inferior a 0.05 rejeita a hipótese nula (H_0), a qual assume que duas ou mais amostras independentes de dois ou mais algoritmos seguem distribuições com a mesma mediana. Em contrapartida, a hipótese alternativa (H_A) assume que as amostras independentes dos experimentos seguem distribuições com medianas diferentes. Nas tabelas mostradas a seguir, os testes com resultados positivos são marcados com o símbolo '+' na última coluna. Por outro lado, o símbolo '-' indica que não foi incontrada significância estatística (p >0.05) para a comparação da distribuição dos resultados entre os algoritmos. Adicionalmente, o melhor resultado para cada problema está destacado em cor cinza.

As tabelas 4.16, 4.17, 4.18 e 4.19 apresentam as medianas e a comparação de significância entre as arquiteturas com e sem aprendizado em oposição quando aplicadas nos problemas multimodais.

Tabela 4.16: Mediana e significância estatística para a arquitetura HPPSO

Problema	HPPSO	HPOPSO	
*Rosenbrock*4D	2.58E-4	2.18E-5	-
*Rosenbrock*6D	3.3423	1.2557	-
*Rosenbrock*10D	12.8623	7.4097	-
*Rastrigin*4D	5.40E-5	4.85E-5	-
*Rastrigin*6D	8.87E-5	7.77E-5	-
*Rastrigin*10D	1.38E-4	1.43E-4	-

Tabela 4.17: Mediana e significância estatística para a arquitetura HPABC

Problema	HPABC	HPOABC	
*Rosenbrock*4D	4.86E-4	2.11E-3	+
*Rosenbrock*6D	7.99E-3	8.91E-3	-
*Rosenbrock*10D	0.5269	0.5446	-
*Rastrigin*4D	5.12E-5	6.54E-5	+
*Rastrigin*6D	4.84E-5	5.24E-5	-
*Rastrigin*10D	1.39E-4	1.31E-4	-

Com base na tabela 4.16 observa-se que embora a arquitetura HPOPSO apresente uma mediana menor do que a arquitetura HPPSO, não é possível concluir que a arquitetura com aprendizado em oposição seja significativamente melhor do que a arquitetura

Tabela 4.18: Mediana e significância estatística para a arquitetura HPFA

Problema	HPFA	HPOFA	
*Rosenbrock*4D	10.9928	0.5996	+
*Rosenbrock*6D	10.0992	0.9275	+
*Rosenbrock*10D	9.1553	3.6880	+
*Rastrigin*4D	4.9748	6.49E-5	+
*Rastrigin*6D	20.3966	8.89E-5	+
*Rastrigin*10D	35.3212	1.25E-4	+

Tabela 4.19: Mediana e significância estatística para a arquitetura HPSFLA

Problema	HPSFLA	HPOSFLA	
*Rosenbrock*4D	0.1057	0.3027	-
*Rosenbrock*6D	0.4878	0.9704	-
*Rosenbrock*10D	5.9619	2.8040	+
*Rastrigin*4D	2.6282	1.39E-3	+
*Rastrigin*6D	5.3190	3.16E-2	+
*Rastrigin*10D	27.8742	2.54E-2	+

HPPSO. Isto é válido inclusive para o caso do problema *Rosenbrock* em que a arquitetura HPOPSO apresenta uma mediana consideravelmente menor para os problemas de maior dimensionalidade do que a arquitetura HPPSO.

A tabela 4.17 mostra que, na maioria dos casos, as implementações de *hardware* do algoritmo ABC com e sem aprendizado em oposição apresentam resultados similares, isto é, nenhuma das duas implementações é melhor que a outra. Apenas foi percebida significância estatística para os problemas de menor dimensionalidade em que a arquitetura HPABC apresenta melhores resultados do que a arquitetura HPOABC para o problema *Rosenbrock*, enquanto a arquitetura com aprendizado em oposição apresenta melhores resultados para o problema *Rastrigin*

A tabela 4.18 mostra que a implementação HPOFA, a qual usa a técnica de aprendizado em oposição, é significativamente melhor do que a arquitetura HPFA para todos os problemas multimodais.

Finalmente, a partir da tabela 4.19 pode-se observar que existe diferença estatística entre as arquiteturas HPSFLA e HPOSFLA para a maioria dos casos. É possível concluir que a arquitetura HPOSFLA apresenta resultados significativamente melhores para o problema *Rastrigin* e para o problema *Rosenbrock* de maior dimensionalidade. Existem dois casos (*Rosenbrock*4D e *Rosenbrock*6D) para os quais não é possível concluir que os resultados sejam significativamente diferentes, embora o valor da mediana seja inferior para a arquitetura HPSFLA.

Uma comparação entre os resultados obtidos pelas arquiteturas implementadas com aprendizado em oposição é apresentada na tabela 4.20. A arquitetura HPOPSO apresenta resultados significativamente melhores do que os outros algoritmos para todos os problemas unimodais. Entretanto, a arquitetura HPOABC apresenta resultados melhores para a maioria dos problemas multimodais. Nestes casos, todos os resultados têm significância estatística, como pode ser observado na última coluna da tabela, onde o símbolo '+' é encontrado. Adicionalmente, é importante destacar que para o problema multimodal *Rastrigin* os testes de significância estatística não mostraram resultados positivos se a comparação for feita apenas entre os três primeiros algoritmos (HPOPSO, HPOABC e HPOFA), pois os mesmos apresentam medianas similares.

Tabela 4.20: Medianas e significância estatística entre as arquiteturas

Problema	HPOPSO	HPOABC	HPOFA	HPOSFLA	
$Esfera$4D	1.7E-38	3.4E-32	5.38E-9	1.2E-16	+
$Esfera$6D	2.4E-38	6.2E-38	1.18E-8	3.1E-15	+
$Esfera$10D	4.1E-38	1.1E-37	2.55E-8	3.7E-13	+
$Quadric$4D	2.1E-38	3.6E-36	5.48E-9	7.9E-11	+
$Quadric$6D	3.7E-38	1.12E-33	2.09E-8	4.2E-10	+
$Quadric$10D	3.1E-18	3.48E-5	6.08E-8	6.87E-7	+
$Rosenbrock$4D	2.18E-5	2.11E-3	0.5996	0.3027	+
$Rosenbrock$6D	1.2557	8.92E-3	0.9275	0.9705	+
$Rosenbrock$10D	7.4097	0.5446	3.6880	2.8041	+
$Rastrigin$4D	4.85E-5	5.24E-5	6.49E-4	1.39E-3	+
$Rastrigin$6D	7.76E-5	6.54E-5	8.89E-5	3.16E-2	+
$Rastrigin$10D	1.43E-4	1.31E-4	1.25E-4	2.54E-2	+

4.18 COMPARAÇÃO DOS RESULTADOS DE CONVERGÊNCIA

As tabelas 4.21, 4.22, 4.23, 4.24 apresentam os resultados de convergência da implementação *software* para as funções custo *Esfera*, *Quadric*, *Rosenbrock* e *Rastrigin*, respectivamente. Para isto, os algoritmos foram codificados em linguagem C e foi usado um PC de escritório operando a 1.6GHz, 2 GB RAM, Windows XP OS. As tabelas mostram os resultados de convergência para as mesmas condições experimentais usadas nas implementações de *hardware*.

Com base nas tabelas 4.21 e 4.22 é possível concluir que os algoritmos propostos são efetivos para a solução de problemas de otimização unimodal. No entanto, observa-se que o algoritmo O-SFLA apresenta uma perda de eficiência em termos de refinamento da solução final para o caso da função *Quadric*, em que a posição da melhor solução global oscila em torno do valor teórico. Isto pode ser explicado dado a falta de um parâmetro de ajuste do salto dos sapos que possibilite um processo de busca local.

Tabela 4.21: Convergência das implementações em *software*. Função *Esfera*

Número de dimensões	Partículas paralelas	Algoritmo *software*	Média	Mediana	Mínimo	Desvio Padrão	Número acertos
4	8 partículas	O-PSO	0.0	0.0	0.0	0.0	16/16
	8 fontes alimento	GBOABC	0.0	0.0	0.0	0.0	16/16
	8 vaga-lumes	GOFA	0.0	0.0	0.0	0.0	16/16
	6 *memeplexes*	O-SFLA	0.0	0.0	0.0	0.0	16/16
6	8 partículas	O-PSO	0.0	0.0	0.0	0.0	16/16
	8 fontes alimento	GBOABC	0.0	0.0	0.0	0.0	16/16
	8 vaga-lumes	GOFA	6.25E-8	0.0	0.0	2.50E-7	16/16
	6 *memeplexes*	O-SFLA	2.24E-5	5.00E-7	0.0	6.54E-5	16/16
10	8 partículas	O-PSO	0.0	0.0	0.0	0.0	16/16
	8 fontes alimento	GBOABC	0.0	0.0	0.0	0.0	16/16
	8 vaga-lumes	GOFA	5.85E-5	3.55E-5	2.00E-6	8.43E-5	16/16
	6 *memeplexes*	O-SFLA	3.80E-4	1.50E-5	1.00E-6	1.30E-3	16/16
6	10 partículas	O-PSO	0.0	0.0	0.0	0.0	16/16
	10 fontes alimento	GBOABC	0.0	0.0	0.0	0.0	16/16
	10 vaga-lumes	GOFA	0.0	0.0	0.0	0.0	16/16
	4 *memeplexes*	O-SFLA	8.13E-7	0.0	0.0	2.17E-6	16/16
6	12 partículas	O-PSO	0.0	0.0	0.0	0.0	16/16
	12 fontes alimento	GBOABC	0.0	0.0	0.0	0.0	16/16
	12 vaga-lumes	GOFA	0.0	0.0	0.0	0.0	16/16
	8 *memeplexes*	O-SFLA	3.12E-7	0.0	0.0	1.01E-6	16/16

Tabela 4.22: Convergência das implementações em *software*. Função *Quadric*

Número de dimensões	Partículas paralelas	Algoritmo *software*	Média	Mediana	Mínimo	Desvio Padrão	Número acertos
4	8 partículas	O-PSO	0.0	0.0	0.0	0.0	16/16
	8 fontes alimento	GBOABC	0.0	0.0	0.0	0.0	16/16
	8 vaga-lumes	GOFA	1.88E-7	0.0	0.0	4.03E-7	16/16
	6 *memeplexes*	O-SFLA	3.60E-3	5.70E-4	0.0	6.80E-3	14/16
6	8 partículas	O-PSO	0.0	0.0	0.0	0.0	16/16
	8 fontes alimento	GBOABC	0.0	0.0	0.0	0.0	16/16
	8 vaga-lumes	GOFA	4.59E-5	2.90E-5	2.00E-6	5.40E-5	16/16
	6 *memeplexes*	O-SFLA	7.40E-3	5.90E-3	5.00E-4	8.10E-3	11/16
10	8 partículas	O-PSO	0.0	0.0	0.0	0.0	16/16
	8 fontes alimento	GBOABC	1.22E-4	1.31E-4	3.85E-5	3.11E-5	16/16
	8 vaga-lumes	GOFA	8.98E-4	7.26E-4	1.85E-4	5.53E-4	16/16
	6 *memeplexes*	O-SFLA	2.70E-2	2.16E-2	5.30E-3	2.54E-2	2/16
6	10 partículas	O-PSO	0.0	0.0	0.0	0.0	16/16
	10 fontes alimento	GBOABC	0.0	0.0	0.0	0.0	16/16
	10 vaga-lumes	GOFA	5.31E-6	2.00E-6	0.0	7.16E-6	16/16
	4 *memeplexes*	O-SFLA	1.85E-2	1.34E-2	2.30E-3	1.28E-2	7/16
6	12 partículas	O-PSO	0.0	0.0	0.0	0.0	16/16
	12 fontes alimento	GBOABC	0.0	0.0	0.0	0.0	16/16
	12 vaga-lumes	GOFA	8.31E-6	5.00E-6	0.0	1.23E-5	16/16
	8 *memeplexes*	O-SFLA	5.70E-3	2.90E-3	3.50E-5	7.60E-3	13/16

Tabela 4.23: Convergência das implementações em *software*. Função *Rosenbrock*

Número de dimensões	Partículas paralelas	Algoritmo *software*	Média	Mediana	Mínimo	Desvio Padrão	Número acertos
4	8 partículas	O-PSO	3.46E-5	0.0	0.0	1.29E-4	16/16
	8 fontes alimento	GBOABC	5.73E-3	6.87E-3	1.32E-3	2.57E-3	16/16
	8 vaga-lumes	GOFA	0.1372	0.0806	0.0188	0.1852	16/16
	6 *memeplexes*	O-SFLA	0.2740	0.1824	0.0069	0.2535	16/16
6	8 partículas	O-PSO	2.17E-2	6.78E-3	0.0	3.82E-2	16/16
	8 fontes alimento	GBOABC	5.35E-2	5.29E-2	1.78E-2	2.62E-2	16/16
	8 vaga-lumes	GOFA	0.6255	0.3019	0.1170	0.5240	11/16
	6 *memeplexes*	O-SFLA	0.8235	0.6756	0.0259	0.4830	11/16
10	8 partículas	O-PSO	3.3131	1.3371	1.08E-2	4.4977	8/16
	8 fontes alimento	GBOABC	0.8021	0.8255	0.4098	0.1899	13/16
	8 vaga-lumes	GOFA	2.6204	3.1022	0.0387	1.3418	3/16
	6 *memeplexes*	O-SFLA	3.0691	2.0452	0.9497	2.3982	1/16
6	10 partículas	O-PSO	2.52E-2	1.46E-4	0.0	7.26E-2	16/16
	10 fontes alimento	GBOABC	4.31E-2	3.77E-2	5.44E-3	2.22E-2	16/16
	10 vaga-lumes	GOFA	0.2645	0.1691	0.0390	0.3448	15/16
	4 *memeplexes*	O-SFLA	1.1768	1.0964	0.4315	0.4901	5/16
6	12 partículas	O-PSO	3.33E-3	2.55E-5	0.0	1.29E-2	16/16
	12 fontes alimento	GBOABC	3.99E-2	3.69E-2	8.25E-3	2.02E-2	16/16
	12 vaga-lumes	GOFA	0.3798	0.1104	0.0501	0.5588	13/16
	8 *memeplexes*	O-SFLA	0.8984	0.8988	0.2258	0.4243	9/16

Tabela 4.24: Convergência das implementações em *software*. Função *Rastrigin*

Número de dimensões	Partículas paralelas	Algoritmo *software*	Média	Mediana	Mínimo	Desvio Padrão	Número acertos
4	8 partículas	O-PSO	6.22E-2	0.0	0.0	0.2487	15/16
	8 fontes alimento	GBOABC	0.0	0.0	0.0	0.0	16/16
	8 vaga-lumes	GOFA	0.3987	0.0	0.0	0.8591	13/16
	6 *memeplexes*	O-SFLA	0.0	0.0	0.0	0.0	16/16
6	8 partículas	O-PSO	0.8706	0.9950	0.0	0.7152	5/16
	8 fontes alimento	GBOABC	0.0	0.0	0.0	0.0	16/16
	8 vaga-lumes	GOFA	1.2655	2.25E-5	0.0	3.4811	14/16
	6 *memeplexes*	O-SFLA	2.49E-5	0.0	0.0	8.80E-5	16/16
10	8 partículas	O-PSO	3.6067	2.9849	0.0	1.7376	1/16
	8 fontes alimento	GBOABC	0.0	0.0	0.0	0.0	16/16
	8 vaga-lumes	GOFA	0.9665	2.84E-3	2.11E-4	3.8197	15/16
	6 *memeplexes*	O-SFLA	0.1279	3.78E-4	4.00E-6	0.3425	13/16
6	10 partículas	O-PSO	0.4975	0.0	0.0	0.7266	10/16
	10 fontes alimento	GBOABC	0.0	0.0	0.0	0.0	16/16
	10 vaga-lumes	GOFA	0.2370	3.00E-6	0.0	0.9479	15/16
	4 *memeplexes*	O-SFLA	5.82E-5	0.0	0.0	1.57E-4	16/16
6	12 partículas	O-PSO	0.3731	0.0	0.0	0.6160	11/16
	12 fontes alimento	GBOABC	0.0	0.0	0.0	0.0	16/16
	12 vaga-lumes	GOFA	1.1543	0.0	0.0	2.5308	13/16
	8 *memeplexes*	O-SFLA	1.88E-7	0.0	0.0	7.50E-7	16/16

Por outro lado, analisando os resultados de convergência apresentados na tabela 4.23 observa-se que o algoritmo GBOABC apresenta os melhores resultados para o caso da função *Rosenbrock*. Adicionalmente, observando o valor da mediana e número de acertos é possível concluir que o algoritmo O-PSO possui um desempenho melhor do que o algoritmo GOFA. Isto último difere da implementação *hardware*, onde a arquitetura HPOFA apresentou melhores soluções do que a arquitetura HPOPSO.

Finalmente, com base nos dados reportados na tabela 4.24 é possível demonstrar que o algoritmo GBOABC apresenta o melhor resultado no caso do problema de teste *Rastrigin*. Observa-se ainda que, com relação à mediana e número de acertos, o algoritmo GOFA possui um desempenho superior que o algoritmo O-PSO, especificamente nos casos de maior dimensionalidade.

Uma comparação numérica dos resultados de convergência entre as implementações de *hardware* e *software* pode ser realizada com base no conjunto de tabelas 4.12-4.15 e 4.21-4.24. No intuito de facilitar esta comparação, as tabelas 4.25, 4.26, 4.27 e 4.28 mostram os dados de convergência de forma agrupada para os algoritmos O-PSO, GBOABC, GOFA e O-SFLA, respectivamente. Apresenta-se o valor da mediana como parâmetro estatístico de comparação da qualidade da solução obtida pelos algoritmos implementados em *hardware* e *software*. Esta escolha obedece ao fato da mediana ser um estimador estatístico mais robusto do que a média amostral.

Tabela 4.25: Comparação de convergência HW/SW para o algoritmo O-PSO

Dimensões, Partículas	Implementação	Valor Mediana			
		Esfera	*Quadric*	*Rosenbrock*	*Rastrigin*
N=4	HPOPSO	1.7E-38	2.1E-38	2.18E-5	4.85E-5
S=8	O-PSO	0.0	0.0	0.0	0.0
N=6	HPOPSO	2.4E-38	3.7E-38	1.2557	7.77E-5
S=8	O-PSO	0.0	0.0	6.78E-3	0.9950
N=10	HPOPSO	4.1E-38	3.1E-18	7.4097	1.43E-4
S=8	O-PSO	0.0	0.0	1.3371	2.9849
N=6	HPOPSO	2.8E-38	3.2E-38	0.7720	7.85E-5
S=10	O-PSO	0.0	0.0	1.46E-4	0.0
N=6	HPOPSO	2.1E-38	3.4E-38	0.0135	8.22E-5
S=12	O-PSO	0.0	0.0	2.55E-5	0.0

Pode-se observar que no caso do algoritmo O-PSO, as implementações de *hardware* e *software* apresentam resultados similares para a solução dos problemas unimodais. Entretanto, no caso da função *Rosenbrock*, a implementação *software* apresenta melhores resultados e, no caso da função *Rastrigin*, ambas as implementações encontram o ótimo global, porém a solução de *software* realiza um refinamento melhor da solução final.

Tabela 4.26: Comparação de convergência HW/SW para o algoritmo GBOABC

Dimensões,	Implementação	Valor Mediana			
Fontes de alimento		*Esfera*	*Quadric*	*Rosenbrock*	*Rastrigin*
N=4	HPOABC	3.4E-32	3.6E-36	2.11E-3	5.24E-5
S=8	GBOABC	0.0	0.0	6.87E-3	0.0
N=6	HPOABC	6.2E-38	1.1E-33	8.91E-3	6.54E-5
S=8	GBOABC	0.0	0.0	5.29E-2	0.0
N=10	HPOABC	1.1E-37	3.47E-5	0.5446	1.31E-4
S=8	GBOABC	0.0	1.31E-4	0.8255	0.0
N=6	HPOABC	5.9E-38	6.6E-38	5.91E-3	9.00E-5
S=10	GBOABC	0.0	0.0	3.77E-2	0.0
N=6	HPOABC	5.9E-38	2.9E-32	6.61E-3	6.35E-5
S=12	GBOABC	0.0	0.0	3.69E-2	0.0

Tabela 4.27: Comparação de convergência HW/SW para o algoritmo GOFA

Dimensões,	Implementação	Valor Mediana			
Vagalumens		*Esfera*	*Quadric*	*Rosenbrock*	*Rastrigin*
N=4	HPOFA	5.38E-9	5.48E-9	0.5996	6.49E-5
S=8	GOFA	0.0	0.0	0.0806	0.0
N=6	HPOFA	1.18E-8	2.09E-8	0.9275	8.89E-5
S=8	GOFA	0.0	2.90E-5	0.3019	2.25E-5
N=10	HPOFA	2.55E-8	6.08E-8	3.6879	1.25E-4
S=8	GOFA	3.55E-5	7.26E-4	3.1022	2.84E-3
N=6	HPOFA	1.05E-8	1.21E-8	1.0162	7.50E-5
S=10	GOFA	0.0	2.00E-6	0.1691	3.00E-6
N=6	HPOFA	1.51E-8	1.61E-8	0.6916	8.59E-5
S=12	GOFA	0.0	5.00E-6	0.1104	0.0

Segundo a tabela 4.26, as implementações de *hardware* e *software* do algoritmo GBOABC apresentam resultados similares para a solução da função *Esfera*. No caso da função *Quadric*, o desempenho da arquitetura de *hardware* é superior para os problemas de maior dimensionalidade. Observa-se ainda que a arquitetura HPOABC apresenta uma leve melhoria no caso da função *Rosenbrock* em comparação com a respectiva solução de *software*. No caso da função *Rastrigin*, ambas as implementações encontram o ótimo global, porém a solução de *software* realiza um refinamento melhor da solução final.

