JONAS GOMES FILHO

APLICAÇÃO DE TÉCNICAS DE RECONFIGURAÇÃO DINÂMICA A PROJETO DE MÁQUINA DE VETOR SUPORTE (SVM)

São Paulo

2010

JONAS GOMES FILHO

APLICAÇÃO DE TÉCNICAS DE RECONFIGURAÇÃO DINÂMICA A PROJETO DE MÁQUINA DE VETOR SUPORTE (SVM)

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para a obtenção do Título de Mestre em Engenharia Elétrica.

São Paulo

2010

JONAS GOMES FILHO

APLICAÇÃO DE TÉCNICAS DE RECONFIGURAÇÃO DINÂMICA A PROJETO DE MÁQUINA DE VETOR SUPORTE (SVM)

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para a obtenção do Título de Mestre em Engenharia Elétrica.

Área de concentração:
Microeletrônica

Orientador:

Prof. Dr. Wang Jiang Chau

São Paulo

2010

Este exemplar foi revisado e alterado em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, 05 de março de 2010.

Assinatura do autor ____________________________

Assinatura do orientador ________________________

FICHA CATALOGRÁFICA

Gomes Filho, Jonas
Aplicação de técnicas de reconfiguração dinâmica a máquina de vetor suporte / J. Gomes Filho. -- ed.rev. --São Paulo, 2010.
p. 93

Dissertação (Mestrado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Sistemas Eletrônicos.

1. Arquitetura reconfigurável 2. Circuitos FPGA 3. Inteligência artificial 4. Microeletrônica I. Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Sistemas Eletrônicos II. t.

DEDICATÓRIA

Dedico este trabalho aos meus pais, minha irmã e à Leila.

Vocês tornam tudo possível.

AGRADECIMENTOS

Ao prof. Dr. Wang Jiang Chau, um excelente orientador, que com muita paciência me mostrou o caminho correto a seguir, transmitindo segurança e muito conhecimento para o desenvolvimento deste trabalho.

Aos meus professores do IFSP, Ricardo Pires, Alexandre Brincalepe Campo, Omar Alves e Luiz Carlos Moreira, por terem acreditado no meu potencial e me incentivado a cursar o mestrado.

Ao Dr. Marius Strum e aos demais colegas do GSEIS, em especial a Edgar Leonardo Romero, que muito me ajudou com sua experiência, Raul Acosta, que além de amigo foi um professor, e cujo trabalho foi fundamental para o desenvolvimento da minha pesquisa. Também a Joel Muñoz, que não mediu esforços para me ajudar com este trabalho.

Ao meu amigo Mário Raffo, que ingressou comigo no mestrado e me acompanhou em todos os passos, inclusive nas matérias cursadas e no tema da pesquisa. Esta amizade rendeu muitos frutos para este trabalho.

A meus pais, que me apoiaram em cada decisão e ajudaram a traçar o caminho que chegou a esse mestrado. Tudo que consegui na minha vida devo a eles.

A minha namorada Leila, que acompanhou lado a lado por todos estes anos e me fez crescer como pessoa. Me apoiou durante todo o mestrado e foi muito compreensiva. É minha constante inspiração.

Aos meus amigos, Felipe Gambier Santos Souza, José Antonio de Melo Neto e Raniel Victor Machado, que também me deram seu apoio e de alguma forma me ajudaram com meu trabalho.

Ao PNM – CNPq, pelo apoio financeiro para a realização deste trabalho.

RESUMO

As Máquinas de Vetores de Suporte (SVMs) têm sido largamente empregadas em diversas aplicações, graças à sua baixa taxa de erros na fase de testes (boa capacidade de generalização) e o fato de não dependerem das condições iniciais. Dos algoritmos desenvolvidos para o treinamento da SVM, o Sequential Minimal Optimization (SMO) é um dos mais rápidos e eficientes para a execução desta tarefa.

Importantes implementações da fase de treinamento da SVM têm sido feitas em FPGAs. A maioria destas implementações tem sérias restrições na quantidade de conjunto de amostras a serem treinadas, pelo fato de implementarem soluções numéricas. De observação na literatura técnica, apenas dois trabalhos implementaram o SMO para o treinamento SVM em hardware e apenas um destes possibilita o treinamento de uma quantidade importante de amostras, porém a aplicação é restrita a apenas um *benchmark* específico.

Na última década, com a tecnologia baseada em RAM estática, os FPGAs apresentaram um novo aspecto de flexibilidade: a capacidade de reconfiguração dinâmica, que possibilita a alteração do sistema em tempo de execução trazendo redução de área. Adicionalmente, apesar de uma potencial penalidade no tempo de processamento, a velocidade de execução continua muito superior quando comparada com soluções em software.