A tabela 4.27 mostra que a implementação de *hardware* HPOFA apresenta soluções mais próximas do valor teórico do que a respectiva implementação *software* para os problemas unimodais, especificamente nos casos de maior dimensionalidade. Em contrapartida, a implementação *software* do algoritmo GOFA apresenta um melhor desempenho no caso da função *Rosenbrock*, porém os resultados obtidos são similares para os problemas de maior dimensionalidade. Um comportamento similar aos dois algoritmos anteriores é observado para o caso da função *Rastrigin*, em que a solução de *software* apresenta soluções com maior grau de refinamento da solução final.

Tabela 4.28: Comparação de convergência HW/SW para o algoritmo O-SFLA

Dimensões,	Implementação	Valor Mediana			
Memeplexes		*Esfera*	*Quadric*	*Rosenbrock*	*Rastrigin*
N=4	HPOSFLA	1.2E-16	7.9E-11	0.3027	1.39E-3
S=8	O-SFLA	0.0	5.70E-4	0.1824	0.0
N=6	HPOSFLA	3.1E-15	4.2E-10	0.9705	0.0316
S=8	O-SFLA	5.00E-7	5.90E-3	0.6756	0.0
N=10	HPOSFLA	3.7E-13	6.87E-7	2.8040	0.0254
S=8	O-SFLA	1.50E-5	2.16E-2	2.0452	3.78E-4
N=6	HPOSFLA	5.9E-16	2.0E-10	1.3404	8.39E-5
S=10	O-SFLA	0.0	1.34E-2	1.0964	0.0
N=6	HPOSFLA	1.3E-14	4.2E-10	0.9355	8.88E-3
S=12	O-SFLA	0.0	2.90E-3	0.8988	0.0

A tabela 4.28 mostra que a implementação *hardware* HPOSFLA apresenta melhores soluções para as funções unimodais, especificamente no caso da função *Quadric*, em que a solução de *software* consegue resultados inferiores aos outros algoritmos implementados. No caso da função *Rosenbrock*, a solução de *software* O-SFLA mostra uma leve melhoria se comparado com a respectiva implementação *hardware*. Finalmente, no caso da função *Rastrigin*, a solução de *software* apresenta resultados mais próximos do valor teórico do que a implementação *hardware*.

Neste ponto, é importante salientar que as diferenças observadas entre as implementações de *hardware* e *software* podem ser entendidas considerando os seguintes dois aspectos: (a) a unidade de geração de números aleatórios das arquiteturas de *hardware* usa o método LFSR (*Linear Feedback Shift Register*), enquanto a técnica LCG (*Linear Congruential Generator*) é usada pelo compilador *GCC* para a implementação da função *rand*. (b) as arquiteturas de *hardware* usam uma representação aritmética de 27 bits e, em consequência, a precisão dos cálculos aritméticos é menor se comparado com a representação de 32 bits utilizada na implementação de *software*. Este último aspecto contribui com o refinamento inferior da solução final observado pela implementação *hardware* do problema *Rastrigin*, em que a função cosseno é calculada com uma precisão menor do que na implementação em *software*.

4.19 COMPARAÇÃO DO TEMPO DE EXECUÇÃO

O tempo de execução dos algoritmos paralelos em *hardware* foi comparado com duas implementações em *software*. A primeira é baseada em uma implementação em código C executado em um processador Intel Core Duo, 2 GB RAM, operando a frequência de

1.6GHz e um sistema operativo Windows XP. A segunda implementação em *software* é baseada em um código C executado em um microprocessador MicroBlaze embarcado no mesmo dispositivo FPGA, com 64KB de memória de programa, operando a 100MHz, o qual foi desenvolvido usando a ferramenta Xilinx Platform Studio [157]. Esta segunda implementação permite fazer uma análise comparativa do desempenho dos algoritmos para a sua aplicação em sistemas embarcados, em que comumente as frequências de operação são baixas em relação às soluções convencionais baseadas em PCs.

No intuito de realizar uma comparação apropriada, os algoritmos paralelos foram validados utilizando a mesma frequência de operação da implementação MicroBlaze, isto é 100MHz. Adicionalmente, a implementação do MicroBlaze utiliza os mesmos elementos lógicos configuráveis (CLBs) do que a implementação *hardware*, permitindo assim uma análise de desempenho sob a mesma plataforma.

A figura 4.27 apresenta a arquitetura utilizada para a implementação do microprocessador embarcado em FPGA. Dois periféricos de *hardware* foram instanciados e conectados ao barramento PLB (*processor local bus*) do MicroBlaze: (a) Um contador de ciclos de relógio que permite medir o tempo de execução por iteração, (b) um bloco de comunicação serial RS-232 para enviar ao usuário os dados da solução do algoritmo e o tempo de execução do mesmo. O contador de ciclos de relógio é ativado durante as primeiras cinco iterações, evitando assim o *overflow* do contador.

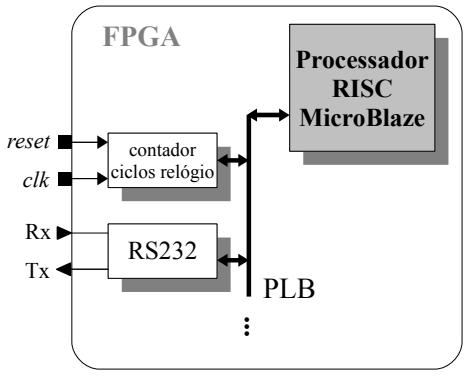


Figura 4.27: Implementação MicroBlaze

Para as implementações de *software* baseadas no processador Intel Core 2 Duo, foi calculado o tempo médio para 16 execuções do mesmo algoritmo. As mesmas condições experimentais mostradas na tabela 4.7 foram usadas para as implementações de *hardware* e de *software*.

A tabela 4.29 apresenta os resultados do tempo médio de execução por iteração para

os problemas de otimização de 10 dimensões.

As implementações de *hardware* possuem um tempo de execução por iteração na ordem de microssegundos. Pode-se observar que a implementação da arquitetura HPOPSO possui o menor tempo de execução por iteração. Por outro lado, a implementação HPOSFLA possui o maior tempo de execução, porém, é necessário lembrar que cada iteração de busca local da arquitetura HPOSFLA requer de 2 avaliações da função custo. Entretanto, a arquitetura HPOABC, que também realiza duas avaliações da função custo por iteração, apresenta o segundo melhor tempo de execução, com exceção da função *Rastrigin*.

Os tempos de execução por iteração no MicroBlaze são da ordem de milissegundos. É importante salientar que os dados obtidos na implementação MicroBlaze correspondem a uma aproximação do tempo de execução por iteração, pois os mesmos foram coletados para as primeiras cinco iterações de cada algoritmo. Contudo, é possível observar que a implementação GOFA no MicroBlaze possui o menor tempo médio de execução por iteração. A implementação do algoritmo O-PSO possui um tempo de execução maior para os problemas unimodais e as implementações GBOABC e O-SFLA apresentam um tempo de execução elevado para o problema multimodal *Rastrigin*.

A implementação de *software* usando o PC de escritório apresenta tempos de execução por iteração na ordem de microssegundos. Foi observado um comportamento similar ao caso da implementação no MicroBlaze, ou seja, a implementação GOFA apresenta o menor tempo médio de execução por iteração enquanto as implementações GBOABC e O-SFLA apresentam um tempo de execução maior para os problemas multimodais.

No intuito de realizar uma análise quantitativa dos resultados de tempo de execução, as tabelas 4.30 e 4.31 apresentam o fator de aceleração do tempo de execução entre as implementações de *hardware* e as implementações *software* no MicroBlaze e no PC de escritório, respectivamente.

Observa-se que o melhor fator de aceleração entre as implementações de *hardware* e de *software* usando o MicroBlaze é obtido pela arquitetura HPOPSO para as funções unimodais, obtendo fatores de 6811 e 5772 vezes para os problemas *Esfera* e *Quadric*, respectivamente. No caso da função *Rosenbrock*, as arquiteturas HPOPSO e HPOSFLA melhoram o tempo de execução por fatores de 6293 e 7030 vezes, respectivamente. No entanto, esta última arquitetura apresenta problemas de convergência

Tabela 4.29: Comparação do tempo de execução por iteração, 10 dimensões

Plataforma	Algoritmo	Tempo de Execução			
		Esfera	*Quadric*	*Rosenbrock*	*Rastrigin*
hardware,	HPOPSO	2.28μs	2.73μs	2.73μs	4.43μs
FPGA,	HPOABC	2.49μs	3.39μs	3.39μs	6.70μs
100MHz	HPOFA	3.06μs	3.51μs	3.51μs	4.68μs
	HPOSFLA	4.32μs	5.22μs	5.22μs	8.62μs
Software,	O-PSO	15.53 ms	15.76 ms	17.18 ms	38.27 ms
FPGA,	GBOABC	6.36 ms	7.02 ms	9.56 ms	59.13 ms
MicroBlaze,	GOFA	3.74 ms	4.06 ms	5.26 ms	32.48 ms
100MHz	O-SFLA	14.40 ms	17.56 ms	36.70 ms	277.0 ms
Software,	O-PSO	24.8μs	15.0μs	12.7μs	26.8μs
Intel Core2 Duo	GBOABC	39.6μs	15.5μs	12.1μs	37.4μs
2GB RAM, 1.6GHz	GOFA	4.2μs	5.0μs	4.2μs	18.6μs
Linguagem C	O-SFLA	35.0μs	37.0μs	33.0μs	136.0μs

Tabela 4.30: Fator de aceleração por iteração *hardware* versus MicroBlaze

Arquitetura	Fator de aceleração			
	Esfera	*Quadric*	*Rosenbrock*	*Rastrigin*
HPOPSO	6811	5772	6293	8638
HPOABC	2554	2070	2820	8825
HPOFA	1222	1156	1498	6940
HPOSFLA	3333	3363	7030	32134

para a solução da função *Rosenbrock*, especificamente para situações de maior dimensionalidade (vide tabela 4.14). Finalmente, as arquiteturas HPOABC e HPOSFLA melhoram o desempenho por fatores de 8825 e 32134 vezes, respectivamente, para a função *Rastrigin*, porém, segundo os resultados de convergência, a arquitetura HPOABC apresenta soluções mais próximas do valor ótimo para este problema (vide tabela 4.15).

A comparação do tempo de execução entre as implementações de *hardware* e *software* baseada no PC de escritório, mostram que a arquitetura HPOABC possui o melhor fator de aceleração, 15.9 vezes, para a solução do problema *Esfera*, enquanto a arquitetura HPOSFLA apresenta o melhor fator de aceleração para a solução da função *Quadric*. No caso da função *Rosenbrock*, as arquiteturas HPOPSO e HPOSFLA melhoram o tempo de execução por fatores de 4.65 e 6.32 vezes, respectivamente, porém a arquitetura HPOSFLA não apresentou resultados satisfatórios em termos de convergência. Finalmente, no caso da função *Rastrigin*, as arquiteturas HPOPSO e HPOSFLA apresentam fatores de aceleração de 6.05 e 15.78 vezes, respectivamente, porém os dados

Tabela 4.31: Fator de aceleração por iteração *hardware* versus PC escritório

Arquitetura	Fator de aceleração			
	Esfera	*Quadric*	*Rosenbrock*	*Rastrigin*
HPOPSO	10.88	5.49	4.65	6.05
HPOABC	15.90	4.57	3.57	5.58
HPOFA	1.37	1.42	1.20	3.97
HPOSFLA	8.11	7.09	6.32	15.78

de convergência mostram que a arquitetura HPOPSO apresentou melhores soluções.

A tabela 4.32 apresenta os resultados de tempo de execução total dos algoritmos estudados. Esta tabela objetiva a comparação do tempo de execução entre as implementações de *hardware* e *software* para o mesmo algoritmo (e não entre algoritmos), para o qual as mesmas condições experimentais, listadas na tabela 4.7, foram aplicadas para ambas as implementações. Observe-se que para todos os algoritmos em cada um dos problemas de otimização, as arquiteturas de *hardware* propostas operando a 100MHz possuem um tempo de execução menor do que a implementação *software* executada no PC de escritório a 1.6GHz.

Tabela 4.32: Comparação do tempo de execução total, 10 dimensões

Plataforma	Algoritmo	Tempo de Execução			
		Esfera	*Quadric*	*Rosenbrock*	*Rastrigin*
hardware,	HPOPSO	34.00 ms	40.80 ms	40.80 ms	66.40 ms
FPGA,	HPOABC	61.20 ms	100.80 ms	101.92 ms	182.40 ms
100MHz	HPOFA	28.00 ms	32.40 ms	32.20 ms	48.80 ms
	HPOSFLA	9.20 ms	10.40 ms	10.40 ms	17.60 ms
Software,	O-PSO	248 ms	150 ms	127 ms	268 ms
Intel Core2 Duo	GBOABC	2377 ms	932 ms	728 ms	2247 ms
2GB RAM, 1.6GHz	GOFA	42 ms	50 ms	42 ms	186 ms
Linguagem C	O-SFLA	35 ms	37 ms	33 ms	136 ms

Com base nos dados obtidos é possível concluir que o melhor fator de aceleração é obtido pela arquitetura HPOABC, a qual atinge fatores de aceleração de 38.83, 9.24, 7.14 e 12.31 vezes para os problemas *Esfera*, *Quadric*, *Rosenbrock* e *Rastrigin*, respectivamente. Adicionalmente, é possível observar que a arquitetura HPOFA possui os fatores de aceleração mais baixos, aproximadamente 1.5, 1.54, 1.30 e 3.81, para os mesmos problemas, respectivamente.

Um ponto importante a ser destacado é que para as implementações de *hardware*, o

tempo de execução foi medido considerando uma implementação em um dispositivo FPGA operando a 100MHz. No entanto, segundo os resultados de síntese, a frequência máxima de operação dos circuitos implementados oscila entre 121 MHz e 130 MHz. Em consequência, os dados reportados nas tabelas são uma aproximação do fator de aceleração, podendo ser maiores se a frequência máxima de operação dos circuitos for usada.

4.20 DISCUSSÕES FINAIS DO CAPÍTULO E CONTRIBUIÇÕES

Neste capítulo foram apresentadas as implementações de *hardware* dos algoritmos de otimização por inteligência de enxames estudados neste trabalho. A abordagem de aprendizado em oposição foi aplicada para cada algoritmo visando melhorar o desempenho dos mesmos.

Os resultados de síntese, convergência e tempo de execução mostram que a implementação em FPGAs dos algoritmos estudados é tecnologicamente viável para a resolução de problemas de otimização embarcada. Algumas considerações práticas foram feitas visando obter soluções de qualidade aceitável com baixo custo em área lógica e tempo de execução.

A primeira consideração é referente ao tamanho de palavra da representação aritmética. Segundo os resultados de síntese apresentados neste capítulo e nas publicações correlatas [147], [148], [149], [150], [151], [152], o número de multiplicadores dedicados disponíveis no FPGA limita a capacidade de paralelização dos algoritmos. Os algoritmos de inteligência de enxames foram implementados em *hardware* usando uma representação de 27 bits, permitindo economizar aproximadamente 50% dos multiplicadores dedicados se comparado com implementações de 32 bits. Neste sentido, os dados de custo em área lógica demonstram que os dispositivos FPGA são uma solução factível para a implementação dos algoritmos de otimização estudados. Em geral, o consumo de recursos de *hardware* é satisfatório, havendo suficiente área lógica para implementações futuras.

A segunda consideração é referente às modificações realizadas nos algoritmos originais. Estas modificações visam facilitar a implementação *hardware* dos algoritmos desenvolvidos de forma que o paralelismo intrínseco dos mesmos possa ser explorado eficientemente. Desta forma, foram propostos os algoritmos GBABC e GFA, os quais

fazem uso da informação social contida no indivíduo com o melhor valor de aptidão visando o melhoramento do resto de indivíduos do enxame.

Adicionalmente, a técnica de aprendizado em oposição (OBL), implementada usando um operador de negação para o caso de espaços de busca simétricos, mostrou-se eficiente para manter a diversidade do enxame, aprimorando a qualidade da solução obtida para os problemas *benchmark*. Os resultados de convergência demonstram que as implementações de *hardware* dos algoritmos propostos usando a técnica OBL apresentam uma mediana menor se comparado com as a implementações de *hardware* dos algoritmos sem uso do aprendizado em oposição. No entanto, a análise de significância estatística mostrou que a técnica OBL contribui em maior medida no aprimoramento da qualidade das soluções obtidas pelas arquiteturas HPOFA e HPOSFLA.

Segundo os resultados de convergência e significância estatística, na maioria dos problemas multimodais estudados, a arquitetura HPOABC apresenta melhores resultados se comparado com os outros algoritmos. No entanto, deve ser considerado que a arquitetura HPOPSO apresenta o menor consumo de recursos de *hardware* em termos de LUTs e DSPs e a maior frequência de operação entre as arquiteturas implementadas. Adicionalmente, a arquitetura HPOPSO apresenta o menor tempo de execução por iteração. Desta forma, e considerando o conjunto de resultados obtidos, a arquitetura HPOPSO é um candidato eficiente em função do fator custo/benefício para aplicações de otimização embarcada.

A seguir são relacionadas as contribuições pontuais referentes às implementações realizadas neste capítulo.

Conforme explicado no Capítulo 2, são encontrados na literatura alguns estudos que abordam as capacidades de paralelização do algoritmo PSO usando FPGAs. Foi explicado ainda que a maioria dos trabalhos prévios que abordam a implementação em FPGAs de algoritmos de otimização por inteligência de enxames utilizam uma representação aritmética de ponto fixo, sendo poucos os estudos realizados usando uma representação aritmética de ponto flutuante [125].

A tabela 4.33 mostra um quadro comparativo em termos do consumo de recursos, convergência e tempo de execução entre a arquitetura de *hardware* do algoritmo PSO proposta neste trabalho e os resultados dos trabalhos mais similares encontrados na literatura [125], [117], [119], [158], [159], [115].

Tabela 4.33: Comparação do custo, convergência e desempenho

Referencia	Parâmetro	Funções *benchmark*		
		$Esfera$	$Rosenbrock$	$Rastrigin$
Este trabalho	Flip-flops	10852	10604	14978
Xilinx Virtex5 xc5vlx110t	LUTs	17519	16813	29171
Ponto flutuante 27 bits	blocos BRAMs	0	0	0
Tamanho do enxame S=8	DSPs 23×18	16	8	16
Dimensionalidade N=10	Frequência (MHz)	130.576	142.167	130.382
	Aptidão (mediana)	4.1E-38	7.4097	1.43E-4
	Tempo por iteração	2.28μs	2.73μs	4.43μs
Tewolde *et. al* [160]	Slices	1523	1811	—
Xilinx Spartan3E xc3s500e	Blocos BRAMs	7	7	—
Ponto flutuante 32 bits	DSPs 18×18	8	9	—
Tamanho do enxame S=30	Frequência (MHz)	33.0	33.0	—
Dimensionalidade N=30	Aptidão (mínimo)	0.00941	98.91	—
	Tempo por iteração	826.7μs	829.1μs	—
Farahani *et. al*[119]	Elementos lógicos	1063	1040	1374
Altera Stratix 1s10es	DSPs 18×18	10	24	18
Ponto fixo 32 bits	Frequência (MHz)	71.0	53.6	60.3
Tamanho do enxame S=8	Aptidão (média)	—	—	—
Dimensionalidade N=3	Tempo por iteração	1.71μs	1.81μs	3.60μs
Palangpour *et. al*[117]	Flip-flops	9249	9940	—
Xilinx Virtex2 xc2vp30	LUTs	20873	25750	—
Ponto fixo — bits	DSPs 18×18	—	—	—
Tamanho do enxame S=20	Frequência (MHz)	21.0	25.0	—
Dimensionalidade N=10	Aptidão (mínimo)	1E-3	8.61	—
	Tempo por iteração	0.392μs	0.8μs	—

Em geral, a arquitetura proposta neste trabalho se destaca positivamente em diversos aspectos se comparado com as outras abordagens. No entanto, é importante salientar que, embora os trabalhos correlatos utilizem funções *benchmark* para validação das arquiteturas, a informação coletada não é suficiente para realizar comparações conclusivas. Isto é devido a que a tecnologia de implementação e as condições experimentais tais como representação aritmética, dimensionalidade do problema, tamanho do enxame e valores dos parâmetros de configuração são diferentes. Adicionalmente, é importante salientar que existem outros aportes reportados na literatura que implementam o algoritmo PSO em *hardware* usando dispositivos FPGAs [161], [162], [114], porém os mesmos não utilizam funções *benchmark* e, portanto, não possibilitam a comparação de desempenho.

Por outro lado, não existem trabalhos reportados na literatura científica que apresen-

tem implementações paralelas em FPGAs dos algoritmos ABC, FA e SFLA. Neste sentido, um aporte inédito do presente trabalho é a implementação *hardware* em FPGAs das arquiteturas paralelas dos algoritmos ABC, FA e SFLA usando a representação aritmética de ponto flutuante.

O estudo da implementação em FPGAs de algoritmos de otimização por inteligência de enxames requer de uma análise de escalabilidade em termos de consumo de recursos e desempenho. Neste trabalho, foi realizado um estudo sobre a capacidade de paralelização dos algoritmos estudados mediante o aumento do número de partículas paralelas e da dimensionalidade dos problemas de otimização, avaliando assim o impacto no consumo de recursos de *hardware* e no tempo de execução. É importante destacar que não existe um estudo similar com tal abrangência na literatura científica e, em consequência, este ponto constitui um aporte do presente trabalho.

Por outro lado, a implementação em FPGAs dos métodos de adição de diversidade artificial para o algoritmo PSO, são também uma contribuição do trabalho. Os três métodos implementados, *atrativo repulsivo*, *congregação passiva seletiva* e *aprendizado em oposição*, fazem uso de operadores simples, facilitando o seu mapeamento em FPGAs. Como mostrado no Apêndice B e nas publicações correlatas, os resultados de convergência mostram que os métodos de adição de diversidade melhoram a qualidade da solução obtida pelo algoritmo PSO básico. Especificamente, o método de aprendizado em oposição apresentou o melhor fator custo benefício em termos de consumo de recursos, qualidade da solução final e tempo de execução.

Com base na análise anterior, a aplicação do método de aprendizado em oposição (OBL) nos algoritmos ABC, FA e SFLA, assim como sua respectiva implementação em FPGAs, constitui também um aporte inédito do trabalho.

Capítulo 5 APLICAÇÃO EM PROBLEMAS DE OTIMIZAÇÃO EMBARCADA

No capítulo anterior alguns experimentos usando funções *benchmark* foram descritos visando demonstrar a viabilidade das arquiteturas propostas na solução de problemas de otimização global. Este capítulo introduze uma arquitetura de *hardware* para a solução de problemas de otimização embarcada. A arquitetura proposta utiliza a implementação HPOPSO para realizar o treinamento *online* de um controlador neural de um robô móvel. Esta solução de *hardware* foi mapeada em um dispositivo FPGA e validada em um ambiente de simulação. Um comportamento de evasão de obstáculos foi utilizado na aplicação de robôs móveis, permitindo demonstrar que a arquitetura proposta é útil para situações em que falhas nos sensores do robô demandam o ajuste do controlador em tempo real. Este capítulo está divido da seguinte forma. Primeiramente é feita uma descrição geral do problema. Em seguida é apresentado o modelo neural utilizado para o controle do robô móvel. Posteriormente é feita a descrição da arquitetura de *hardware* proposta e são apresentados os resultados de síntese e validação usando um ambiente de simulação. Finalmente, é feita uma discussão do capítulo.

5.1 DESCRIÇÃO DO PROBLEMA

A seguir são descritos dois tipos de problemas de otimização embarcada que envolvem o treinamento *online* de controladores neurais de robôs móveis autônomos.

1. *Aproximação de modelos comportamentais*: algumas aplicações em robótica móvel requerem do desenvolvimento de modelos matemáticos que permitam emular a forma como um operário controla um robô perante tarefas específicas. Um exemplo prático deste tipo de aplicação é o caso de robôs submarinos utilizados em plataformas de extração de petróleo em alto mar. Estes robôs são controlados manualmente visando realizar tarefas diversas. A informação referente aos sistemas sensoriais e de posicionamento são enviados ao usuário de forma que possa efetuar as ações de controle pertinentes.

Nestas aplicações, o processo cognitivo do operário representa o modelo comportamental que deve ser aproximado mediante processos de aprendizado que comumente fazem uso de métodos heurísticos, por exemplo, redes neurais artificiais, visando a implementação de aproximadores para imitar um comportamento desejado. Entretanto, em plataformas portáteis em que as capacidades computacionais são limitadas, o processo de aprendizado exige de um tempo de execução elevado. Neste trabalho, a arquitetura de *hardware* HPOPSO é utilizada para acelerar o processo de aprendizado de um robô móvel. Este processo é realizado por meio do treinamento *online* de uma rede neural, envolvendo o problema de minimização do erro de aproximação entre as saídas obtidas pelo modelo matemático e as saídas desejadas.

2. *Ajuste do controlador na presença de falhas nos sensores*: o segundo tipo de problema estudado envolve o treinamento *online* do controlador neural do robô móvel quando uma ou mais falhas nos sensores são detectadas e isoladas. Nesta situação, deseja-se garantir o correto funcionamento do robô, isto é, um comportamento pré-estabelecido deve ser mantido. Assim, é necessário especificar um sistema tolerante a falhas, com requisitos de portabilidade e curto tempo de execução, de forma que o controlador do robô possa ser ajustado sem pausas apreciáveis, garantindo o cumprimento de uma tarefa específica.

   É importante salientar que a detecção de falhas nos sensores é um problema concreto para o qual existem diversas soluções. Estudos sobre diagnóstico e prognóstico de falhas nos sensores de um robô móvel são importantes, porém estão fora do escopo deste trabalho. Assume-se aqui que os sensores possuem um modelo de observação conhecido e que um método de detecção e isolamento de falhas é possível de ser implementado. Entre estes métodos podem-se destacar as análises estatísticas de medições consecutivas dos sensores ou o uso de filtros de Kalman sintonizados para tipos específicos de falhas [163].

5.2 O MODELO NEURAL

O algoritmo de otimização HPOPSO foi escolhido para adaptar os pesos do controlador neural de um robô móvel de tração diferencial em uma situação de evasão de obstáculos. O robô trabalha em um ambiente de 4.5m x 3.0m com paredes ortogonais e obstáculos internos. A plataforma robótica está equipada com um anel de sete sensores de proximidade infravermelho que realizam medições de distância entre 20cm e

150cm, como mostrado na figura 5.1a.

No intuito de realizar um controle de velocidade do robô, utilizou-se uma rede neural *Perceptron* de uma camada e dois neurônios de saída, cada um formado por um modelo *Perceptron de Rosenblatt* com bias igual a zero. O primeiro neurônio corresponde à saída da velocidade angular w e o segundo da velocidade linear v do robô. Foi usada uma função de ativação sigmoide em ambos os neurônios. As entrada da rede são os valores dos sete sensores infravermelho, vide figura 5.1b, estabelecendo 14 pesos de conexão que devem ser ajustados durante o processo de treinamento. Desta forma, as ações de controle após a interação do robô com o ambiente externo podem ser realizadas mediante a configuração das velocidades linear e angular. O cálculo final das velocidades de cada roda ($\dot{\theta}_{R,L}$) podem ser calculadas usando a equação da cinemática inversa de um robô com tração diferencial, equação 5.1.

$$\begin{bmatrix} \dot{\theta_L} \\ \dot{\theta_R} \end{bmatrix} = \frac{1}{2\pi r} \begin{bmatrix} 1 & -d/2 \\ 1 & d/2 \end{bmatrix} \times \begin{bmatrix} v \\ w \end{bmatrix} \tag{5.1}$$

onde d é a distância entre as rodas e r é o rádio das rodas. Na figura, l_0, l_1 e l_2 representam os sensores do lado esquerdo, r_0, r_1 e r_2 os sensores do lado direito e fr é o sensor frontal.

Modelos similares têm sido previamente estudados e aplicados ao problema de treinamento não supervisionado de robôs móveis com tração diferencial, em que cada neurônio controla a velocidade de giro de um motor. Soluções usando algoritmos genéticos [164] e o PSO para aprendizado paralelo de enxames de robôs heterogêneos têm sido propostos na literatura [165] e [166].

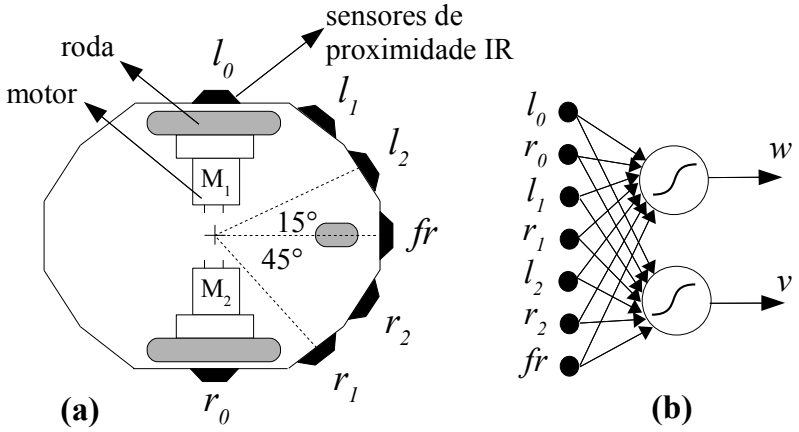


Figura 5.1: (a) Descrição do robô móvel, (b) Modelo do controlador neural

5.3 DESCRIÇÃO DA ARQUITETURA

A implementação em *hardware* do algoritmo HPOPSO, apresentada na figura 4.6, pode ser facilmente modificada para solucionar o problema de treinamento supervisionado do controlador neural. A principal modificação na arquitetura HPOPSO é a inclusão de uma memória ROM para armazenamento dos dados de treinamento e um protocolo de comunicação com o usuário/robô de forma que os pesos da rede possam ser utilizados pelo controlador da planta. A figura 5.2 apresenta a arquitetura de *hardware* proposta para o treinamento do controlador neural do robô móvel acima descrito.

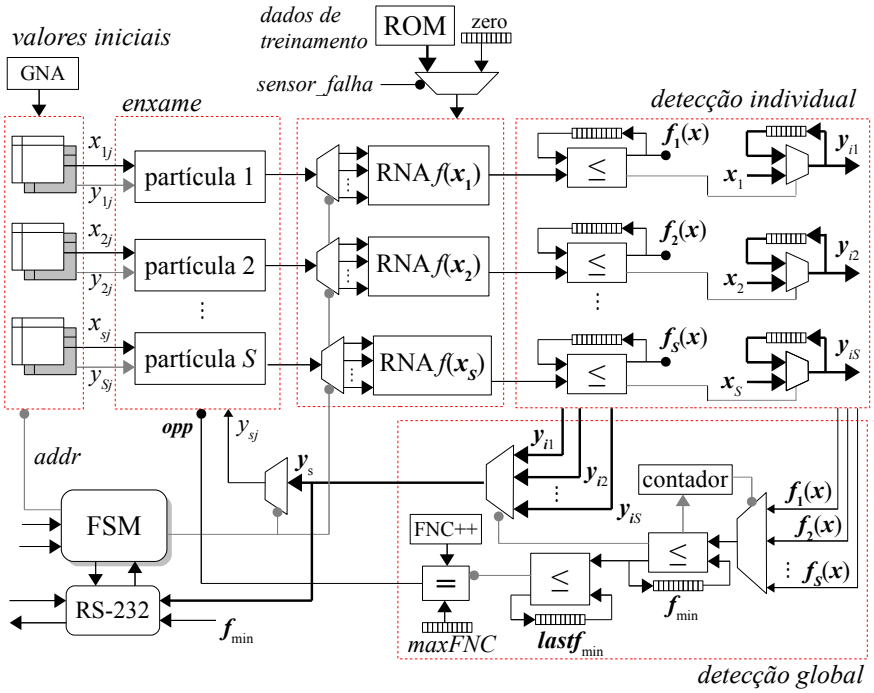


Figura 5.2: Arquitetura HPOPSO-RNA para treinamento do controlador neural

Esta arquitetura, chamada HPOPSO-RNA, utiliza a implementação em *hardware* do algoritmo HPOPSO com S partículas em paralelo. Observe-se que os dados de treinamento são armazenados em memórias ROM e utilizados durante o processo de avaliação da função custo, descrito na seguinte subseção. É importante salientar que os conjuntos de dados de treinamento (sete dados de entrada e duas saídas desejadas) são armazenados separadamente em blocos de memória embarcados no dispositivo FPGA, os quais podem ser acessados de forma simultânea. Adicionalmente, um módulo de comunicação serial RS-232 é utilizado para receber alguns comandos de entrada, tais como o sinal de *start* e o estado do vetor $sensorfalha$, assim como para enviar ao usuário a posição da melhor partícula no enxame $\boldsymbol{y}_s$, a qual possui a informação dos pesos da

rede após o processo de treinamento.

Neste problema de otimização global, cada partícula está constituída por quatorze variáveis de decisão (14 pesos de conexão), formando assim um problema de 14 dimensões (N=14). O sinal $sensorfalha$ é um vetor de sete posições binarias, que indica se algum dos sete sensores apresenta uma falha. Quando uma falha em um ou mais sensores infravermelho é detectada e isolada, a arquitetura rejeita a informação respectiva endereçando um valor nulo no dado de treinamento correspondente ao sensor em falha. Desta forma, o algoritmo HPOPSO pode treinar o controlador neural do robô móvel visando associar os valores dos outros sensores no intuito de compensar as falhas detectadas.

5.3.1 Implementação *hardware* da função custo

No processo de treinamento, cada partícula se movimenta aleatoriamente no espaço de busca 14-dimensional e valida seu desempenho por meio da avaliação do modelo do controlador neural do robô móvel. A função custo proposta para este fim está baseada na soma dos desvios quadráticos entre as saídas simuladas e as saídas desejadas (armazenadas na memória ROM), vide equação 5.2.

$$f_x = \sum_{i=1}^{NDT} (w_i - w_{di})^2 + \sum_{i=1}^{NDT} (v_i - v_{di})^2 \tag{5.2}$$

sendo NDT o número de amostras ou dados de treinamento e w_d e v_d as saídas desejadas para as velocidades angular e linear, respectivamente.

Desta forma, além do cálculo da função custo, é necessário implementar em *hardware* o modelo do controlador neural, apresentado na figura 5.1b, visando calcular, para cada conjunto de entradas, os valores de velocidade angular w e linear v.

A figura 5.3 apresenta a arquitetura de *hardware* para o processo de avaliação da função custo. Dois processos sequenciais são executados: (a) a avaliação do controlador neural e (b) o cálculo da função custo. Estes processos são realizados para cada conjunto de dados de treinamento, sendo que cada conjunto está constituído pelos valores dos sensores l_0, l_1, l_2, fr, r_2, r_1 e r_0 e as respectivas saídas desejadas w_d e v_d.

A arquitetura usa três unidades $FPadd$ e duas unidades $FPmul$, compartilhadas em diferentes estados. Uma FSM sincroniza os sinais de *start* e *ready* dos operadores nos

diferentes estados e atualiza o sinal de controle de alguns multiplexadores (não mostrados na figura) utilizados para endereçar os pesos e os dados de entrada. Durante o processo de avaliação do controlador neural, as entradas dos sensores são multiplicadas pelos respectivos pesos de conexão e os resultados são acumulados (modelo *Perceptron*). Este processo é realizado em paralelo para cada neurônio. Posteriormente é calculada a função de ativação sigmoide de cada neurônio (função $atan()$), a qual é implementada usando as três unidades $FPadd$ para implementar a arquitetura $Cordicatan$ mostrada na figura 3.7 do Capítulo 3. Em pose dos valores de velocidade angular w e linear v é avaliada a função custo mediante a implementação da equação 5.2. Posteriormente, um novo conjunto de dados de entrada pode ser simulado. Para cada novo conjunto de dados de entrada é avaliada a função custo, o resultado é acumulado e o contador i é incrementado. Após NDT avaliações, a FSM ativa o sinal de $ready$ indicando que o valor final na função custo $f(x)$ está disponível.

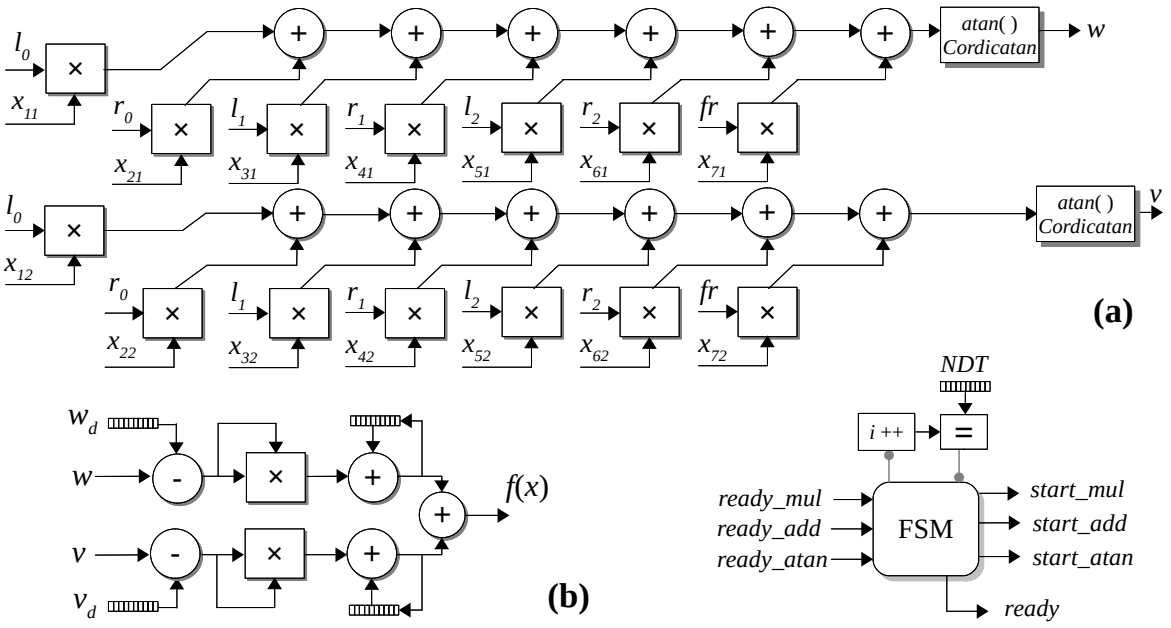


Figura 5.3: (a)Arquitetura do modelo neural (b) Arquitetura da função custo

Dado que o processo de avaliação da função custo depende do movimento prévio das partículas (algoritmo PSO síncrono), é possível compartilhar os recursos de *hardware* utilizados pelos processos de movimento das partículas e de avaliação da função custo. Portanto, as mesmas unidades $FPadd$ e $FPmul$ da arquitetura anterior são usadas para a implementação da arquitetura para o movimento das partículas, descrita na figura 4.5 do capítulo anterior.

5.4 O AMBIENTE DE DESENVOLVIMENTO

A ferramenta de simulação de robôs móveis *EyeSim* foi utilizada para efeitos de validação da arquitetura em *hardware* HPOPSO-RNA. A ferramenta *Eyesim* fornece os modelos matemáticos dos motores de tração, sensores de posicionamento, sensores inerciais, câmeras, assim como da cinemática de diversos robôs móveis comerciais [167]. O *EyeSim* pode ser executado em plataforma Windows e usa a linguagem de programação C para desenvolvimento das rotinas de controle (compilador gcc).

A figura 5.4 apresenta a estrutura do sistema de validação, o qual está constituído de um PC no qual se realiza a simulação do robô móvel e a detecção de falha dos sensores e de um dispositivo FPGA no qual foi mapeada a arquitetura em *hardware* do algoritmo HPOPSO-RNA para reconfiguração do controlador neural.