No presente trabalho, uma solução genérica é proposta para o treinamento SVM em hardware (i.e. uma arquitetura que possibilite o treinamento para diversos tipos de amostras de entrada), e, motivado pela natureza seqüencial do algoritmo SMO, uma arquitetura dinamicamente reconfigurável é desenvolvida.

Um estudo da implementação genérica com codificação em ponto fixo é apresentada, assim como os efeitos de quantização. A arquitetura é implementada no dispositivo Xilinx Virtex-IV XC4VLX25. Dados de tempo e área são obtidos e detalhes da síntese são explorados. É feita uma simulação da reconfiguração dinâmica através de chaves de isolação para a validação do sistema sob reconfiguração dinâmica. A arquitetura foi testada para três diferentes *benchmarks*,

com resultados indicando que o treinamento no hardware reconfigurável foi acelerado em até 30 vezes quando comparado com a solução em software e os estudos apontaram que uma economia de até 22,38% de área útil do FPGA pode ser obtida dependendo das metodologias de síntese e implementação adotadas.

ABSTRACT

Support Vector Machines have been largely used in different applications, due to their high classifying capability without errors (generalization capability) and the advantage of not depending on the initial conditions. Among the developed algorithms for the SVM training, the Sequential Minimal Optimization (SMO) is one of the fastest and the one of the most efficient algorithms for executing this task.

Important dedicated hardware implementations of the training phase of the SVM have been proposed for digital FPGA. Most of them are very restricted about the quantity of input samples to be trained due to the fact that they implement numeric solutions. Only two works with implementation in the SMO algorithm for the SVM training in hardware have been reported recently, and just one is able to train an important quantity of input samples, however it is restricted for only one specific *benchmark*.

In the last decade, with the technology based on static memory (SRAM), FPGAs has provided a unique aspect of flexibility: the capability of dynamic reconfiguration, which involves altering the programmed design at run-time and allows area's saving. In addition, although leading to some time penalty, the execution time is still faster when compared with purely software solutions.

In this work we present a totally hardware general-purpose implementation of the SMO algorithm. In this general-purpose approach, training of examples with different number of samples and elements are possible, and, motivated by the sequential nature of some of the SMO tasks, a dynamically reconfigurable architecture is developed.

A study of the general-purpose implementation with fixed-point codification is presented, as well as the quantization effects. The architecture is implemented in the Xilinx Virtex-IV XC4VLX25 device, and timing and area data are provided. Synthesis details are exploited. A simulation using dynamic circuit switching is carried out in order to validate the system’s dynamic reconfiguration aspects. The architecture was tested in the training of three different *benchmarks*; the training on the reconfigurable hardware was accelerated up to 30 times when compared with software solution, and

studies points to an area saving up to 22.38% depending on the synthesis and implementation methodologies adopted in the project.

.

SUMÁRIO

LISTA DE ILUSTRAÇÕES..3
LISTA DE TABELAS..5
LISTA DE ABREVIATURAS E SIGLAS..6
1. INTRODUÇÃO..1
1.1. Contexto..1
1.2. Motivação..4
1.3. Justificativas..6
1.4. Objetivos..6
1.5. Organização da Dissertação..7
2. ASPECTOS TEÓRICOS..8
2.1. Reconfiguração em FPGAs..8
2.1.1. Tipos de Reconfiguração..8
2.1.2. Arquiteturas de Sistemas Dinamicamente Reconfiguráveis.........12
2.2. Support Vector Machine (Ou Máquina de Vetor de Suporte)..........17
2.2.1. Definição Matemática e Algorítmica da SVM..............................18
2.2.2. Condições Karush-Kuhn-Tucker..24
2.2.3. O algoritmo SMO, *Sequential Minimal Optimization*....................25
3. TRABALHOS CORRELATOS..32
3.1. Implementações em Hardware da Fase de Teste SVM..................32
3.2. Implementações em Hardware da Fase de Treinamento SVM.......33
3.3. Proposta de metodologia para um SDR aplicado a SVM................36
4. METODOLOGIA DE PROJETO..38
4.1. Arquitetura básica do algoritmo SMO..38

4.2. Aspectos de Implementação da função de kernel 41
4.3. Definição de uma solução SVM genérica. 43
5. Implementação Plana 46
5.1. Análise dos dados em ponto fixo 46
5.2. A implementação da exponenciação 50
5.3. Fluxo de Projeto 53
5.4. Simulações e análise de tempo 55
6. Proposta de arquitetura e implementação para o SDR 58
6.1. Proposta de particionamento para a reconfiguração dinâmica 58
6.2. Análise da Viabilidade da Arquitetura Reconfigurável 61
6.3. Metodologia de Síntese 65
6.4. Metodologia de Simulação por DCS. 70
6.5. Simulação DCS aplicada a arquitetura SVM. 73
6.6. Resultados da Simulação DCS 76
7. CONCLUSÃO DO TRABALHO 79
7.1. Contribuições 79
7.2. Conclusões 81
7.3. Trabalhos Futuros 82
REFERÊNCIAS 84
APÊNDICE A - Algoritmo SMO 90

LISTA DE ILUSTRAÇÕES

Figura 1 - Utilização de FPGAs X ASICs 2

Figura 2 - Particionamento temporal em projetos de DRFPGAs. 4

Figura 3 - Reconfiguração Dinâmica Parcial 9

Figura 4 - Reconfiguração Estática (a) X Reconfiguração Dinâmica (b) 10

Figura 5 - Arquitetura do Virtex-II 12

Figura 6 - Bloco lógico configurável. 13

Figura 7 - Slice do Virtex-II. 14

Figura 8 - Estrutura de bus-macros em um DRFPGA 15

Figura 9 - Bus-macros extensos x curtos 16

Figura 10 - Bus-macros aninhadas 17

Figura 11 - Mapeamento não linear em SVM 19

Figura 12 - Hiperplano ótimo 20

Figura 13 - Representação dos possíveis valores de αp e αq para $yp \neq yq$. 27

Figura 14 - Representação dos possíveis valores de αp e αq para $yp = yq$. 28

Figura 15 - Arquitetura VLSI da implementação MP-SMO 35

Figura 16 - Arquitetura modular. 40

Figura 17 - Algoritmo para serialização do cálculo do *kernel* 44

Figura 18 - *Ten-fold cross validation*. 49

Figura 19 - Arquitetura de *kernel* proposta por Anguita et al. 51

Figura 20 - Arquitetura de *kernel* modificada. 51

Figura 21 - Resultados de exponenciação para 8 (a), 16 (b) e 24 (c) bits e iterações no intervalo $0 \leq F \leq 0{,}3$. 52

Figura 22 - Estrutura de arquivos adotada no BSV. 54

Figura 23 - Uma iteração da arquitetura plana. 59

Figura 24 - Uma iteração da arquitetura reconfigurável. 60

Figura 25 - Partições no DRFPGA ao longo do tempo. 60

Figura 26 - Penalidade de tempo para $0 \leq tr \leq 502\mu s$. 64

Figura 27 - Etapas da Metodologia Early Access 65

Figura 28 - Roteamento dos módulos reconfiguráveis: kernel (a) e otimizador (b) ...68
Figura 29 - Roteamento dos módulos estáticos. 69
Figura 30 - Exemplo de simulação DCS. 70
Figura 31 - Simulação DCS para portas lógicas E/OU. 72
Figura 32 - Formas de onda da simulação DCS. 73
Figura 33 - Máquina de estados da unidade de controle. 74
Figura 34 - Configuração das chaves de isolação..75
Figura 35 - Simulação DCS para arquitetura do SDR. 76
Figura 36 - Detalhe dos estados 2 e 3 da unidade de controle. 77

LISTA DE TABELAS

Tabela 1 - Erro de teste e número de iterações para um conjunto de benchmarks. .48

Tabela 2 - Número de *Slices* ocupados no FPGA de acordo com a precisão do *kernel*. 53

Tabela 3 - Número de ciclos necessários para computar uma iteração SMO. 56

Tabela 4 - Tempo total de treinamento na arquitetura plana. 56

Tabela 5 - Dados de área ocupada no FPGA 58

Tabela 6 - Tempo total de treinamento na arquitetura reconfigurável. 64

Tabela 7 - Tabela da verdade das chaves de isolação. 71

Tabela 8 - Eventos da máquina de estado da unidade de controle 75

Tabela 9 - Valores da memória α depois da 359ª iteração. 78

LISTA DE ABREVIATURAS E SIGLAS

ASIC	Application Specific Integrated Circuit
BSV	Bluespec System Verilog
CLB	Configurable Logic Block
DCM	Digital Clock Manager
DCS	Dynamic Circuit Switching
FPGA	Field Programmable Gate Array
GPP	General-Purpose Processors
GSEIS	Grupo Projeto de Sistemas Eletrônicos Integrados e Software Aplicado
IOB	Input Output Block
KKT	Karush-Kuhn-Tucker
LUT	Look-up Table
PRR	Partial Reconfigurable Region
RAM	Random Access Memory
RTL	Register Transfer Level
RTR	Run-Time Reconfiguration
SDR	Sistema Dinamicamente Reconfigurável
SMO	Sequential Minimal Optimization
SoC	System on Chip
SPR	Sistema Parcialmente Reconfigurável
SRAM	Static RAM
SVM	Support Vector Machine
VHDL	Very High Speed Integrated Circuit Hardware Description Language