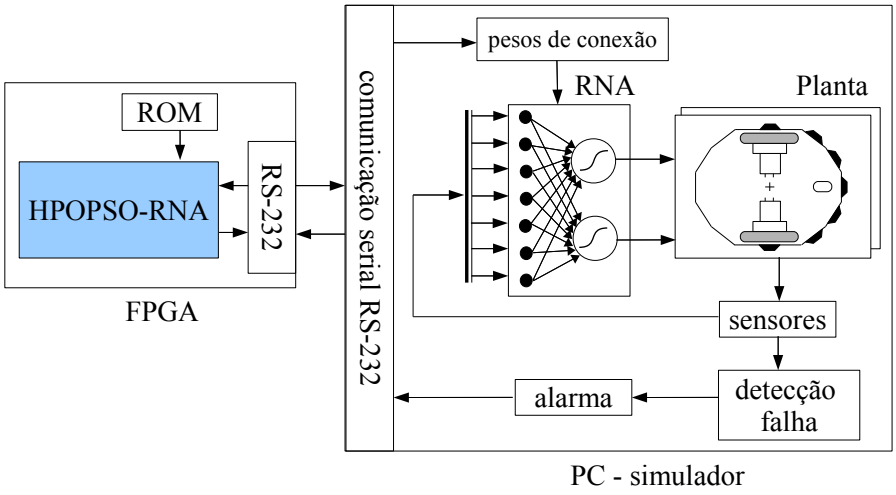


Figura 5.4: Plataforma de validação

Os dados de treinamento foram obtidos mediante uma operação manual do robô (usando o teclado), no ambiente de simulação. Os valores de distância dos sensores e as respectivas ações de controle (velocidades angular e linear) foram armazenados em arquivos de texto e utilizados para a criação dos arquivos de inicialização das memórias embarcadas no FPGA. A configuração de espaço livre mostrada na figura 5.5 foi utilizada para obter os dados de treinamento. Dois tipos de comportamentos foram usados visando realizar trajetórias livres de colisão. No primeiro, o robô realizou trajetórias em sentido anti-horário, mantendo o percurso no centro da configuração do espaço livre. No segundo comportamento, o robô realizou trajetórias em sentido anti-horário mantendo-se o mais próximo possível das paredes externas. Para cada tipo de comportamento foram

obtidas 808 amostras que constituem os dados de treinamento, isto é, NDT=808 na equação 5.2.

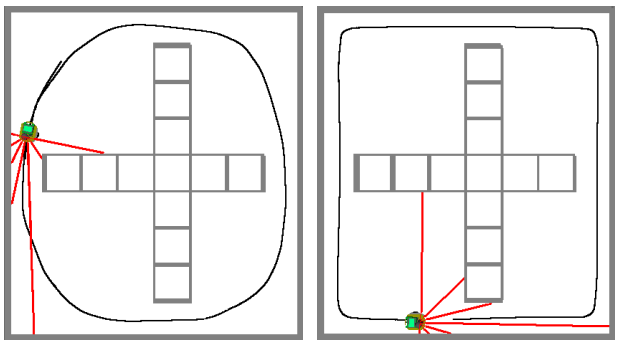


Figura 5.5: Ambiente para obtenção dos dados de treinamento. (a) Percurso no centro da configuração do espaço livre. (b) Percurso no contorno das paredes externas.

5.5 RESULTADOS DE SÍNTESE

Nesta seção apresentam-se os resultados de síntese da arquitetura HPOPSO-RNA implementada em FPGA. O circuito foi descrito em linguagem de descrição de *hardware* VHDL para uma representação aritmética de 27 bits e sintetizada usando a ferramenta XST da Xilinx.

A tabela 5.1 apresenta as condições experimentais e configuração dos parâmetros do algoritmo HPOPSO-RNA usadas para a etapa de implementação e simulação.

Tabela 5.1: Configuração de parâmetros, 10 partículas

Parâmetro	Valor
Tamanho do enxame	10
Dimensionalidade	14
Máximo número de iterações	5000
$maxFNC$ Max. iterações sem mudança de aptidão	30
Tamanho do espaço de busca	[-15, 15]
Valor de *threshold*	1E-7
Peso de inercia	[0.9 a 0.1]
Coeficiente cognitivo	c_1=2.0
Coeficiente social	c_2=2.0
Velocidade máxima	[-7.0, 7.0]

A tabela 5.2 mostra os dados de síntese da arquitetura HPOPSO-RNA mapeada em um dispositivo FPGA Virtex5 (chip xc5vlx110t). O consumo de recursos de *hardware* é representado em termos de flip-flops (FF), Lookup Tables (LUTs), blocos de RAM e blocos DSP. A frequência de operação máxima do circuito é em torno de 130 MHz.

Tabela 5.2: Resultados de síntese da arquitetura HPOPSO-RNA

Arquitetura	FF	LUTs	RAM	DSP48E	Freq.
	69120	69120	148	64	MHz
HPOPSO-RNA 27bits	20499(27.7%)	43595(63.1%)	9(6.1%)	20(31.2%)	130.761

Com base nestes resultados é possível concluir que a arquitetura proposta é viável para ser mapeada no dispositivo FPGA selecionado, havendo mais de 36% dos recursos disponíveis para futuras implementações. É importante destacar que existem dispositivos FPGA da família Virtex5 que possuem maior quantidade de recursos de *hardware*, permitindo usar mais partículas em paralelo, possibilitando assim a solução de problemas de otimização de maior complexidade.

5.6 RESULTADOS DE SIMULAÇÃO

A tabela 5.3 apresenta os pesos da rede obtidos após o processo de treinamento para cada um dos comportamentos simulados. Observe-se que no caso do comportamento 1, ou seja, percursos no centro da configuração do espaço livre, a contribuição dos pesos de conexão dos sensores l_i e r_i, $i = \{0, 1, 2\}$, é maior para o cálculo da velocidade angular w (neurônio 1) do que para o cálculo da velocidade linear (neurônio 2). Por outro lado, a contribuição do sensor frontal fr é bastante maior para o cálculo da velocidade linear v do que para a velocidade angular w. Este modelo, indica que o robô aumenta ou decrementa a velocidade linear segundo a distância frontal aos obstáculos enquanto realiza giros com maior ou menor velocidade segundo as distâncias laterais aos obstáculos.

No caso do comportamento 2, ou seja, percursos próximos às paredes externas, a contribuição dos pesos de conexão dos sensores do lado esquerdo (l_0, l_1 e l_2) é pequena para o cálculo das velocidades angular w (conexão com o neurônio 1) e linear v (conexão com o neurônio 2). Entretanto, a contribuição dos pesos de conexão dos sensores do lado direito (r_0, r_1 e r_2) e do sensor frontal são maiores para o cálculo das velocidades linear e angular. Este fato possibilita ao robô realizar percursos dando maior peso aos

sensores do lado direito, viabilizando trajetórias em sentido anti-horário próximas às paredes externas.

Tabela 5.3: Pesos da rede obtidos pela arquitetura HPOPSO-RNA

Comportamento 1		Comportamento 2	
Pesos de conexão	Valor	Pesos de conexão	Valor
$y_{s00} = l_0 - neuronio1$	0.069869	$y_{s00} = l_0 - neuronio1$	0.024279
$y_{s10} = r_0 - neuronio1$	0.041551	$y_{s10} = r_0 - neuronio1$	0.855401
$y_{s20} = l_1 - neuronio1$	0.182169	$y_{s20} = l_1 - neuronio1$	0.307474
$y_{s30} = r_1 - neuronio1$	-0.188934	$y_{s30} = r_1 - neuronio1$	-0.378247
$y_{s40} = l_2 - neuronio1$	0.149396	$y_{s40} = l_2 - neuronio1$	0.052946
$y_{s50} = r_2 - neuronio1$	-0.031821	$y_{s50} = r_2 - neuronio1$	-0.504158
$y_{s60} = fr - neuronio1$	-0.079617	$y_{s60} = fr - neuronio1$	-0.064920
$y_{s01} = l_0 - neuronio2$	-0.003519	$y_{s01} = l_0 - neuronio2$	-0.057201
$y_{s11} = r_0 - neuronio2$	0.006841	$y_{s11} = r_0 - neuronio2$	-0.912649
$y_{s21} = l_1 - neuronio2$	-0.005563	$y_{s21} = l_1 - neuronio2$	0.032133
$y_{s31} = r_1 - neuronio2$	-0.007534	$y_{s31} = r_1 - neuronio2$	1.471020
$y_{s41} = l_2 - neuronio2$	-0.011730	$y_{s41} = l_2 - neuronio2$	0.002326
$y_{s51} = r_2 - neuronio2$	-0.006720	$y_{s51} = r_2 - neuronio2$	0.361089
$y_{s61} = fr - neuronio2$	0.368095	$y_{s61} = fr - neuronio2$	0.019588

A figura 5.6 ilustra as trajetórias obtidas para os dois tipos de comportamento. As linhas contínuas correspondem aos percursos realizados manualmente para obter os dados de treinamento. As linhas tracejadas correspondem à trajetória realizada pelo robô após o processo de treinamento. Pode-se se observar que a rede neural treinada pela arquitetura HPOPSO-RNA consegue aproximar ambos os comportamentos simulados.

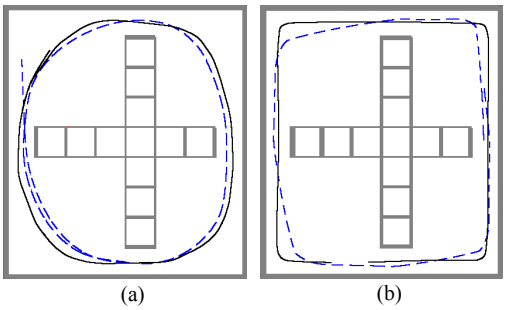


Figura 5.6: Resultados de simulação para o comportamento de evasão de obstáculos. (a) percurso no meio da configuração de espaço livre, (b) percurso próximo às paredes externas.

No caso anterior, a simulação foi feita para ao mesmo ambiente usado durante a coleta dos dados de treinamento. No entanto, uma forma de validar a eficiência dos

resultados obtidos pelo processo de treinamento da rede é estudar a capacidade do robô se locomover em ambientes desconhecidos. A figura 5.7 mostra as trajetórias seguidas pelo robô móvel após o processo de treinamento do controlador neural usando ambientes desconhecidos. As trajetórias do lado esquerdo da figura correspondem ao comportamento para percursos no centro do espaço livre, enquanto as trajetórias do lado direito da figura correspondem aos percursos mantendo-se próximo das paredes externas. Adicionalmente, os pontos vermelhos indicam colisão com os obstáculos.

É importante salientar que os conjuntos de testes de validação foram realizados em ambientes com diferentes graus de complexidade. Em geral, o robô móvel consegue realizar os percursos desejados para a maioria das situações. No entanto, foram observadas algumas colisões para o segundo tipo de comportamento, especificamente nos casos em que o robô encontra obstáculos contíguos às paredes (vide figuras 5.7h e 5.7j). Contudo, deve ser lembrado que durante a coleta dos dados de treinamento esta situação não foi considerada.

5.7 TESTES DE TOLERÂNCIA A FALHA NOS SENSORES

Uma das principais vantagens de usar arquiteturas de *hardware* para treinamento de redes neurais artificiais é a possibilidade de obter soluções de qualidade aceitável em um pequeno período de tempo se comparado com soluções convencionais em *software*. No caso de aplicações embarcadas, por exemplo, em robótica móvel, é indispensável o uso de plataformas de alto desempenho que possibilitem a reconfiguração *online* dos controladores, visando se adaptar a novas situações.

Neste trabalho a arquitetura HPOPSO-RNA foi aplicada para a reconfiguração do controlador neural no caso de falha nos sensores infravermelho do robô móvel. Após a detecção e isolação da falha enviam-se os códigos dos sensores em falha para o dispositivo FPGA, iniciando um novo treinamento da rede neural, porém ignorando a informação relacionada com os sensores em falha. A tabela 5.4 apresenta os valores dos pesos obtidos para três situações de falha: (a) no sensor fr,(b) nos sensores l_2 e r_2 e (c) nos sensores r_0 e l_0. A segunda coluna mostra os pesos obtidos durante a operação normal do robô (sem falha no anel de sensores). As simulações foram feitas para os percursos no centro da configuração do espaço livre.

Observe-se que no caso de falha no sensor fr os sensores vizinhos r_2 e l_2 (pesos y_{s41}

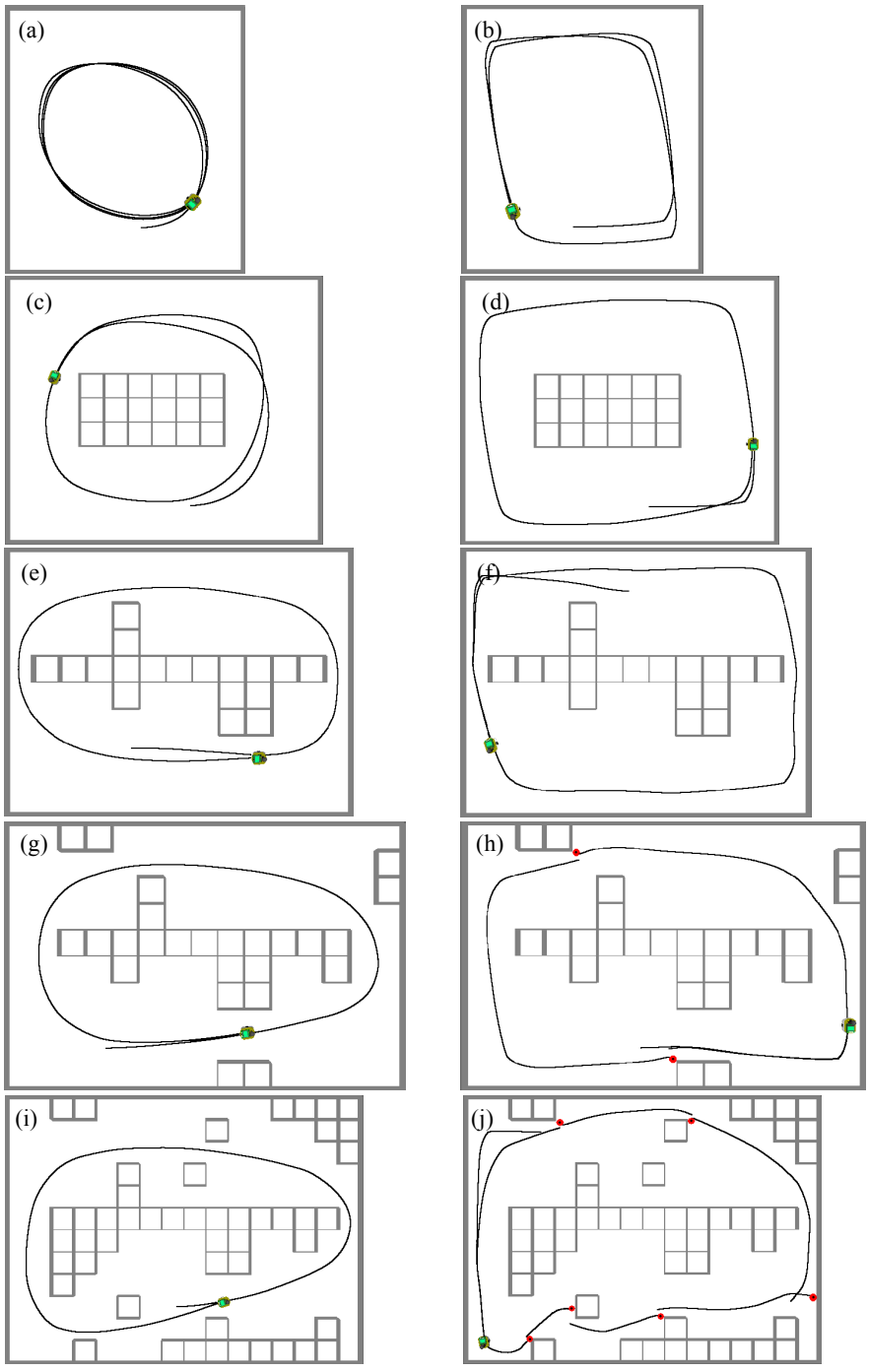


Figura 5.7: Resultados de simulação para ambientes desconhecidos. A coluna esquerda representa o comportamento para percursos no centro do espaço livre e a coluna da direita representa os percursos mantendo-se próximo das paredes externas. Os pontos vermelhos indicam colisões na trajetória.

Tabela 5.4: Pesos da rede obtidos pela arquitetura HPOPSO-RNA quando falhas nos sensores IR são detectadas

Pesos de conexão	Sem falha	falhas nos sensores		
		fr	l_2,r_2	l_0,r_0
y_{s00}	0.069869	0.075708	0.107307	0.000000
y_{s10}	0.041551	0.014642	0.103043	0.000000
y_{s20}	0.182169	0.186576	0.203626	0.196303
y_{s30}	-0.188934	-0.193240	-0.221292	-0.151724
y_{s40}	0.149396	0.106532	0.000000	0.160072
y_{s50}	-0.031821	-0.074847	0.000000	-0.056022
y_{s60}	-0.079617	0.000000	0.009205	0.070009
y_{s01}	-0.003519	0.035967	0.006154	0.000000
y_{s11}	0.006841	0.100241	-0.053956	0.000000
y_{s21}	-0.005563	-0.002861	0.000085	0.000499
y_{s31}	-0.007534	0.030455	0.020045	0.002083
y_{s41}	-0.011730	0.153950	0.000000	-0.009881
y_{s51}	-0.006720	0.145633	0.000000	-0.000868
y_{s61}	0.368095	0.000000	0.359959	0.368050

e y_{s51}) são usados para estimar a velocidade linear, visando compensar a falha do sensor frontal que têm um peso nulo. No caso de falha simultânea nos sensores l_2 e r_2, o resultado após o processo de treinamento foi a compensação da falha por meio do incremento da contribuição dos sensores l_1 e r_1 (pesos y_{s20} e y_{s30}), os quais têm sinal oposto o que viabiliza a realização de curvas acentuadas. Finalmente, no caso da detecção de falhas nos sensores l_0 e r_0, um pequeno ajuste da contribuição dos sensores l_1 e r_1 é realizado se comparado com a situação sem falha no anel de sensores.

A figura 5.8 mostra as trajetórias seguidas pelo robô móvel após a reconfiguração *online* do controlador neural quando falhas são detectadas e isoladas no anel de sensores. É importante lembrar que o processo de detecção e isolamento das falhas foi emulado desde a interface de usuário. As linhas continuas representam a trajetória seguida pelo robô sob uma condição de operação normal e as linhas tracejadas representam a trajetória seguida após a compensação da falha. Os pontos vermelhos indicam colisões observadas nas trajetórias.

As figuras 5.8a, 5.8b e 5.8c representam as trajetórias obtidas no mesmo ambiente de coleta dos dados de treinamento. Pode-se observar que a trajetória obtida após a detecção de falha no sensor fr apresenta algumas colisões devido à realização de curvas menos fechadas, especificamente nos locais com pouco espaço livre para movimentação do robô. No caso de falha nos sensores l_2 e r_2, trajetórias livres de colisão foram

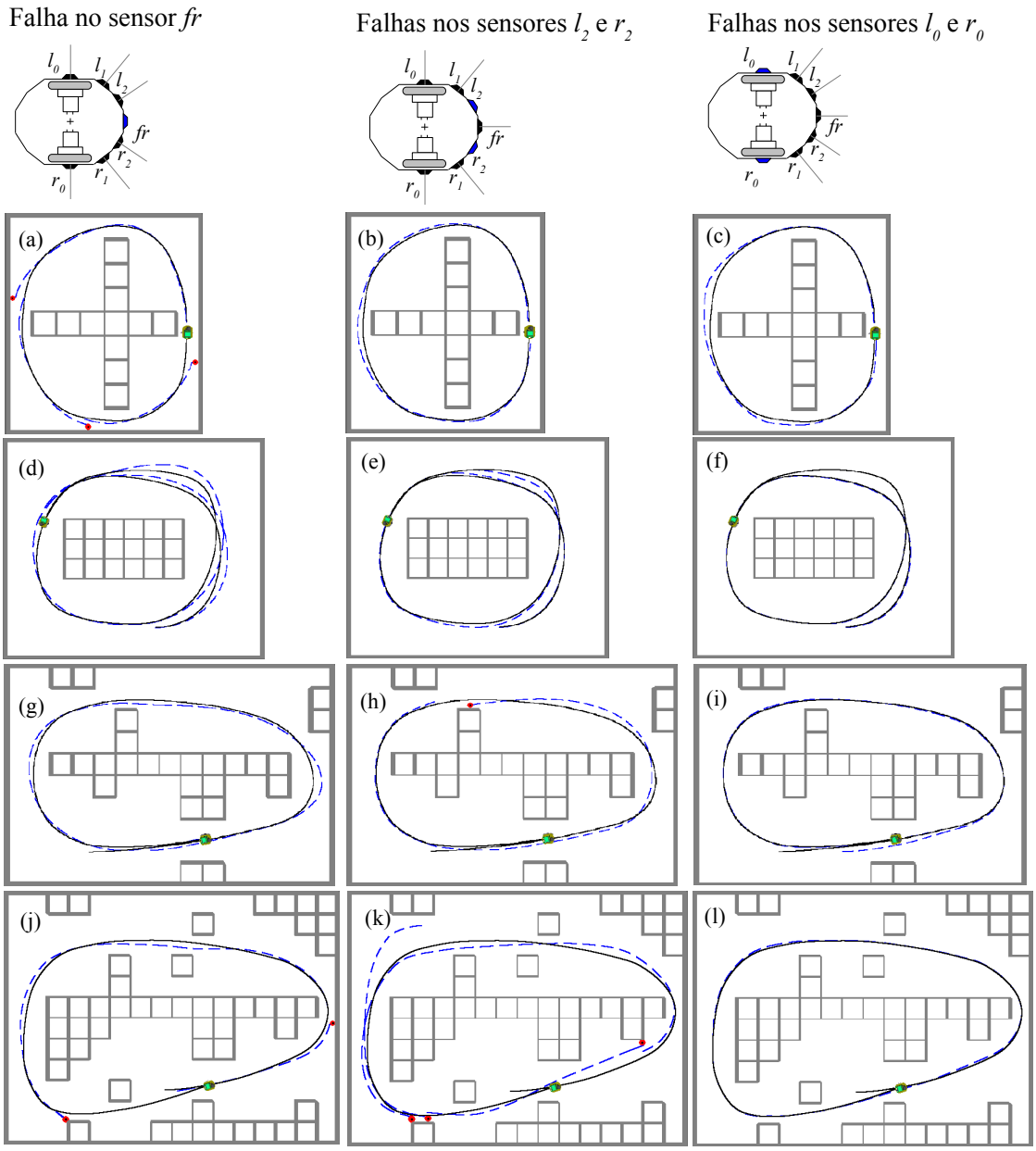


Figura 5.8: Resultados de simulação para o comportamento de evasão de obstáculos com tolerância a falha nos sensores. A linha continua representa a trajetória obtida sem falha no anel de sensores. A linha tracejada é a trajetória obtida após o processo de reconfiguração do controlador neural quando uma ou mais falhas no anel de sensores são detectadas. Os pontos vermelhos indicam colisões observadas nas trajetórias.

obtidas para os primeiros dois ambientes de teste (figuras 5.8b, 5.8e). No entanto, algumas colisões nos ambientes mais complexos foram observadas. Isto é devido a que a compensação feita pelo aumento da contribuição dos sensores l_1 e r_1 não é suficiente para estimar a distância diagonal aos obstáculos próximos ao robô. Finalmente, no caso de falhas nos sensores $l0$ e $r0$, os pesos do controlador neural foram efetivamente ajustados, obtendo trajetórias livres de colisão com os obstáculos.

5.8 COMPARAÇÃO DO TEMPO DE EXECUÇÃO

No intuito de demonstrar a viabilidade da arquitetura HPOPSO-RNA proposta é importante comparar o tempo de execução entre a solução de *hardware* e as soluções de *software* convencionais para robótica móvel. O alto custo computacional dos algoritmos de otimização baseados em inteligência de enxames é a principal desvantagem para aplicações portáteis, nas quais os recursos computacionais são limitados.

O tempo de execução da arquitetura HPOPSO-RNA depende principalmente de três fatores: (a) frequência de operação do circuito f_{op}, (b) número de dados de treinamento NDT e (c) número total de iterações NI. Um valor aproximado do tempo de execução por iteração pode ser calculado usando a equação 5.3. A cada iteração deve ser adicionado o tempo requerido pela unidade de detecção global, que depende do número de partículas (S partículas) e um acréscimo de três ciclos de relógio para a atualização do valor de aptidão mínimo f_{min} e da melhor posição global $\boldsymbol{y}_s$. Adicionalmente, são necessários 5 ciclos de relógio durante a processo de avaliação de mudança do valor de aptidão. Por outro lado, é importante considerar que em algumas iterações aplica-se a abordagem de aprendizado em oposição (OBL). Neste caso, a posição das partículas em cada dimensão é atualizada em três ciclos de relógios (inversão do bit mais significativo).

$$tempo_{iter} \approx \frac{1}{f_{op}}(NDT \cdot ciclosrelogio_{RNA} + N \cdot ciclosrelogio_{particula} + S + 3 + 5) \quad (5.3)$$

Os resultados de simulação usando a ferramenta ModelSim mostram que cada iteração do algoritmo HPOPSO-RNA requer de 106118 ciclos de relógio (vide equação 5.4). Embora os resultados de síntese mostram que a frequência máxima de operação do circuito é aproximadamente de 130MHz, a frequência do relógio utilizada na placa de desenvolvimento é de 100MHz. Desta forma para um total de 5000 iterações o tempo de execução total é de 5.305 segundos.

$$tempo_{iter} \approx \frac{1}{100^6}(808 \cdot 131 + 14 \cdot 18 + 10 + 3 + 5) = \frac{1}{100^6}(106118) = 0.0010612seg \quad (5.4)$$

Estes resultados foram conferidos na implementação física no dispositivo FPGA e medidos no osciloscópio. A figura 5.9 mostra o tempo transcorrido durante o treinamento do controlador neural do robô móvel, desde o envio do sinal *start* até o sinal *ready* mudar de estado, indicando que o processo de treinamento está concluído.

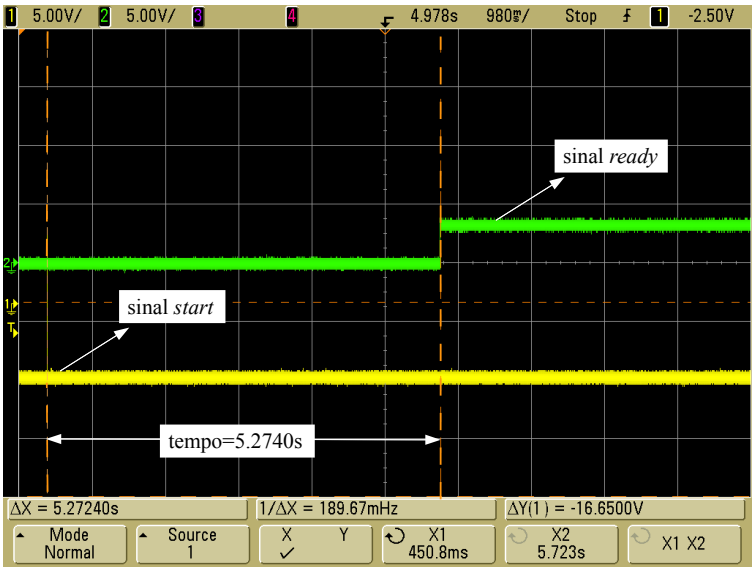


Figura 5.9: Tempo de execução da arquitetura HPOPSO-RNA

Observe-se que o tempo registrado no osciloscópio é levemente inferior ao tempo estimado em simulação. Isto pode ser explicado considerando que a arquitetura HPOPSO proposta aplica a abordagem OBL em algumas iterações, situação em que a atualização das partículas é feita em três ciclos de relógio.

O algoritmo OPSO-RNA com as mesmas condições experimentais foi codificado em linguagem C e implementado em duas soluções de *software*: (a) processador convencional Intel Core 2 Duo, operando a 1.6GHz, 2GB RAM, Windows OS 32 bits; (b) processador RISC Microblaze mapeado no dispositivo FPGA, operando a 100MHz com 64KB de memória de programa. Na primeira solução de *software* calculou-se a média de 10 amostras do tempo de execução em milissegundos, o qual foi obtido usando a função *GetTickCount* no inicio e final do ciclo iterativo do algoritmo OPSO. Para a segunda solução de *software* foi utilizada a arquitetura mostrada na figura 4.27 e o tempo de execução em ciclos de relógio por iteração foi estimado usando um contador de ciclos de relógio.

A tabela 5.5 mostra um quadro comparativo do tempo de execução por iteração entre as soluções de *hardware* e *software*. A primeira solução de *software* (operando a 1.6GHz)

mostrou um tempo de execução total de 19.291 segundos. Portanto, a arquitetura de *hardware* proposta (operando a 100MHz) atinge um fator de aceleramento aproximado de 3.64 vezes. Por outro lado, a implementação *software* do algoritmo OPSO-RNA no MicroBlaze tem um tempo de execução de 6.63 segundos por iteração. Desta forma a arquitetura proposta mapeada no mesmo dispositivo e operando à mesma frequência alcança um fator de aceleramento de 6248.58 vezes.

Tabela 5.5: Comparação do tempo de execução por iteração

Hardware	*Software*	*Software*
HPOPSO-RNA	Intel Core2 Duo	MicroBlaze
FPGA xc5vlx110t	2GB RAM, 1.6GHz	64KB, 100MHz
100MHz	Linguagem C	Linguagem C
1.061 ms	3.858 ms	6.63 s

5.9 DISCUSSÕES FINAIS DO CAPÍTULO E CONTRIBUIÇÕES

Neste capítulo foi apresentada uma arquitetura de *hardware* para o treinamento *online* de um controlador neural de um robô móvel. A arquitetura proposta, chamada de HPOPSO-RNA, está baseada na arquitetura HPOPSO descrita no capítulo anterior, a qual utiliza a técnica de aprendizado em oposição para melhorar a qualidade da solução, assim como para manter a diversidade do enxame.

Os resultados de síntese mostram que há aproximadamente 35% dos recursos disponíveis no dispositivo FPGA para futuras implementações, por exemplo, para a implementação de um microprocessador embarcado onde sejam realizadas operações que não são críticas para o treinamento do controlador neural. É importante salientar que o dispositivo FPGA escolhido não é o dispositivo com maior capacidade de recursos da família Virtex5. Implementações em dispositivos FPGA com maior capacidade de recursos viabiliza o aumento do número de partículas paralelas, assim como a implementação *hardware* que permitam acelerar os métodos de detecção e diagnóstico de falhas nos sensores.

A arquitetura proposta mostrou resultados positivos para o processo de treinamento do controlador neural. Foi possível aproximar os dois comportamentos simulados, encontrando trajetórias próximas às usadas manualmente para a obtenção dos dados de treinamento.

Os resultados obtidos para ambientes desconhecidos permite demonstrar a eficiência do método de treinamento e do modelo neural utilizado para o controlador do robô móvel. Entretanto, com base nos resultados mostrados nas figuras 5.6 e 5.7 é possível concluir que o processo de treinamento apresenta melhores resultados para o primeiro comportamento simulado, ou seja, para percursos no centro da configuração do espaço livre, se comparado com o comportamento para trajetórias próximas às paredes externas. Isto pode ser entendido, levando em consideração que os dados de treinamento coletados para o segundo tipo de comportamento não consideram obstáculos contíguos às paredes externas. Portanto, esta situação representa um distúrbio no sistema para o qual não se tem suficiente informação que permita aproximar o comportamento desejado.

Finalmente, com base nos resultados de tempo de execução é possível concluir que a arquitetura proposta consegue realizar o processo de treinamento em aproximadamente cinco segundos para 5000 iterações do algoritmo. Este resultado melhora significativamente implementações em *software*, porém, o tempo de execução ainda pode ser alto para aplicações embarcadas. No entanto, o uso de mais partículas em paralelo e um critério de parada baseado no erro de aproximação podem contribuir com o decremento do tempo de execução, validando assim a proposta para problemas de otimização embarcada.

A aplicação de arquiteturas paralelas do algoritmo PSO visando acelerar o processo de treinamento de redes neurais artificiais é atualmente objeto de estudo por parte da comunidade acadêmica [120], [161], [168]. Uma das principais contribuições deste trabalho é a aplicação da arquitetura HPOSPO para o treinamento do controlador de um robô móvel baseado em redes neurais *Perceptron*. Na implementação, o processo de treinamento da rede neural é feito *on-chip*, permitindo acelerar a execução dos algoritmos viabilizando assim possíveis aplicações embarcadas. Neste sentido, a arquitetura proposta foi aplicada para explorar a capacidade de ajuste *online* do controlador neural do robô móvel na presença de falhas em um ou mais dos sensores de medição de distância do robô.

Capítulo 6 CONCLUSÕES

Este capítulo resume os resultados e contribuições desta tese, seguidos de uma discussão sobre o encaminhamento dos trabalhos futuros.

6.1 ASPECTOS GERAIS

Neste trabalho foi apresentado um estudo da implementação de arquiteturas paralelas de algoritmos de otimização por inteligência de enxames usando dispositivos reconfiguráveis. O trabalho tem foco em implementações baseadas em dispositivos FPGAs para aplicações embarcadas, de forma que o paralelismo intrínseco dos algoritmos possa ser explorado visando ganhos de desempenho em termos do tempo de execução e qualidade das soluções.

As implementações de *hardware* realizadas estão baseadas em bibliotecas de cálculo aritmético de ponto flutuante, o que permite realizar uma análise de compromisso entre consumo de recursos, desempenho, consumo de energia e precisão e faixa dinâmica das operações matemáticas, visando atingir requisitos específicos de possíveis aplicações na área de sistemas embarcados.

6.2 UNIDADES DE CÁLCULO ARITMÉTICO E TRIGONOMÉTRICO EM PONTO FLUTUANTE

O capítulo 3 apresentou as arquiteturas de *hardware* propostas para os operadores de cálculo aritmético e trigonométrico em ponto flutuante. Foram implementadas em FPGAs unidades de cálculo aritmético de soma, subtração, multiplicação, divisão e raiz quadrada, assim como unidades de cálculo das funções trigonométricas seno, cosseno e arco-tangente e da função exponencial. As implementações foram realizadas para diferentes tamanhos de palavra, incluindo a representação de precisão simples (32 bits) e precisão dupla (64 bits).

Com base nos resultados de síntese lógica é possível concluir que todos os operadores aritméticos e trigonométricos são satisfatoriamente implementados em termos dos recursos disponíveis no dispositivo FPGA. A unidade de soma/subtração $FPadd$ possui um consumo maior de lógica combinacional do que os outros operadores aritméticos, entre os quais a unidade de multiplicação $FPmul$ requer menos recursos, porém faz uso de blocos DSPs embarcados. Por outro lado, a arquitetura $Cordicsincos$ para o cálculo das funções seno e cosseno apresenta um maior consumo de recursos de *hardware* entre os operadores trigonométricos.

Adicionalmente, os resultados de síntese permitiram concluir que as implementações de 27 bits (1 bit de sinal, 8 bits de expoente e 18 bits de mantissa) usam 50% menos blocos DSPs do que as implementações de 32 bits (1 bit de sinal, 8 bits de expoente e 23 bits de mantissa), sendo, portanto, uma solução desejada em casos em que os recursos disponíveis são um fator crítico. É importante salientar que dado que os expoentes possuem o mesmo tamanho, a implementação de 27 bits apresenta uma faixa dinâmica similar à implementação de 32 bits, porém com um incremento no erro associado.

Por outro lado, foi observado que, em geral, a frequência de operação dos operadores é maior para representações com tamanho de palavra menor. A maior frequência de operação foi encontrada para o operador de soma e de cálculo da função *atan*, aproximadamente 164MHz, 160MHz e 130MHz para as representações de 27, 32 e 64 bits, respectivamente. Por outro lado, os operadores que usam multiplicações intermediarias mostram a frequência de operação mais baixa. Concretamente, a unidade de multiplicação em ponto flutuante apresentou uma frequência de 63 MHz para a representação de 64 bits e, portanto, os operadores de cálculo de divisão e raiz quadrada e das funções seno, cosseno e exponencial apresentam frequências de operação similares.

Os resultados de validação mostram que as implementações de 27 bits das unidades $FPadd$ e $FPmul$ possuem um erro quadrático médio (MSE) de 1E-5 e 3E-4, respectivamente, enquanto que as unidades $divNR$ e $sqrtNR$ apresentam um MSE de 2E-9 e 2E-10, respectivamente. Isto é devido aos refinamentos sucessivos realizados pelo algoritmo NR para o cálculo da divisão e raiz quadrada, enquanto a unidade $FPmul$ apresenta erros maiores devido aos processos de truncamento após a multiplicação das mantissas. Contudo, para as aplicações almejadas em sistemas embarcados tais como robótica móvel, treinamento de redes neurais e ajuste de filtros digitais, os resultados apresentados pela implementação de 27 bits são satisfatórios.

Neste contexto, a implementação de 27 bits pode usufruir de uma economia de recursos de *hardware* associado a um erro maior se comparado com as implementações de 32 e 64 bits. Na representação de 32 bits, as unidades $FPadd$ e $FPmul$ apresentam um MSE de 1E-7, aproximadamente. Enquanto para a representação de 64 bits, as mesmas unidades apresentam um MSE de 2E-10 e 8E-13, respectivamente. Adicionalmente, as unidades $divNR$ e $sqrtNR$ mostram valores de MSE menores a 1E-11 para ambas as representações, permitindo concluir que os operadores aritméticos desenvolvidos são adequados para aplicações de sistemas embarcados que envolvem processamento digital de sinais.

O MSE dos operadores trigonométricos $Cordicsincos$ e $Cordicatan$ é em torno de 9E-10 e 1E-9, respectivamente, para uma representação de 32 bits, enquanto que o MSE é de 5E-10 e 9.6E-10, respectivamente, para uma representação de 64 bits. Assim, é possível observar que a diferença no erro associado é pequena entre as implementações de 32 e 64 bits. Isto pode ser explicado considerando os seguintes aspectos: (a) o caráter de refinamento iterativo do algoritmo CORDIC e (b) o valor pré-definido das microrrotações que independe da precisão usada na representação aritmética.

No caso da função exponencial, a unidade $CordicTaylorexp$ que utiliza uma abordagem híbrida *CordicTaylor*, mostrou melhores resultados se comparado com a implementação usando apenas o algoritmo CORDIC. Neste caso, o erro de aproximação depende do argumento de entrada, sendo maior conforme o argumento se torna mais positivo. Para argumentos na faixa de valores [1,5] o MSE é aproximadamente 1E-5, independente do tamanho de palavra, e para argumentos na faixa [5,10] o MSE é aproximadamente 1E-1 para a representação de 27 bits e 7E-5 e 1E-7 para as representações de 32 e 64 bits, respectivamente. Por outro lado, para argumentos de entrada negativos ou positivos pequenos (na faixa [0,1]) o erro de aproximação foi em torno de 1E-10, independente da representação. Desta forma, pode-se concluir que a representação de 27 bits para o cálculo da função exponencial apresenta o melhor fator custo benefício para argumentos de entrada negativos ou pequenos (na faixa [-0.05,0]). Em contraste, as representações de 32 e 64 bits são mais adequadas quando o valor do argumento de entrada é positivo.

6.3 IMPLEMENTAÇÃO DOS ALGORITMOS DE INTELIGÊNCIA DE ENXAMES

O capítulo 4 apresentou as arquiteturas de *hardware* dos algoritmos de otimização por inteligência de enxames. A abordagem de aprendizado em oposição foi aplicada para cada algoritmo visando melhorar o desempenho dos mesmos.

Com base nos resultados experimentais de consumo de recursos, é possível concluir que os algoritmos de otimização por inteligência de enxames podem ser mapeados em FPGAs, viabilizando possíveis aplicações em sistemas de otimização embarcados. Entretanto algumas considerações devem ser levadas em conta: (a) a implementação *hardware* da função custo deve considerar o compromisso entre custo em área lógica e desempenho em termos de tempo de execução. Portanto, arquiteturas paralelas para a implementação da função custo podem ser formuladas mediante uma análise dos recursos disponíveis no dispositivo FPGA; (b) o número de instâncias paralelas da função custo depende do tamanho do enxame, onde para cada partícula se tem uma função custo implementada.

Segundo os resultados de convergência e significância estatística, na maioria dos problemas multimodais estudados, a arquitetura HPOABC apresenta melhores resultados em termos de aproximação ao valor ótimo se comprado com os outros algoritmos. No entanto, deve ser considerado que a arquitetura HPOPSO apresenta o menor consumo de recursos de *hardware* em termos de LUTs e DSPs e a maior frequência de operação entre as arquiteturas implementadas (aproximadamente 130MHz).