1. INTRODUÇÃO

O propósito deste capítulo é apresentar argumentos que justifiquem a utilização de dispositivos de *hardware* com propriedades de reconfiguração parcial para o projeto e implementação de sistemas dinamicamente reconfiguráveis. O presente trabalho terá como interesse de aplicação, um algoritmo de classificação binária de dados conhecido como Máquina de Vetor de Suporte (*Support Vector Machine*, SVM). Baseando-se neste contexto serão apresentadas as justificativas, as motivações e os objetivos deste trabalho.

1.1. Contexto

Até a década de 1990, o cenário para implementação de algoritmos de uso específico estava polarizada em dois tipos de dispositivos: os processadores de propósito geral (*general-purpose processors*, GPPs) e os circuitos integrados de aplicação específica (*application specific integrated circuits*, ASICs). O primeiro, baseado na arquitetura Von Neumann, possui grande flexibilidade de aplicação, além de permitir edições e modificações de projeto. Porém devido à natureza seqüencial e às características multiuso dos GPPs, estes apresentam desempenho computacional muito abaixo das soluções em hardware dedicado. Os ASICs por outro lado, apresentam alta eficiência computacional, porém pouca flexibilidade e nenhum tipo de alteração em campo. Com o advento dos arranjos de portas programáveis em campo (*field programmable gate arrays*, FPGAs), encontrou-se uma solução de compromisso, obtendo parte da flexibilidade dos GPPs e parte do alto desempenho computacional típico dos ASICs.

A principal característica que distingue os FPGAs de outras soluções é a sua configurabilidade. Existem FPGAs baseados em tecnologia *antifuse* que são configuráveis apenas uma única vez. Todavia, o desenvolvimento de FPGAs com

tecnologia EPROM e SRAM tornou possível a reconfiguração[1] destes (i.e., um número ilimitado de configurações em um mesmo dispositivo), bastando para isso carregar um novo *bitstream* na memória de configuração interna.

As soluções com processadores de uso geral continuam sendo a alternativa mais flexível, pois admitem novas aplicações e modificações de projeto através de alterações no software, porém sob penalidade de desempenho e maior consumo comparado aos ASICs. Para tarefas específicas em que o desempenho do hardware é uma questão crucial, os ASICs ainda são a solução mais recomendada, entretanto, o projeto com ASICs deve ser justificado por um alto volume de produção para compensar os investimentos iniciais do projeto e dos altos custos de fabricação. Os FPGAs, por outro lado, tendem a possuir um ciclo de projeto menor, possibilitando alterações do projeto, com custo menor, nas fases mais avançadas e com o desempenho relativo aumentando e aproximando-se cada vez mais dos ASICs. Por estes motivos, os FPGAs vêm tomando cada vez mais espaço de mercado dos ASICs, como se pode observar na Figura 1 (HENKEL; PARAMESWARAN, 2007); além disso, os FPGAs não são somente interessantes para a substituição dos ASICs no uso final, mas também para a prototipagem destes.

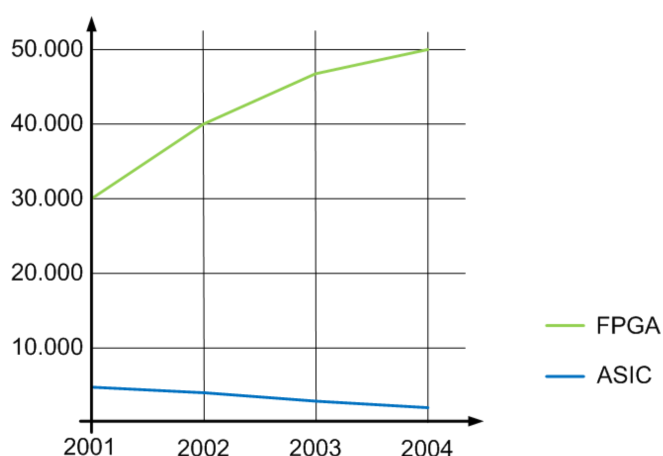


Figura 1 - Utilização de FPGAs X ASICs

[1] Detalhes sobre reconfiguração serão apresentados na Seção 2.1.