Adicionalmente, a técnica de aprendizado em oposição (OBL) mostrou-se eficiente para manter a diversidade do enxame, aprimorando a qualidade da solução obtida. Os resultados de convergência demonstram que as implementações de *hardware* dos algoritmos propostos usando a técnica OBL apresentam uma mediana menor se comparado com as implementações de *hardware* dos algoritmos sem uso do aprendizado em oposição. No entanto, observou-se significância estatística em maior medida para as arquiteturas HPOFA e HPOSFLA. Isto permite concluir que a técnica de aprendizado em oposição contribui em maior medida no aprimoramento da qualidade das soluções obtidas pelas arquiteturas de *hardware* propostas para os algoritmos FA e SFLA.

Finalmente, segundo os resultados de tempo de execução, todas as arquiteturas de *hardware* apresentaram fatores de aceleração do tempo de execução de três ordens

de magnitude se comparadas com as implementações utilizando um microprocessador RISC MicroBlaze mapeado na FPGA e operando à mesma frequência de relógio (100 MHz). Por outro lado, fatores de aceleração desde 1.3 até 15.8 vezes foram obtidos para a comparação do tempo de execução entre as arquiteturas de *hardware* e a implementação em *software* baseada em um computador de escritório usando um processador Intel Core 2 Duo operando a 1.6GHz.

Com base nestes resultados, foi possível concluir que a arquitetura HPOPSO apresenta o menor tempo de execução por iteração para os problemas unimodais, atingindo valores de 2.28 μs e 2.73 μs para os problemas *Esfera* e *Quadric*, respectivamente. Contudo, nestes casos a arquitetura HPOABC mostra o melhor fator de aceleração por iteração (aproximadamente 15.9 vezes) para a função *Esfera* e a arquitetura HPOPSO mostra o melhor fator de aceleração (aproximadamente 5.49 vezes) para a função *Quadric*. Por outro lado, no caso dos problemas multimodais, a arquitetura HPOPSO mostrou o segundo melhor fator de aceleração por iteração (4.65 e 6.05 vezes para as funções *Rosenbrock* e *Rastrigin*, respectivamente), sendo superada apenas pela arquitetura HPOSFLA. No entanto, os resultados de convergência mostraram que a arquitetura HPOSFLA não apresentou resultados de convergência satisfatórios para os problemas multimodais.

Desta forma, e considerando o conjunto de resultados obtidos, é possível concluir que a arquitetura HPOPSO é uma alternativa factível, em termos do fator custo benefício, para a aplicação em problemas de otimização embarcada.

6.4 TREINAMENTO SUPERVISIONADO DO CONTROLADOR NEURAL DE UM ROBÔ MÓVEL

No capítulo 6 a arquitetura HPOPSO foi aplicada no problema de treinamento supervisionado de um controlador neural de um robô móvel de pequeno porte. Nesta aplicação, o processo de treinamento tem como foco a aproximação do comportamento de um controlador humano, assim como o ajuste *online* do controlador quando uma ou mais falhas nos sensores do robô são detectadas e isoladas.

Neste contexto, a arquitetura HPOPSO-RNA proposta é utilizada para acelerar o processo de aprendizado do robô móvel, em que o processo cognitivo do controlador humano representa o modelo comportamental que deve ser aproximado pelo modelo

neural com capacidade de aprendizado baseado no algoritmo O-PSO. Desta forma, o problema envolve a minimização do erro de aproximação entre as saídas obtidas pelo modelo neural e as saídas desejadas.

Os resultados de síntese demonstram que a arquitetura proposta usa aproximadamente 60% dos LUTs disponíveis no dispositivo FPGA, atingindo uma frequência de operação em torno de 130MHz.

Adicionalmente, os testes de validação, realizados por meio de um processo de simulação, demonstraram a eficiência do modelo neural com capacidade de aprendizado *online*. Foi possível aproximar os comportamentos realizados pelo controlador humano. Foram realizados testes em ambientes desconhecidos pelo robô e testes de tolerância a falhas nos sensores, encontrando trajetórias que seguem o comportamento do controlador humano.

Os resultados de tempo de execução mostrar que a arquitetura proposta consegue realizar o processo de treinamento em aproximadamente cinco segundos. O fator de aceleração da arquitetura de *hardware* em comparação com a implementação no processador MicroBlaze foi aproximadamente de quatro ordens de magnitude. Por outro lado, foi observado um fator de aceleração de 3.6 vezes em comparação com o PC de escritório.

Com base nos resultados experimentais de consumo de recursos, qualidade da solução e tempo de execução, é possível concluir a viabilidade da implementação em FPGAs da arquitetura HPOPSO para aplicações em sistemas de otimização embarcados.

6.5 CONTRIBUIÇÕES

As contribuições deste trabalho são um aporte ao estado do conhecimento em computação bioinspirada, especificamente na área de algoritmos por inteligência de enxames. As principais contribuições são listadas junto com as citações das publicações realizadas até a data da elaboração desta tese.

- No capítulo três foram apresentadas as arquiteturas de *hardware* propostas para os operadores de cálculo aritmético e trigonométrico em ponto flutuante. Imple-

mentações usando plataformas reconfiguráveis foram estudadas para os operações aritméticas de soma, subtração, multiplicação, divisão e raiz quadrada [132]. Arquiteturas inéditas baseadas no algoritmo CORDIC foram propostas para o cálculo das funções trigonométricas seno, cosseno e arco-tangente [131], [129]. Adicionalmente, uma arquitetura híbrida baseada no algoritmo CORDIC e na expansão por séries de Taylor foi proposta para o cálculo da função exponencial, visando diminuir o erro associado para pequenos argumentos de entrada [129].

- No projeto das arquiteturas propostas, a precisão foi tratada como uma variável de projeto. Isto permitiu realizar uma análise de compromisso entre o erro associado, o custo em área lógica, frequência de operação, tempo de execução e consumo de potência dos operadores implementados [130]. Foi possível estabelecer que a implementação de 27 bits possibilita uma economia de 50% de multiplicadores embarcados no dispositivo FPGA se comparado com a implementação de precisão simples, ao tempo que atinge um erro de aproximadamente 1E-5, 3E-4, 2E-9, 2E-10 para os operadores de soma, multiplicação, divisão e raiz quadrada, respectivamente, sendo, aceitável para diversas aplicações de sistemas embarcados que envolvem processamento digital de sinais.

- Foi proposta a ferramenta de *software* $vFPUgen$ para geração automática de código VHDL dos operadores aritméticos e trigonométricos usando ponto flutuante. Esta ferramenta usa o padrão IEEE-754 e permite configurar o tamanho de palavra e os parâmetros dos algoritmos, de forma que as arquiteturas implementadas cumpram requisitos específicos de precisão, faixa dinâmica, custo em área lógica e desempenho.

- No capítulo quatro foram apresentadas as arquiteturas de *hardware* dos algoritmos de otimização por inteligência de enxames. Foram implementadas em FPGAs as seguintes abordagens paralelas: (a) otimização por enxame de partículas (HPPSO - *Hardware Parallel Particle Swarm Optimization*) [147], [149]; (b) otimização por colônia de abelhas artificial (HPABC - *Hardware Parallel Artificial Bee Colony Algorithm*) [152]; (c) otimização por colônia de vaga-lumes (HPFA - *Hardware Parallel Firefly Algorithm*) e (d) otimização por embaralhamento de salto aleatório de sapos (HPSFLA - *Hardware Parallel Shuffled Frog Leaping Algorithm*) [150].

- Algumas modificações foram aplicadas nos algoritmos originais ABC (*Artificial Bee Colony Algorithm*) e FA (*Firefly Algorithm*) no intuito de facilitar a implementação *hardware* de forma que o paralelismo intrínseco dos mesmos possa ser

explorado eficientemente. Neste sentido foram propostos os algoritmos GBABC (*Global Bee Artificial Bee Colony*) e GFA (*Global Firefly Algorithm*), os quais fazem uso da informação social contida no indivíduo com o melhor valor de aptidão para melhorar a posição do resto de indivíduos do enxame.

- A metodologia de análise da capacidade de paralelização das arquiteturas propostas também constitui uma contribuição do trabalho. Para isto foi estudado o impacto no custo em área lógica e no tempo de execução do aumento do número de partículas paralelas e da dimensionalidade dos problemas de otimização.

- A implementação em FPGAs dos métodos de adição de diversidade artificial para o algoritmo PSO, são também uma contribuição do trabalho na área otimização por enxames de partículas. Os três métodos implementados em FPGAs foram: (a) *atrativo repulsivo* [148]; (b) *congregação passiva seletiva* [151]; (c) *aprendizado em oposição* [153].

- Adicionalmente, a aplicação da técnica de aprendizado em oposição nos algoritmos ABC, FA e SFLA são aportes inéditos na área de otimização por inteligência de enxames. Neste sentido, foram propostos os algoritmos GBOABC (*Global Bee Opposition based Artificial Bee Colony*), GOFA (*Global Opposition based Firefly Algorithm*) e O-SFLA (*Opposition based Shuffled Frog Leaping Algorithm*). A implementação *hardware* destes algoritmos constitui uma contribuição adicional do trabalho.

- Uma ferramenta de geração automática de código VHDL, chamada de *Swarm Generator*, foi desenvolvida no Matlab no intuito de facilitar a implementação *hardware* dos algoritmos de otimização por inteligência de enxames. Esta ferramenta permite que o projetista selecione o tamanho de palavra da representação em ponto flutuante, o tipo de algoritmo, o número de partículas paralelas, a dimensionalidade do problema de otimização, o tipo de técnica de adição de diversidade (no caso do algoritmo PSO), entre outros parâmetros de projeto. Isto facilita a sua implementação em FPGAs acelerando o tempo de desenvolvimento de um sistema de otimização embarcado.

- No capítulo seis, foi apresentada uma aplicação da arquitetura HPOPSO (*Hardware Parallel Opposition-based Particle Swarm Optimization*) para a solução de um problema de otimização embarcada. Neste sentido, a arquitetura HPOPSO foi utilizada para o treinamento supervisionado de um controlador neural de um robô móvel de pequeno porte. Embora o modelo neural utilizado não seja inédito,

a implementação em FPGAs de um controlador neural com capacidade de aprendizado *online* usando o algoritmo PSO é um aporte original deste trabalho na área de robótica móvel [153].

6.6 TRABALHOS FUTUROS

Embora o sucesso da implementação em FPGAs dos algoritmos por inteligência de enxames para aplicações embarcadas, diversos estudos podem ser realizados visando complementar e melhorar os resultados apresentados neste trabalho.

6.6.1 Estimação do consumo de potência

Uma das principais variáveis de projeto de sistemas embarcados é o consumo de energia. Neste sentido, uma análise de *tradeoff* entre o número de partículas paralelas dos algoritmos e consumo de energia permitirá complementar os dados reportados neste trabalho. Adicionalmente, esta análise também pode ser realizada visando comparar a dissipação de potência entre implementações puramente em *hardware*, implementações de *software* usando processadores embarcados e implementações de co-projeto HS.

6.6.2 Técnicas autoadaptativas para melhoria dos algoritmos

Atualmente, as pesquisas sobre técnicas autoadaptativas que permitam melhorias na qualidade de solução dos algoritmos de inteligência de enxames são um campo aberto na área. O estudo de novos algoritmos e técnicas autoadaptativas, assim como sua implementações em FPGAs, viabiliza o desenvolvimento de arquiteturas de *hardware* flexíveis, de forma que possam ser aplicadas a diferente tipo de aplicações de problemas de otimização embarcada.

6.6.3 Uso de instruções customizadas em processadores embarcados

Um dos maiores problemas da implementação puramente em *hardware* dos algoritmos de inteligência de enxames é o tempo de desenvolvimento da implementação da função

objetivo, a qual depende da aplicação desejada. Embora existam diversas ferramentas de descrição de circuitos em alto nível, o que permite acelerar o tempo de desenvolvimento de projetos de circuitos digitais, ainda é um processo que exige conhecimentos específicos na área de projeto de circuitos digitais.

Neste sentido, uma técnica que pode ser explorada é o uso de multiprocessadores embarcados, por exemplo o MicroBlaze, em que a avaliação da função custo possa ser acelerada mediante o uso de instruções customizadas. Desta forma, as operações aritméticas ou trigonométricas podem ser aceleradas em co-processadores de *hardware*, os quais estão conectados no barramento do processador de *software*. Outro ponto interessante desta abordagem é que a representação aritmética de ponto flutuante pode ser mantida, garantindo a precisão e faixa dinâmica das operações.

6.6.4 Aplicações em otimização embarcada

As implementações realizadas no contexto deste trabalho viabilizam aplicações em diversos campos da engenharia, especificamente na área de otimização embarcada, em que os algoritmos podem ser acelerados visando ganhos no tempo de execução. As possíveis aplicações se concentram em sistemas não lineares e sistemas adaptativos, em que a complexidade dos problemas envolvidos e dos algoritmos de solução demandam o uso de aceleradores de *hardware* ou processadores de alto desempenho.

Com referencia à aplicação de treinamento *online* de um controlador neural de um robô móvel de pequeno porte, descrita neste trabalho, é importante salientar que a coleta de dados de treinamento considerando obstáculos contíguos às paredes externas poderão aprimorar a aproximação dos comportamentos desejados.

Por outro lado, dentre as aplicações futuras podem-se mencionar o desenho de filtros digitais adaptativos, redes de sensores, extração de características, processos de aprendizado de máquinas, síntese de som, planejamento e otimização de trajetória em robótica móvel e de manipuladores, entre outras. É importante destacar que, até a data de elaboração desta tese, algumas destas aplicações são inéditas e estão em fase de desenvolvimento no grupo de pesquisa GRACO-ENM-UnB.

6.6.5 Implementação em GPUs

No intuito de realizar uma comparação de desempenho em termos do tempo de execução e qualidade das soluções, um estudo da implementação em GPUs dos algoritmos e técnicas de adição de diversidade pode ser realizado. Embora esta abordagem não permita a aplicação em sistemas embarcados portáteis, possibilita a exploração do paralelismo intrínseco dos algoritmos por inteligência de enxames para resolver problemas de alta complexidade em termos do número de variáveis de decisão.

Referências Bibliográficas

[1] RAO, S. *Engineering Optimization, Theory and Practice*. USA: John Wiley & Sons, 1996.

[2] Sean Luke. *Essentials of Metaheuristics*. [S.l.]: Lulu, 2009. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[3] KENNEDY, J.; EBERHART, R. *Swarm Intelligence*. [S.l.]: Morgan Kaufmann, 2001.

[4] EIBEN, A.; SMITH, J. *Introduction to Evolutionary Computing*. Germany: Springer, 2003.

[5] FLOREANO, D.; MATTIUSSI, C. *Bio-Inspired Artificial Intelligence: Theories, Methods, and Technologies*. [S.l.]: MIT Press, 2008.

[6] POLI, R.; KENNEDY, J.; BLACKWELL, T. Particle swarm optimization. *Swarm Intelligence*, v. 1, n. 1, p. 33–57, 2007.

[7] BANKS, A.; VINCENT, J.; ANYAKOHA, C. A review of particle swarm optimization. part I: background and development. *Int. J. Natural Computing*, v. 6, n. 4, p. 467–484, 2007.

[8] BANKS, A.; VINCENT, J.; ANYAKOHA, C. A review of particle swarm optimization. part ii: hybridisation, combinatorial, multicriteria and constrained optimization, and indicative applications. *Int. J. Natural Computing*, v. 7, n. 1, p. 109–124, 2008.

[9] HARTENSTEIN, R. Why we need reconfigurable computing education. In: *Proc. International Workshop on Reconfigurable Computing Education*. Karlsruhe, Germany: IEEE, 2006. p. Invited Keynote.

[10] SASS, R.; SCHMIDT, A. *Embedded Systems Design with Platform FPGAs Principles and Practices*. [S.l.]: Morgan Kaufmann, 2010.

[11] WOF, W. *High-Performance Embedded Computing, Architectures, Applications, and Methodologies*. [S.l.]: Morgan Kaufmann, 2007.

[12] GAJSKI, D. et al. *Embedded System Design: Modeling, Synthesis and Verification*. New York, NY, USA: Springer, 2009.

[13] BOYD, S. *Real time embedded convex optimization, International Symposium on Mathematical Programming*. Aug. 2009. Online. Disponível em: <http://ismp2009.eecs.northwestern.edu/Plenaries/>.

[14] MATTINGLEY, J.; BOYD, S. Real-time convex optimization in signal processing. *IEEE Signal Processing Magazine*, p. 50–61, May 2010.

[15] CALAZANS, N. *Projeto Lógico Automatizado de Sistemas Digitais Sequênciais*. [S.l.]: Imprinta Gráfica e Editora Ltda, 1998.

[16] MENDES, R. *Population Topologies and Their Influence in PSO*. Tese (Doutorado) — Universidade do Minho, 2004. Portugal.

[17] MICHALEWICZ, Z. Heuristic methods for evolutionary computation techniques. *Journal of Heuristics*, v. 2, n. 1, p. 177–206, 1995.

[18] HOLLAND, J. *Adaptation in Natural and Artificial Systems*. USA: University of Michigan Press, 1975.

[19] KIRKPATRICK, S.; GELATT, C.; VECCHI, M. Optimization by simulated annealing. *Science*, v. 220, n. 4598, p. 671–680, 1983.

[20] BONABEAU, E.; DORIGO, M.; THERAULAZ, G. *Swarm intelligence: from natural to artificial systems*. New York, USA: Oxford University Press, 1999.

[21] KENNEDY, J.; EBERHART, R. Particle swarm optimization. In: *Proc. International Conference on Neural Networks*. Perth, Australia: IEEE, 1995. p. 1942–1948.

[22] YANG, S. *Nature-Inspired Metaheuristic Algorithms*. Cambridge, UK: Luniver Press, 2010.

[23] MICHALEWICZ, Z. Evolutionary computation techniques for nonlinear programming problems. *International Trans. in Operational Research*, v. 1, n. 2, p. 223–240, 1994.

[24] KOZA, J. *Genetic programming: on the programming of computers by means of natural selection*. USA: MIT Press, 1992.

[25] CASTRO, L. de. *Fundamentals of natural computing: basic concepts, algorithms, and applications*. USA: CRC Press, 2006.

[26] DREO, J. et al. Adaptive learning search, a new tool to help comprehending metaheuristics. *International Journal on Artificial Intelligence Tools*, v. 16, n. 3, p. 1–23, 2007.

[27] SERAPIÃO, A. Fundamentos de otimização por inteligência de enxames: uma visão geral. *Revista Controle & Automação*, v. 20, n. 3, p. 271–304, 2009.

[28] HACKWOOD, S.; WANG, J. The engineering of cellular robotic system. In: *Proc. International Symposium on Intelligent Control*. Arlington, USA: IEEE, 1988. p. 70 – 75.

[29] HACKWOOD, S.; BENI, G. Self-organization of sensors for swarm intelligence. In: *Proc. International Conference on Robotics and Automation*. Nice , France: IEEE, 1992. p. 819 – 829.

[30] BERGH, F. van den. *An Analysis of Particle Swarm Optimizers*. Tese (Doutorado) — Department of Computer Science, University of Pretoria, 2001. South Africa.

[31] MOORE, G. The microprocessor: engine of the technology revolution. *Communications of the association for computing machinery ACM*, v. 40, n. 2, p. 112–114, 1997.

[32] SMITH, M. *Application-Specific Integrated Circuits*. USA: Addison-Wesley, 1997.

[33] WU, K.; TSAI, Y. Structured ASIC, evolution or revolution. In: *Proc. International Symposium on Physical Design*. Phoenix, USA: [s.n.], 2004.

[34] HUTTON, M. et al. A methodology for FPGA to structured-ASIC synthesis and verification. In: *Proc. International Conference of Design, Automation and Test in Europe*. Munich, Germany: IEEE, 2006. p. 64–69.

[35] HAMBLEN, J.; FURMAN, M. *Rapid Prototyping of Digital Systems*. Boston, USA: Kluver Academin Publishers, 2001.

[36] HARTENSTEIN, R. The digital divide of computing. In: *Proc. Conference On Computing Frontiers*. Ischia, Italy: ACM, 2004. p. 357 – 362.

[37] WULF, W.; MCKEE, S. Hitting the memory wall implications of the obvious. *Computer Architecture News*, v. 23, p. 20–24, 1994.

[38] MINGJIE, L. et al. Performance benefits of monolithically stacked 3D FPGA. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, v. 26, n. 2, p. 216–229, 2006.

[39] DEHON, A. The density advantage of configurable computing. *Computer*, v. 30, n. 4, p. 41–49, 2000.