Muitos projetos têm utilizado FPGAs para as mais variadas aplicações. Uma das aplicações em FPGA que têm sido desenvolvidas nos últimos anos e reportada na literatura técnica é a implementação de algoritmos SVM de treinamento e classificação (ANGUITA; BONI; RIDELLA, 2003; BIASI; BONI; ZORAT, 2005; ACOSTA HERNANDEZ et al., 2009) que é o foco de interesse de aplicação deste trabalho.

SVM é um classificador binário baseado na teoria da aprendizagem estatística (CORTES; VAPNIK, 1995; VAPNIK, 1998). A principal vantagem deste algoritmo, quando comparado com outros algoritmos de aprendizagem tais como Redes Bayesianas, Discriminante Linear de Fisher e Redes Neurais Artificiais tradicionais, é a sua resolução através de otimização por Programação quadrática, onde o problema dos mínimos locais é ausente. Este novo classificador tem sido cada vez mais utilizado pela comunidade científica em aplicações como classificação de padrões, reconhecimento de imagens, seleção de genes, classificação de textos, entre outras[2].

O posterior desenvolvimento de FPGAs com tecnologia SRAM projetados pela Xilinx levou a uma estrutura que suporta uma forma especial de operação na qual podem ser feitas alterações em determinadas áreas do dispositivo, sem alterar as demais. Este tipo de reconfiguração, denominada reconfiguração dinâmica parcial, possibilitou a implementação de vários subcircuitos que podem ser implementados sucessivamente em uma estrutura única de hardware, aumentando assim a densidade lógica de um chip (i.e., permitindo maior alocação de recursos por área disponível no dispositivo) (LIU; WONG, 1999). Os FPGAs que suportam esse tipo de operação, serão definidos neste trabalho como FPGAs dinamicamente reconfiguráveis (Dynamically Reconfigurable FPGAs, DRFPGAs) (ZHANG; NG, 2000).

As tecnologias de reconfiguração parcial permitiram a implementação de um circuito em partições sucessivas, temporalmente excludentes, em um mesmo dispositivo. Assim, uma dimensão temporal foi adicionada à visão meramente espacial utilizada no projeto de ASICs.

---

[2] Uma lista de aplicações do classificador SVM está disponível no seguinte endereço: http://www.clopinet.com/isabelle/Projects/SVM/applist.html (acessado em 17/01/2010).

A Figura 2 mostra o deslocamento temporal da alocação de recursos no FPGA em um esquema de reconfiguração parcial. Note que se torna necessário considerar aspectos antes inexistentes, tais como dependência de dados na seqüência das partições, via de dados adaptada às várias partições, projeto do controlador de partições, entre outros. Conclui-se que a atenção antes voltada somente ao domínio dos recursos deve agora incluir o domínio do tempo. Para projetos que envolvam arquiteturas dinamicamente reconfiguráveis é necessária uma estratégia de partição bidimensional com um domínio espacial e outro temporal, sendo necessários novos métodos de projeto.

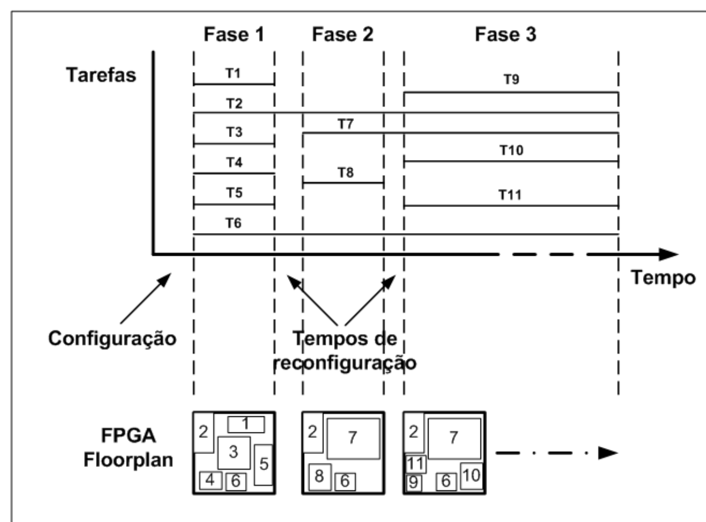


Figura 2 - Particionamento temporal em projetos de DRFPGAs.

1.2. Motivação

O desenvolvimento de DRFPGAs abriu novas possibilidades na área de projetos, tornando possível o desenvolvimento de Sistemas Dinamicamente Reconfiguráveis (SDRs). Estes sistemas permitem melhor utilização da área do FPGA, aumentam a densidade lógica do chip e reduzem o consumo de potência, fator importante no desenvolvimento de sistemas embutidos. No entanto esta é uma área que ainda está em fase de maturação, havendo necessidade de criação de

novas metodologias de projeto, métodos de particionamento temporal e ferramentas específicas (GONZALEZ, 2006).