[40] KILTS, S. *Advanced FPGA Design: Architecture, Implementation and Optimization*. New Jersey, USA: John Wiley & Sons, 2007.

[41] MEYER-BAESE, U. *Digital Signal Processing with Field Programmable Gate Arrays*. Tallahassee, USA: Springer, 2004.

[42] HUEBNER, M.; BECKER, T.; BECKER, J. Real-time LUT-based network topologies for dynamic and partial FPGA self-reconfiguration. In: *Proc. International Symposium on Integrated Circuits and System Design*. Porto de Galinhas, Brazil: ACM, 2004. p. 28–32.

[43] TAGHAVI, T. et al. Innovate or perish: FPGA physical design. In: *Proc. International Symposium on Physical Design*. [S.l.: s.n.], 2004. p. 148–155. Phoenix, United States.

[44] LIN, Y.; LI, F.; HE, L. Routing track duplication with fine-grained power-gating for FPGA interconnect power reduction. In: *Proc. Asia and South Pacific Design Automation Conference*. Shanghai, China: ACM, 2005. p. 645–650.

[45] PAULSSON, K.; HUBNER, M.; BECKER, J. Cost-and power optimized FPGA based system integration: Methodologies and integration of a low-power capacity-based measurement application on Xilinx FPGAs. In: *Proc. Design, Automation and Test in Europe Conference*. Munich, Germany: IEEE, 2008. p. 50–55.

[46] SASS, R. et al. Reconfigurable computing cluster (RCC) project: investigating the feasibility of FPGA-based petascale computing. In: *Proc. Symposium on Field-Programmable Custom Computing Machines*. Napa, USA: IEEE, 2007. p. 127–140.

[47] EBERHART, R.; KENNEDY, J. A new optimizer using particle swarm theory. In: *Proc. International Symposium Micro Machine and Human Science*. Nagoya, Japan: IEEE, 1995. p. 39–43.

[48] REYNOLDS, C. Flocks, herds and schools: A distributed behavioral model. *Computer Graphics*, v. 21, n. 4, p. 25–34, 1987.

[49] HEPPNER, F.; GRENANDER, U. A stochastic nonlinear model for coordinated bird flocks. *The Ubiquity of Chaos AAAS publication*, 1990.

[50] REEVES, W. Particle systems - a technique for modeling a class of fuzzy objects. *ACM Trans. on Graphics*, v. 2, n. 2, p. 91–108, 1983.

[51] KENNEDY, J.; EBERHART, R. A discrete binary version of the particle swarm algorithm. In: *Proc. International Conference on Systems, Man, and Cybernetics*. Orlando, USA: IEEE, 1997. p. 4104 – 4108.

[52] BERGH, F. van den; ENGELBRECHT, A. Training product unit networks using cooperative particle swarm optimisers. In: *Proc. International Conference on Neural Networks*. Washington DC, USA: IEEE, 2001. p. 126 – 131.

[53] SHI, Y.; EBERHART, R. A modified particle swarm optimizer. In: *Proc. Congress on Computational Intelligence*. Anchorage, Alaska, USA: IEEE, 1998. p. 69–73.

[54] CLERC, M. The swarm and the queen: Towards a deterministic and adaptive particle swarm optimization. In: *Proc. Congress on Evolutionary Computation*. Washington DC, USA: IEEE, 1999. p. 1951–1957.

[55] SHI, Y.; EBERHART, R. Empirical study of particle swarm optimization. In: *Proc. Congress on Evolutionary Computation,*. Washington DC, USA: IEEE, 1999. p. 1945–1950.

[56] CLERC, M.; KENNEDY, J. The particle swarm - explosion, stability, and convergence in a multi-dimensional complex space. *IEEE Trans. on Evolutionary Computation*, v. 6, n. 1, p. 58–73, 2002.

[57] BERGH, F. van den; ENGELBRECH, A. A new locally convergent particle swarm optimizer. In: *Proc. International Conference on Systems Man, and Cybernetics*. Hammmamet, Tunisia: IEEE, 2002. p. 96–101.

[58] BERGH, F. van den; ENGELBRECH, A. A cooperative approach to particle swarm optimization. *Trans. IEEE Evolutionary Computation*, v. 8, n. 3, p. 225–239, 2004.

[59] BRITS, R.; ENGELBRECH, A.; BERGH, F. van den. Locating multiple optima using particle swarm optimization. *Applied Mathematics and Computation*, v. 189, n. 2, p. 1859–1883, 2007.

[60] KENNEDY, J. Small worlds and megaminds: Effects of neighborhood topology on particle swarm performance. In: *Proc. International Congress on Evolutionary Computation*. Washington DC, USA: IEEE, 1999. p. 1931–1938.

[61] KENNEDY, J.; MENDES, R. Population structure and particle swarm performance. In: *Proc. Congress on Evolutionary Computation*. Honolulu, Hawaii: IEEE, 2002. p. 1671 – 1676.

[62] ANGELINE, P. Evolutionary optimization versus particle swarm optimization: philosophy and performance differences. In: *Proc. International Conference on Evolutionary Programming VII*. San Diego, USA: LNCS 1447, 1998. p. 601–610.

[63] LOVBJERG, M.; RASMUSSEN, T.; KRINK, T. Hybrid particle swarm optimizer with breeding and subpopulations. In: *Proc. Conference on Genetic and Evolutionary Computation GECCO*. San Fransisco, USA: ACM, 2001. p. 469–476.

[64] MAEDA, Y.; KURATANI, T. Simultaneous perturbation particle swarm optimization. In: *Proc. Congress on Evolutionary Computation*. Vancouver, Canada: IEEE, 2006. p. 672–676.

[65] AFSHINMANESH, F.; MARANDI, A.; RAHIMI-KIAN, A. A novel binary particle swarm optimization method using artificial immune system. In: *Proc. Conference on Computer as a Tool EUROCON*. Belgrado, Serbia e Montenegro: IEEE, 2005. p. 217–220.

[66] ZHANG, W.; XIE, X. Hybrid particle swarm with differential evolution operator. In: *Proc. International Conference on Systems, Man and Cybernetics*. Washington DC, USA: IEEE, 2003. p. 3816–3821.

[67] MONSON, C.; SEPPI, K. The kalman swarm: A new approach to particlemotion in swarm optimization. In: *Proc. Genetic and Evolutionary Computation Conference*. Seattle, USA: LNCS 3103, 2004. p. 140–150.

[68] SHI, Y.; EBERHART, R. Fuzzy adaptive particle swarm optimization. In: *Proc. Congress on Evolutionary Computation*. Seoul, Korea: IEEE, 2001. p. 101–106.

[69] KARABOGA, D. *An Idea based on honey bee swarm for numerical optimization*. [S.l.], 2005. Reporte técnico.

[70] TERESHKO, V.; LOENGAROV, A. Collective decision-making in honey bee foraging dynamics. *Journal of Computing and Information Systems*, v. 9, n. 3, p. 1–7, 2005.

[71] CAZAMINE, S.; SNEYD, J. A model of collective nectar source selection by honey bees. *Journal of Theoretical Biology*, v. 149, n. 4, p. 547–571, 1991.

[72] TERESHKO, V. Reaction-diffusion model of a honeybee colony foraging behaviour. *Parallel Problem Solving from Nature VI. Lecture Notes in Computer Science.*, v. 1917, p. 807–816, 200.

[73] KARABOGA, D.; BASTURK, B. A powerful and efficient algorithm for numerical function optimization ABC. *Journal of Global Optimization*, v. 39, p. 459–471, 2007.

[74] YANG, S. Firefly algorithms for multimodal optimization. *Lecture Notes on Computers Sciences: Stochastic Algorithms: Foundations and Applications*, v. 5792, p. 169–178, 2009.

[75] EUSUFF, M.; LANSEY, K. Optimization of water distribution network design using the shuffled frog leaping algorithm. *Journal of Water Resources Planning & Management*, v. 129, n. 3, p. 210–225, 2003.

[76] MOSCATO, P. *On evolution search optimization genetic algorithms and martial arts towards memetic algorithms*. [S.l.], 1989. Reporte técnico.

[77] EUSUFF, M.; LANSEY, K.; PASHA, F. Shuffled frog leaping algorithm: a memetic meta-heuristic for discrete optimization. *Journal of Engenieering Optimization*, v. 38, n. 2, p. 129–154, 2006.

[78] DAWKINS, R. *The Selfish Gene*. Oxford, UK: Oxford University Press, 1976.

[79] MOSCATO, P.; COTTA, C. Una introducción a los algoritmos meméticos. *Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial*, n. 19, p. 131–148, 2003.

[80] ELBELTAGI, E.; HEGAZY, T.; GRIERSON, D. Comparison among five evolutionary-based optimization algorithms. *Journal of Advanced Engineering Informatics*, v. 19, n. 1, p. 43–53, 2005.

[81] LUO, X.; YANG, Y. Solving TSP with shuffled frog-leaping algorithm. In: *Proc. Conferece on Intelligent Systems Design and Applications*. Kaohsiung, Taiwan: IEEE, 2008. p. 228–232.

[82] ELBEHAIRY, H.; ELBELTAGI, E.; HEGAZY, T. Comparison of two evolutionary algorithms for optimization of bridge deck repairs. *Journal of Computer-Aided Civil and Infrastructure Engineering*, v. 21, p. 561–572, 2006.

[83] HUYNH, T. A modified shuffled frog leaping algorithm for optimal tuning of multivariable PID controllers. In: *Proc. Conference on Industrial Techcnology*. Chengdu, China: IEEE, 2008. p. 1–6.

[84] HUYNH, T.; NGUYEN, D. Fuzzy controller design using a new shuffled frog leaping algorithm. In: *Proc. Conference on Industrial Technology*. Gipssland, Australia: IEEE, 2009. p. 1–6.

[85] SUN, X.; WANG, Z.; ZHANG, D. A web document classification method based on shuffled frog leaping algorithm. In: *Proc. International Conference on Genetic and Evolutionary Computing*. Jingzhou, China: IEEE, 2008. p. 205–208.

[86] AMIRI, B.; FATHIAN, M.; MAROOSI, A. Application of shuffled frog-leaping algorithm on clustering. *Journal of Advanced Manufacturing Technology*, v. 45, p. 199–209, 2009.

[87] BHADURI, A. A clonal selection based shuffled frog leaping algorithm. In: *Proc. Conference on Advance Computing*. Patiala, India: IEEE, 2009. p. 125–130.

[88] ZHIJIN, Z.; KEQIANG, Y.; ZHIDONG, Z. Discrete shuffled frog leaping algorithm for multi-user detection in DS-CDMA communication system. In: *Proc. Conference on Communication Technology*. Hangzhou, China: IEEE, 2008. p. 421–424.

[89] BLACKWELL, T.; BENTLEY, P. Don't push me! collision-avoiding swarms. In: *Proc. Congress on Evolutionary Computation*. Honolulu, USA: IEEE, 2002. p. 1691–1696.

[90] XIE, X.; ZHANG, W.; YANG, Z. A dissipative particle swarm optimization. In: *Proc. Congress on Evolutionary Computation*. Honolulu, Hawaii: IEEE, 2002. p. 1456–1461.

[91] KRINK, T.; VESTERSTROM, J.; RIGET, J. Particle swarm optimization with spatial particle extension. In: *Proc. Congress on Evolutionary Computation*. Honolulu, Hawaii: IEEE, 2002. p. 1474–1479.

[92] MONSON, C.; SEPPI, K. Adaptive diversity in PSO. In: *Proc. Conference on Genetic and Evolutionary Computation GECCO*. Seatle, USA: ACM, 2006. p. 59–66.

[93] LOVBJERG, M.; KRINK, T. Extending particle swarms with self-organized criticality. In: *Proc. Congress on Evolutionary Computation*. Honolulu, Hawaii: IEEE, 2002. p. 1588–1593.

[94] RIGET, J.; VESTERSTROM, J. *A diversity-guided particle swarm optimizer - the ARPSO*. [S.l.], 2002. Reporte técnico.

[95] PARRISH, J.; HAMNER, W. *Animal Groups in Three Dimensions*. Cambridge, UK: Cambridge University Press, 1997.

[96] HE, S. et al. A particle swarm optimizer with passive congregation. *Bio-Systems Iternational Journal*, v. 78, p. 135–147, 2004.

[97] VOIT, J. *Otimização por exame de partículas com congregação passiva seletiva*. Tese (Doutorado) — Universidade Federal do Rio de Janeiro, Brasil, 2010.

[98] TIZHOOSH, H. Opposition-based learning a new scheme for machine intelligence. In: *Proc. Int. Conference on Computational Intelligence for Modelling, Control and Automation*. Vienna, Austria: [s.n.], 2005. p. 695–701.

[99] ALQUNAIEER, F.; TIZHOOSH, H.; RAHNAMAYAN, S. Opposition based computing a survey. In: *Proc. Int. Joint Conference on Neural Networks*. Barcelona, Spain: [s.n.], 2010. p. 1098–7576.

[100] RAHNAMAYAN, S.; TIZHOOSH, H.; SALAMA, M. Opposition versus randomness in soft computing techniques. *Journal Applied Soft Computing*, v. 8, p. 906–918, 2008.

[101] MALISIA, A.; H.R., T. Applying opposition-based ideas to the ant colony system. In: *Proc. IEEE Swarm Intelligence Symposium*. Honolulu, HI: [s.n.], 2007. p. 182–189.

[102] JABEEN, H.; JALIL, Z.; BAIG, A. Opposition based initialization in particle swarm optimization (O-PSO). In: *Proc. ACM Conference on Genetic and Evolutionary Computation*. Montreal, Canada: [s.n.], 2009. p. 2047–2052.

[103] SCHUTTE, J. et al. Parallel global optimization with the particle swarm algorithm. *Journal for Numerical Methods in Engineering*, v. 6, n. 13, p. 2296–2315, 2003.

[104] PARSOPOULOS, K.; TASOULIS, D.; VRAHATIS, M. Multiobjective optimization using parallel vector evaluated particle swarm optimization. In: *Proc. International Conference Artificial Intelligence and Applications*. Innsbruck, Austria: IASTED, 2004. p. 823–828.

[105] VENTER, G.; SOBIESZCZANSKI, J. A parallel PSO algorithm accelerated by asynchronous evaluations. In: *Proc. Structural and Multidisciplinary Optimization World Congress*. Rio de Janeiro, Brazil: AIAA, 2005. p. 123–137.

[106] KOH, B. et al. Parallel asynchronous particle swarm optimization. *Journal for Numerical Methods in Engineering*, v. 67, n. 4, p. 578–595, 2006.

[107] WAINTRAUB, M. *Algoritmos paralelos de otimização por enxame de partículas em problemas nucleares*. Tese (Doutorado) — Universidade Federal do Rio de Janeiro, Brasil, 2009.

[108] NVIDIA. *GPU Programming Guide G80*. December 2008. Disponível em: <http://www.nvidia.com>.

[109] CHE, S. et al. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, v. 68, n. 10, p. 1370–1380, 2008.

[110] VERONESE, L.; KROHLING, R. Swarms's flight: accelerating the particles using C-CUDA. In: *Proc. International Congress on Evolutionary Computation*. Trondheim, Norway: IEEE, 2009. p. 3264–3270.

[111] ZHOU, Y.; TAN, Y. GPU-based parallel particle swarm optimization. In: *Proc. International Congress on Evolutionary Computation*. Trondheim, Norway: IEEE, 2009. p. 1493–1500.

[112] MUSSI, L.; S., C.; DAOLIO, F. GPU-based road sign detection using particle swarm optimization. In: *Proc. IEEE. Conference on Intelligent Systems Design and Applications*. Pisa, Italy: IEEE, 2009. p. 152–157.

[113] REYNOLDS, P. et al. FPGA implementation of particle swarm optimization for inversion of large neural networks. In: *Proc. Swarm Intelligence Symposium*. Pasadena, CA, USA: IEEE, 2005. p. 389–392.

[114] KÓKAI, G.; CHRIST, T.; FRHAUF, H. Using hardware-based particle swarm method for dynamic optimization of adaptive array antennas. In: *Proc. International Conference on Adaptive Hardware and Systems*. Istanbul, Turkey: NASA/ESA, 2006. p. 51–58.

[115] PENA, J.; UPEGUI, A.; SÁNCHEZ, E. Particle swarm optimization with discrete recombination: an online optimizer for evolvable hardware. In: *Proc. International Conference Adaptive Hardware and Systems*. Istanbul, Turkey: NASA/ESA, 2006. p. 163–170.

[116] MAEDA, Y.; MATSUSHITA, N. Simultaneous perturbation particle swarm optimization using FPGA. In: *Proc. International Conference on Neural Networks*. Orlando, Florida, USA: IEEE, 2007. p. 2695–2700.

[117] PALANGPOUR, P.; VENAYAGAMOORTHY, G.; SMITH, S. Particle swarm optimization: A hardware implementation. In: *Proc. International Conference on Computer Design*. Las Vegas, USA: CSREA Press, 2009. p. 134–139.

[118] FARMAHINI-FARAHANI, A.; FAKHRAIE, S.; SAFARI, S. SOPC-Based architecture for discrete particle swarm optimization. In: *Proc. International Conference on Electronics, Circuits and Systems*. Marrakech, Morocco: IEEE, 2007. p. 1003–1006.

[119] FARMAHINI-FARAHANI, A. et al. Parallel scalable hardware implementation of asynchronous discrete particle swarm optimization. *Engineering Applications of Artificial Intelligence*, v. 23, p. 177–187, 2010.

[120] LIN, C.; TSAI, H. FPGA implementation of a wavelet neural network with particle swarm optimization learning. *Trans. on Mathematical and Computer Modeling*, v. 47, n. 9, p. 982–996, 2008.

[121] GAO, Z. et al. FPGA implementation of adaptive IIR filters with particle swarm optimization algorithm. In: *Proc. International Conference on Communication Systems*. Singapore: IEEE, 2008. p. 1364–1367.

[122] MEHMOOD, S. et al. Hardware-oriented adaptation of a particle swarm optimization algorithm for object detection. In: *Proc. Euromicro International Conference on Digital System Design*. Parma, Italy: IEEE, 2008. p. 904–911.

[123] CHOWDHURY, S.; CHAKRABARTI, D.; SAHA, H. Medical diagnosis using adaptive perceptive particle swarm optimization and its hardware realization using field programmable gate array. *Journal of Medical Systems*, v. 33, n. 6, p. 447–465, 2009.

[124] TEWOLDE, G.; HANNA, D.; HASKELL, R. Hardware PSO for sensor network applications. In: *Proc. Swarm Intelligence Symposium*. St. Louis, USA: IEEE, 2008. p. 1–8.

[125] TEWOLDE, G. *Performance Enhancement and Hardware Implementation of Particle Swarm Optimization*. Tese (Doutorado) — Oakland University, USA, 2008.

[126] TEWOLDE, G.; HANNA, D.; HASKELL, R. Accelerating the performance of particle swarm optimization for embedded applications. In: *Proc. International Congress on Evolutionary Computation*. Trondheim, Norway: IEEE, 2009. p. 2294–2300.

[127] TEWOLDE, G.; HANNA, D.; HASKELL, R. Multi-swarm parallel PSO hardware implementation. In: *Proc. Swarm Intelligence Symposium*. Nashville, USA: IEEE, 2009. p. 60–66.

[128] AVCI, G. *Implementation of Artificial Bee Colony (ABC) Algorithm on FPGA for Real-Time Applications*. Dissertação (Mestrado) — Nigde University, 2011. Turkey.

[129] MUÑOZ, D. et al. FPGA-based floating-point library for CORDIC algorithms. In: *Proc. International Southern Programmable Logic Conference*. Porto de Galinhas, Brazil: IEEE, 2010. p. 55–60.

[130] MUÑOZ, D. et al. Tradeoff of FPGA design of a floating-point library for arithmetic operators. *International Journal of Integrated Circuits and Systems*, v. 5, n. 1, p. 42–52, 2010.

[131] MUÑOZ, D. et al. Tradeoff of FPGA design of floating-point transcendental functions. In: *Proc. International Conference on Very Large Scale Integration*. Florianopolis, Brazil: IEEE, 2009. p. 1–4.

[132] SÁNCHEZ, D. et al. Parameterizable floating-point library for arithmetic operations in FPGAs. In: *Proc. International Symposium on Integrated Circuits and System Design*. Natal, Brazil: ACM, 2009. p. 253–258.

[133] IEEE. *IEEE standard for binary floating-point arithmetic*. [S.l.], 1985. Reporte técnico.

[134] WANG, X. *Variable Precision Floating-point Divide and Square Root for Efficient FPGA Implementation of Image and Signal Processing Algorithms*. Tese (Doutorado) — Northeastern University, USA, 2007.