Algumas implementações em reconfiguração dinâmica têm sido anunciadas em várias aplicações computacionais, trazendo ganhos em área e consumo de potência. Incluem-se nesta lista, aplicações em telecomunicações (ESQUIAGOLA et al., 2005), sistemas automotivos (BECKER et al., 2007), arquitetura de computadores (MAJER et al., 2007; LU et al., 2008), entre outros.

Implementações de SVMs em FPGAs são ainda relativamente recentes, não contemplando a reconfiguração dinâmica, devido às dificuldades algorítmicas nas diversas soluções existentes para o problema. Alguns trabalhos têm apresentando implementações de resolução numérica do problema SVM (GENOV; CAUWENBERGHS, 2003; ANGUITA; BONI; RIDELLA, 2003; BIASI; BONI; ZORAT, 2005), tanto na fase de treinamento quanto na fase de teste. A resolução numérica é feita por operações de produto interno entre vetores, operações estas com alto custo computacional, muitas vezes gerando problemas de precisão caso o número de elementos dos vetores seja elevado. Uma resolução analítica foi proposta com o algoritmo de Otimização Mínima Sequencial (Sequential Minimal Optmization, SMO) (PLATT, 1999), sendo o problema original decomposto sucessivamente em dois subproblemas, onde um deles apresenta apenas duas variáveis e o outro apresenta as variáveis restantes, diminuindo o número de operações vetoriais necessárias. Esta solução é muito mais simples que a numérica, além de apresentar a grande vantagem de não requerer um armazenamento de uma matriz com o quadrado do número de exemplos de treinamento. Apesar de seus pontos positivos que se refletem em restrições menores para implementações em hardware, até o momento, apenas uma implementação em hardware com resolução analítica foi desenvolvida (ACOSTA HERNANDES et al., 2009), baseada em uma arquitetura especifica adequada ao conjunto de exemplos utilizado na fase de treinamento.

O algoritmo SMO é divido em três tarefas específicas que são executadas em uma iteração. Estas são basicamente: 1- Seleção do conjunto de trabalho; 2- Otimização de um par de coeficientes; 3- Atualização global de dados. Estas tarefas são executadas seqüencialmente, sugerindo poder ser positiva a aplicação das técnicas de reconfiguração dinâmica ao algoritmo.

1.3. Justificativas

O grupo de Projeto de Sistemas Eletrônicos Integrados e Software Aplicado (grupo SEIS) do Laboratório de Microeletrônica da USP (LME-USP) têm como seu interesse principal de investigação as metodologias de projeto de sistemas digitais. O grupo tem desenvolvido importantes trabalhos na área de reconfiguração dinâmica (ESQUIAGOLA, 2005; ESQUIAGOLA, 2006; GONZALEZ, 2006; KOJIMA, 2007) e também contribuído para área de desenvolvimento de implementações de SVMs (GONZALEZ, 2006; ACOSTA HERNANDES, 2009). Este trabalho vem dar continuidade à linha de trabalho do grupo SEIS e apresentar um desenvolvimento unindo essas duas linhas de pesquisa.

1.4. Objetivos

Este trabalho tem como objetivo geral contribuir no amadurecimento das técnicas de implementação de sistemas dinamicamente reconfiguráveis, tendo como aplicação o algoritmo de classificação binária Support Vector Machine. Introduzindo uma arquitetura com grande gama de aplicabilidade, serão sistematizadas metodologias de projeto para DRFPGAs e técnicas de particionamento temporal. Mais especificamente os objetivos que este trabalho procura atingir são os seguintes:

- O desenvolvimento de uma arquitetura em hardware para o algoritmo de treinamento de SVM genérica, ou seja, aplicável e configurável a uma grande gama de exemplos/benchmarks.
- Proposta de uma arquitetura reconfigurável para o algoritmo de treinamento de SVM, ou seja,definindo-se as partes fixas e reconfiguráveis, utilizando técnicas conhecidas.

- Implementação da arquitetura SVM reconfigurável em nível RTL e a sua simulação, comprovando o funcionamento correto.
- Síntese do projeto RTL, com posicionamento e roteamento das áreas reconfiguráveis para comprovar a viabilidade do projeto segundo a ocupação de área.

1.5. Organização da Dissertação

Este trabalho está divido em sete capítulos. O capítulo dois apresenta os aspectos teóricos que são a base deste trabalho, subdivido em dois blocos fundamentais: a teoria de FPGAs e da reconfiguração dinâmica, e as definições matemáticas e algorítmicas da SVM. O terceiro capítulo apresenta alguns trabalhos correlatos e que foram utilizados como base para esta pesquisa. O quarto capítulo apresenta as metodologias empregadas neste projeto. No capítulo cinco, a solução de implementação do algoritmo SVM-SMO sem reconfiguração dinâmica é detalhado, enquanto que a proposta da solução com reconfiguração dinâmica é explorada no capítulo seis. O sétimo capítulo apresenta as conclusões, e, posteriormente, são exibidas as referências utilizadas neste trabalho.

2. ASPECTOS TEÓRICOS

Neste capítulo serão abordados fundamentos teóricos importantes para o desenvolvimento deste trabalho, focando principalmente nas duas bases deste projeto: os princípios da reconfiguração dinâmica em FPGAs e as Máquinas de Vetores de Suporte (mais conhecidas como *Support Vector Machines*, SVM).

2.1. Reconfiguração em FPGAs

A principal característica que distingue os FPGAs das soluções ASIC e GPP é a sua configurabilidade ou programabilidade. A evolução dos FPGAs levou ao surgimento de uma série de implementações com estratégias diferentes de configuração que serão apresentadas nas seções seguintes.

2.1.1. Tipos de Reconfiguração

A área de pesquisa de arquiteturas reconfiguráveis é relativamente nova e está em constante desenvolvimento. Ainda não existe uma terminologia unificada totalmente aceita neste campo. Muitos trabalhos têm utilizado termos como configuração, reconfiguração, reconfiguração estática, reconfiguração dinâmica, entre muitos outros, de maneiras distintas com significados distintos, gerando conflitos de interpretação. Serão definidos nesta seção os tipos de reconfiguração e as respectivas terminologias associadas.

Uma classificação interessante, baseada na configurabilidade dos dispositivos foi proposta no trabalho de (LYSAGHT; DUNLOP, 1993) e depois atualizado (ZHANG; NG, 2000). Por definição, todos os FPGAs são programáveis (ou configuráveis), mas existem FPGAs que são enquadrados apenas nesta categoria, que são aqueles baseados em tecnologia *antifuse*[3]. Com o desenvolvimento de FPGAs com tecnologia E2PROM, FLASH e SRAM, surgiu a categoria dos dispositivos que podem ser reconfigurados indefinidamente, porém mantendo essa configuração durante toda a execução. Esse conceito de reconfigurabilidade ficou posteriormente conhecido como reconfiguração estática (ZHANG; NG, 2000). A tecnologia SRAM também gerou outra possibilidade: uma reconfiguração seletiva (ou reconfiguração parcial), onde é possível reconfigurar uma área do FPGA sem modificar os circuitos programados nas áreas restantes. Na prática, o dispositivos comerciais com reconfiguração parcial realizam-na em tempo de execução (ou reconfiguração dinâmica), onde não é necessário interromper o funcionamento do FPGA, para carregar um novo *bitstream*, ou seja, o FPGA é reconfigurado em uma área específica, enquanto outras áreas do FPGA continuam ativas, executando outras operações.

A Figura 3 (LYSAGHT; STOCKWOOD, 1996) detalha o mecanismo da reconfiguração dinâmica parcial ilustrando a atualização seletiva das células lógicas de um FPGA e os recursos de roteamento a ela associados, enquanto que outras áreas do FPGA continuam ativas, sem interrupção.

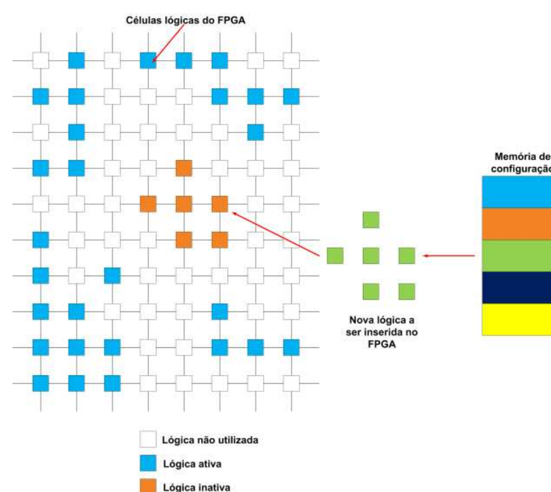
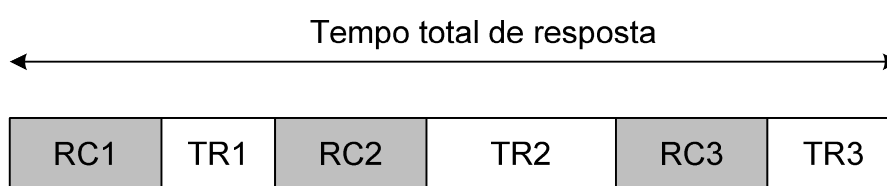