[135] MARKSTEIN, P. Software division and square root using Goldschmidt's algorithms. In: *Proc. Conference on Real Nambers ans Computers*. Dagstuhl, Germany: RNC, 2004. p. 146–157.

[136] SÁNCHEZ, D. *Implementação em VHDL de uma biblioteca parametrizável de operadores aritméticos em ponto flutuante para ser usada em problemas de robótica*. Dissertação (Mestrado) — Universidade de Brasília, Brasília, Brazil, 2009.

[137] SODERQUIST P.; LEESER, M. Division and square root choosing the right implementation. *IEEE Micro*, v. 17, n. 4, p. 56–66, 1997.

[138] SMITH, R. A continued-fraction analysis of trigonometric argument reduction. *IEEE Trans. on Computers*, v. 44, n. 11, p. 1348–1351, 1995.

[139] BOUDABOUS, A. et al. Implementation of hyperbolic functions using CORDIC algorithm. In: *Proc. International Conference on Microelectronics*. Tunis, Tunisia: IEEE, 2004. p. 738–741.

[140] YUDI, J. et al. FPGA-based image processing for omnidirecctional vision on mobile robots. In: *Proc. International Symposium on Integrated Circuits and System Design*. João Pessoa, Brazil: [s.n.], 2011. p. 113–118.

[141] YUDI, J. et al. An FPGA-based omnidirectional vision sensor for motion detection on mobile robots. *International Journal of Reconfigurable Computing*, v. 1, p. 1–16, 2012.

[142] MENTOR GRAPHICS. *ModelSim*. 2010. Disponível em: <http://www.mentor.com/>.

[143] PARHAMI, B. *Computer Arithmetic: Algorithms and Hardware Designs*. New York, USA: Oxford University Press., 2007.

[144] XILINX. *MicroBlaze processor reference guide*. [S.l.], Outubro 2009. Datasheet.

[145] GRIFFITH, R.; PANG, F. *MicroBlaze system performance tuning*. [S.l.], 2008. Reporte técnico.

[146] GOVINDU, G.; SCROFANO, R.; PRASANA, V. A library of parameterizable floating-point cores for FPGAs and their application to scientific computing. In: *Proc. International Conference Engineering Reconfigurable Systems and Algorithms*. Las Vegas, USA: IEEE, 2005. p. 137–148.

[147] MUÑOZ, D. et al. Hardware architecture for parallel particle swarm optimization using floating-point arithmetic. In: *Proc. International Conference on Intelligent Systems Design and Applications*. Pisa, Italy: IEEE, 2009. p. 243–248.

[148] MUÑOZ, D. et al. Hardware particle swarm optimization based on the attractive-repulsive scheme for embedded applications. In: *Proc. Int. Conf. on Reconfigurable Computing and FPGAs*. Cancún, México: IEEE, 2010. p. 55–60.

[149] MUÑOZ, D. M. et al. Comparison between two FPGA implementations of the particle swarm optimization algorithm for high performance embedded applications. In: *Proc. Int. Conf. on Bio-inspired Computing, Theories and Applications*. Liverpool, UK.: [s.n.], 2010. p. 1637–1645.

[150] MUÑOZ, D. M. et al. Accelerating the shuffled frog leaping algorithm by parallel implementations in FPGAs. In: *Proc. Int. Conf. on Bio-inspired Computing, Theories and Applications*. Liverpool, UK.: [s.n.], 2010. p. 1526–1534.

[151] MUÑOZ, D. M. et al. Hardware particle swarm optimization with passive congregation for embedded applications. In: *Proc. Int. Southern Programmable Logic Conf.* Córdoba, Argentina: [s.n.], 2011. p. 173–178.

[152] MUÑOZ, D. M. et al. Accelerating the artificial bee colony algorithm by hardware parallel implementations. In: *Proc. Latinoamerican Conf. on Circuits and Systems*. Playa del Carmen, México: IEEE, 2012. p. 1–4.

[153] MUÑOZ, D. M. et al. Hardware opposition-based PSO applied to mobile robot controllers. *Submitted to Engineering Applications of Artificial Intelligence*, 2012.

[154] THOMAS, D.; LUK, W. Resource efficient generators for the floating-point uniform and exponential distributions. In: *Proc. International Conference on Application-specific Systems, Architecture and Processors*. Leuven, Belgium: IEEE, 2008. p. 102–107.

[155] XILINX CORP. *Efficient shift registers, LFSR counters and long pseudo random sequence generators*. July 1996. Disponível em: <http://www.xilinx.com/>.

[156] TANG, K. et al. Benchmark functions for the CEC2010 special session and competition on large-scale global optimization. In: *World Congress on Computational Intelligence*. Barcelona, Spain: IEEE, 2010. p. 1–23.

[157] XILINX CORP. *Xpower Tutorial: FPGA Design*. 2009. Disponível em: <ftp://ftp.xilinx.com/pub/documentation/tutorials>.

[158] FARMAHINI-FARAHANI, A.; FAKHRAIE, S.; SAFARI, S. Scalable architecture for on-chip neural network training using swarm intelligence. In: *Design Automation and Test in Europe Conference*. Munich, Germany: IEEE, 2008. p. 1340–1345.

[159] PENA, J.; UPEGUI, A. A population-oriented architecture for particle swarms. In: *Proc. International Conference Adaptive Hardware and System*. Edinburgh, Scotland: NASA/ESA, 2007. p. 563–571.

[160] TEWOLDE, G.; HANNA, D.; HASKELL, R. A modular and efficient hardware architecture for particle swarm optimization algorithm. *Microprocessors and Microsystems*, v. 36, p. 289–302, 2012.

[161] CAVUSLU, M.; KARAKUZU, C.; KARAKAYA, F. Neural identification of dynamic system on FPGA with improved PSO learning. *Applied Soft Computing*, v. 12, p. 2707–2718, 2012.

[162] GUPTA, L.; MEHRA, R. Modified PSO based IRR filter design for system identification on FPGA. *International Journal of Computer Applications*, v. 22, n. 5, p. 1–7, 2011.

[163] ROUMELIOTIS, S.; SUKHATME, G.; BEKEY, G. Sensor fault detection and identification in a mobile robot. In: *Proc. Conference on Intelligent Robots and Systems*. Victoria, BC, Canada: IEEE, 1998. p. 1383–1388.

[164] FLOREANO, D.; MONDADA, F. Evolution of homing navigation in a real mobile robot. *Trans. IEEE Systems, Man and Cybernetics - Part B*, v. 26, n. 3, p. 396–407, 1996.

[165] PUGH, J.; MARTINOLI, A. Multi-robot learning with particle swarm optimization. In: *Proc. Conference on Autonomous Agents and Multiagent Systems*. Hakodate, Japan: ACM, 2006. p. 441–448.

[166] PUGH, J.; MARTINOLI, A. Parallel learning in heterogeneous multi-robot swarms. In: *Proc. IEEE Conf. On Evolutionary Computation*. Singapore: [s.n.], 2007. p. 3839–3845.

[167] BRÄULN, T. *Embedded Robotics*. [S.l.]: Springer-Verlag, 2006. Germany.

[168] VASUMATHI, B.; MOORTHI, S. Implementation of hybrid ANN-PSO algorithm on FPGA for harmonic estimation. *Engineering Applications of Artificial Intelligence*, v. 25, n. 3, p. 476–483, 2012.

APÊNDICES

Apêndice A PUBLICAÇÕES REALIZADAS

A.1 TRABALHOS PUBLICADOS

A.1.1 Operadores aritméticos e trigonométricos em ponto flutuante

- Sánchez, D.; Muñoz, D.M.; Llanos, C.; Ayala-Rincón, M. Parameterizable Floating-point Library for Arithmetic Operations in FPGAs. In: *Proc. International Symposium on Integrated Circuits and System Design*. Natal, Brazil: ACM, 2009, p. 253-258

- Muñoz, D.M.; Sánchez, D.; Llanos, C.; Ayala-Rincón, M. Tradeoff of FPGA Design of Floating-point Transcendental Functions. In: Proc. *International Conference on Very Large Scale Integration*. Florianópolis, Brazil: IEEE, 2009, p.1-4

- Muñoz, D.M.; Sánchez, D.; Llanos, C.; Ayala-Rincón, M. FPGA-based Floating-point Library for CORDIC Algorithms. In: *Proc. International Southern Programmable Logic Conference*. Porto de Galinhas, Brazil: IEEE, 2010, p. 55-60

- Muñoz, D.M.; Sánchez, D.; Llanos, C.; Ayala-Rincón, M. Tradeoff of FPGA Design of a Floating-point Library for Arithmetic Operators. In: *International Journal of Integrated Circuits and Systems*, SBC, vol. 5, n. 1, p. 42-52, 2010.

A.1.2 Algoritmos PSO, SFLA e ABC

- Muñoz, D.M.; Llanos, C.; Coelho, L. S.; Ayala-Rincón, M. Hardware Architecture for Parallel Particle Swarm Optimization using Floating-point Arithmetic. In: *Proc. International Conference on Intelligent Systems Design and Applications*. Pisa, Italy: IEEE, 2009, p. 243-248.

- Muñoz, D.M.; Llanos, C.; Coelho, L. S.; Ayala-Rincón, M. Comparison Between two FPGA Implementations of the Particle Swarm Optimization Algorithm for High-performance Embedded Applications. In: *Proc. International Conference*

on Bio-inspired Computing: Theories and Applications. Liverpool, UK: IEEE, 2010, p. 1637-1645.

- Muñoz, D.M.; Llanos, C.; Coelho, L. S.; Ayala-Rincón, M. Accelerating the Shuffled Frog Leaping Algorithm by Parallel Implementations in FPGAs. In: *Proc. International Conference on Bio-inspired Computing: Theories and Applications*. Liverpool, UK: IEEE, 2010, p. 1526-1534.

- Muñoz, D.M.; Llanos, C.; Coelho, L. S.; Ayala-Rincón, M. Hardware Particle Swarm Optimization based on the Attractive-Repulsive scheme for Embedded Applications. In: *Proc. International Conference on Reconfigurable Computing and FPGAs*. Cancún, México: IEEE, 2010, p. 55-60.

- Muñoz, D.M.; Llanos, C.; Coelho, L. S.; Ayala-Rincón, M. Hardware Particle Swarm Optimization with Passive Congregation for Embedded Applications. In: *Proc. International Southern Programmable Logic Conference*. Córdoba, Argentina: IEEE, 2011, p. 173-178.

- Muñoz, D.M.; Llanos, C.; Coelho, L. S.; Ayala-Rincón, M. Opposition-based Shuffled PSO with passive congregation applied to FM Matching Synthesis. In: *Proc. International Congress on Evolutionary Computation*. New Orleans, USA: IEEE, 2011, p. 2775-2781.

- Muñoz, D.M.; Llanos, C.; Coelho, L. S.; Ayala-Rincón, M. Accelerating the Artificial Bee Colony Algorithm by Hardware Parallel Implementations. In: *Proc. Latin American Symposium on Circuits and Systems*. Playa del Carmen, México: IEEE, 2012, p. 1-4.

A.2 TRABALHOS SUBMETIDOS EM JOURNAL INTERNACIONAL

- Muñoz, D.M.; Llanos, C.; Coelho, L. S.; Ayala-Rincón, M. Hardware Opposition-based PSO Applied to Mobile Robot Controllers. Submetido a: *International Journal of Engineering Applications of Artificial Intelligence*. Science Direct, 2012.

A.3 PUBLICAÇÕES RELACIONADAS

- Yudi, J.; Arias-Garcia, J.; Sánchez-Ferreira, C.; Muñoz, D.M.; Llanos, C.; Motta J.M.S.T. An FPGA-based omnidirectional vision sensor for motion detection on

mobile robots. In: *Proc. International Journal of Reconfigurable Computing.* Hindawi, v. 2012, 2012.

- Sánchez, D.; Muñoz, D.M.; Llanos, C.; Motta J.M.S.T. Reconfigurable system approach to the direct kinematics of a 5 d.o.f robotic manipulator. In: *Proc. International Journal of Reconfigurable Computing.* Hindawi, v. 2010, 2010.

- Yudi, J.; Muñoz, D.M.; Arias-Garcia, J.; Llanos, C.; Motta J.M.S.T. FPGA-based image processing for omnidirecctional vision on mobile robots. In: *Proc. International Symposium on Integrated Circuits and System Design.* João Pessoa, Brazil: ACM, 2011, p.113-118.

- Yudi, J.; Sánchez-Ferreira, C.; Muñoz, D.M.; Llanos, C.; Berger, P. An unified approach for convolution-based image filtering on reconfigurable systems. In: *Proc. International Southern Programmable Logic Conference.* Córdoba, Argentina: IEEE, 2011, p. 63-68.

- Sánchez, D.; Muñoz, D.M.; Llanos, C.; Motta J.M.S.T. FPGA Implementation for Direct Kinematics of a Spherical Robot Manipulator. In: *Proc. International Conference on Reconfigurable Computing and FPGAs.* Quintana Roo, México: IEEE, 2009, p. 416-421.

Apêndice B ESTUDO DE CONVERGÊNCIA DOS MÉTODOS DE DIVERSIDADE ARTIFICIAL

Este apêndice apresenta um estudo de convergência dos métodos de adição de diversidade artificial estudados neste trabalho. Para isto, foram feitas as implementações em *software* do algoritmo PSO e suas respectivas variantes usando as técnicas: (a) PSO atrativo-repulsivo (arPSO) [94]; (b) PSO com congregação passiva seletiva (PSOpc) [97]; (c) PSO com aprendizado em oposição (O-PSO) [98], [102].

A implementação foi realizada no Matlab, usando um processador IntelCore Duo, operando a 1.6GHz, 2 GB RAM, Windows XP OS. As validações foram feitas usando os problemas *benchmark Esfera*, *Quadric*, *Rosenbrock* e *Rastrigin* para casos de 6, 10 e 14 dimensões.

Na seção 2.7 são explicados os mecanismos de funcionamento de cada técnica de adição de diversidade artificial. A tabela B.1 mostra as condições experimentais usadas para os testes de validação.

Tabela B.1: Condições experimentais para os algoritmos PSO, arPSO, PSOpc e O-PSO

Algoritmo	Parâmetro	Valor
parâmetros comuns a todos os algoritmos	Tamanho do enxame	8
	Dimensionalidade	6, 10 e 14
	Número de iterações	2000
	Peso de inercia	[0.8,0.1]
	Coeficientes cognitivo e social	c_1=c_2=2.1
	Velocidade máxima	[-6.0,6.0]
	Espaço de busca	[-8.0,8.0]
	Valor *threshold*	0.01
arPSO	valor inicial *dlow*	0.1
	valor inicial *dhigh*	1.0
PSOpc	Coeficiente de congregação passiva	c_1=0.5
O-PSO	Max. iterações sem mudança de aptidão	100

A tabela B.2 apresenta os resultados de convergência obtidos após 32 execuções de cada algoritmo para cada função de teste nos casos de 6, 10 e 14 dimensões. A tabela mostra o valor médio, mediana, desvio padrão e o valor de aptidão mínimo entre todos os experimentos. Adicionalmente, o melhor resultado para cada problema está destacado em cor cinza.

Tabela B.2: Resultados de convergência dos métodos de adição de diversidade artificial

Problema	Algoritmo *software*	Média	Mediana	Mínimo	Desvio Padrão	Número acertos
	PSO	6.10E-43	1.85E-63	9.96E-80	3.45E-42	32/32
$Esfera\ 6D$	arPSO	3.13E-49	1.80E-64	2.10E-83	1.77E-38	32/32
	PSOpc	3.46E-34	4.60E-39	6.02E-49	1.54E-33	32/32
	O-PSO	1.67E-51	9.35-67	3.65E-84	9.41E-51	32/32
	PSO	3.07E-19	1.36E-25	1.73E-32	1.64E-18	32/32
$Quadric\ 6D$	arPSO	3.16E-17	1.14E-24	5.60E-33	1.79E-16	32/32
	PSOpc	7.70E-13	1.64E-14	2.44E-18	1.81E-12	32/32
	O-PSO	1.65E-23	1.05E-28	3.76E-34	5.41E-23	32/32
	PSO	5.1877	0.7926	0.0106	8.6239	18/32
$Rosenbrock\ 6D$	arPSO	2.3898	0.5705	0.0176	4.7255	19/32
	PSOpc	2.5319	0.7257	0.0812	4.3876	18/32
	O-PSO	2.4132	0.7229	0.0050	4.3484	17/32
	PSO	1.9277	1.9899	0.0	1.4057	3/32
$Rastrigin\ 6D$	arPSO	1.7471	1.9899	0.0	1.0627	3/32
	PSOpc	1.7146	1.0541	0.0	2.5194	7/32
	O-PSO	1.4313	0.9950	0.0	0.9784	6/32
	PSO	3.75E-23	1.74E-28	6.56E-42	1.20E-22	32/32
$Esfera\ 10D$	arPSO	4.46E-23	5.46E-29	2.90E-45	2.44E-22	32/32
	PSOpc	3.12E-15	1.04E-17	3.02E-21	1.49E-14	32/32
	O-PSO	2.57E-18	2.32-30	1.56E-40	1.45E-17	32/32
	PSO	1.01E-6	1.85E-8	2.63E-11	3.38E-6	32/32
$Quadric\ 10D$	arPSO	8.99E-7	9.59E-9	3.75E-11	3.77E-6	32/32
	PSOpc	3.22E-3	9.93E-4	1.04E-5	5.40E-3	29/32
	O-PSO	1.16E-7	6.27E-9	1.67E-11	4.94E-7	32/32
	PSO	8.8509	7.8759	0.3273	7.2969	4/32
$Rosenbrock\ 10D$	arPSO	10.6894	8.1658	0.1507	10.2721	3/32
	PSOpc	11.5607	8.8821	2.1727	8.6307	0/32
	O-PSO	10.8511	7.7333	0.1749	8.7853	4/32
	PSO	6.6228	6.4672	1.9899	3.1827	0/32
$Rastrigin\ 10D$	arPSO	6.1253	5.9698	0.9950	2.8266	0/32
	PSOpc	14.4312	11.3229	0.9950	11.8218	0/32
	O-PSO	5.2867	4.9748	1.9899	2.2391	0/32
	PSO	1.70E-11	1.26E-16	4.31E-23	9.62E-11	32/32
$Esfera\ 14D$	arPSO	2.06E-13	1.14E-16	2.55E-24	7.74E-13	32/32
	PSOpc	1.44E-7	8.07E-9	6.13E-11	5.37E-7	32/32
	O-PSO	4.37E-15	1.23-19	2.79E-25	1.91E-14	32/32
	PSO	1.91E-3	1.17E-3	2.04E-5	2.15E-3	32/32
$Quadric\ 14D$	arPSO	3.73E-3	1.65E-3	1.05E-4	4.79E-3	28/32
	PSOpc	0.3497	0.3066	0.0394	0.2425	0/32
	O-PSO	1.10E-3	6.09E-4	3.94E-5	1.32E-3	32/32
	PSO	20.3936	16.0468	1.0678	13.3227	0/32
$Rosenbrock\ 14D$	arPSO	23.5118	20.4522	1.4999	14.2798	0/32
	PSOpc	25.1141	20.0982	3.9432	22.5923	0/32
	O-PSO	19.6847	18.2349	0.5163	12.0620	1/32
	PSO	13.9972	13.9294	3.9798	6.6240	0/32
$Rastrigin\ 14D$	arPSO	12.5826	12.9349	2.9849	5.1410	0/32
	PSOpc	30.9038	24.7600	4.7202	20.2927	0/32
	O-PSO	13.1556	11.9395	5.9783	5.6230	0/32

Observa-se que todos os algoritmos apresentam resultados próximos do valor ótimo para os problemas de otimização unimodal. O algoritmo O-PSO atinge melhores resultados para o caso de maior dimensionalidade. Por outro lado, o algoritmo O-PSO apresenta os melhores valores de convergência (valores da mediana pequenos) para os problemas multimodais, especificamente para o caso da função *Rastrigin*.

É importante salientar que ajustes dos parâmetros dos algoritmos podem melhorar a qualidade dos resultados. No entanto, nos testes realizados não foi realizado nenhum processo de sintonização dos parâmetros e apenas valores convencionais encontrados na literatura foram usados.

Com base nestes resultados e considerando a facilidade de implementação da técnica de aprendizado em oposição, é possível concluir que o algoritmo O-PSO é um bom candidato para ser implementado em *hardware*, possibilitando arquiteturas com menor consumo de recursos de *hardware* e soluções de maior qualidade se comparado com o algoritmo PSO básico.