Figura 3 - Reconfiguração Dinâmica Parcial

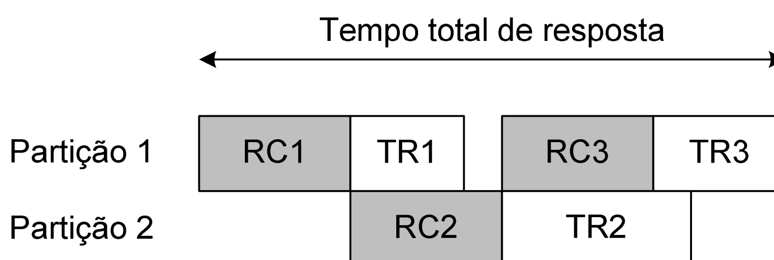
---

[3] Esta tecnologia permite apenas uma configuração (i.e., uma vez configurado o circuito permanecerá naquela FPGA).

O conceito da diferença entre reconfiguração estática e reconfiguração dinâmica adotada neste trabalho, será ilustrado na Figura 4. Considere três tarefas a serem executadas “TR1”, “TR2” e “TR3”. Estas tarefas são independentes entre si. No caso de reconfiguração estática Figura 4(a), o FPGA configuraria a tarefa 1 e depois esta seria executada, após a execução o FPGA baixaria um *bitstream* que reconfiguraria toda a sua lógica interna e implementaria a tarefa 2. Em seguida o mesmo procedimento seria executado para a reconfiguração e execução da tarefa 3.



(a) Reconfiguração Estática



(b) Reconfiguração Dinâmica

Figura 4 - Reconfiguração Estática (a) X Reconfiguração Dinâmica (b)

Para a exposição do comportamento da reconfiguração dinâmica, vamos considerar na Figura 4(b), onde há duas partições reservadas internamente no FPGA, de forma que uma fique em modo de execução enquanto outra fique em modo de reconfiguração. Na primeira partição seria configurada a tarefa 1 e em seguida esta seria executada ao mesmo tempo que a tarefa 2 seria reconfigurada. Após o término da reconfiguração da tarefa 2 esta seria executada ao mesmo em que a tarefa 3 seria reconfigurada. Por fim a tarefa 3 seria executada. Deste modo a reconfiguração é feita *on-line* e de modo parcial.

A estratégia de reconfiguração dinâmica tem o potencial de otimizar o uso de área, pois, reconfigurando repetidas vezes uma mesma partição do FPGA, a quantidade de recursos utilizados (para processamento real) supera a quantidade de recursos existentes. A remoção de um módulo que está inativo disponibiliza a área correspondente para novas funções.

Ao contrário da arquitetura clássica Von Neumann, o FPGA permite a exploração da simultaneidade, sendo possível realizar várias tarefas em paralelo. Isso pode ser explorado na reconfiguração dinâmica de forma a mascarar o tempo de reconfiguração, reduzindo o tempo total de resposta, como se pode observar na Figura 4.

No presente trabalho os seguintes termos serão definidos:

- **Reconfiguração Estática** – Reconfiguração que é carregada no FPGA e nele permanece por toda a duração da aplicação. O termo reconfiguração total será considerado sinônimo.
- **Reconfiguração Parcial** – Reconfiguração seletiva das células lógicas de uma FPGA e os recursos de roteamento a elas associados, enquanto outras áreas do FPGA continuam ativas ou não.
- **Reconfiguração Dinâmica** – Reconfiguração em tempo de execução (run-time) do FPGA, das células lógicas e dos recursos de roteamento a elas associados, enquanto outras áreas do FPGA continuam ativas. Serão considerados sinônimos: reconfiguração parcial dinâmica; *run-time Reconfiguration* – RTR (LYSAGHT; DUNLOP, 1993), *on-line reconfiguration*, *logic caching* (ZHANG; NG, 2000).
- **Sistema Dinamicamente Reconfigurável[4] – SDR** – Sistema implementado em FPGA, ou não, que inclua reconfiguração dinâmica. Será considerado termo equivalente a Sistema Parcialmente Reconfigurável – SPR (HORTA, 2002).
- **FPGA Dinamicamente Reconfigurável – DRFPGA** – FPGA que inclui infraestrutura de reconfiguração parcial dinâmica.

---

[4] Termo também adotado por (SOARES, 2005), (ESKINAZI, 2005).

2.1.2. Arquiteturas de Sistemas Dinamicamente Reconfiguráveis

A Xilinx hoje é o principal fabricante de DRFPGAs, sendo a linha Virtex a mais adotada por acadêmicos e pesquisadores de SDRs. Com o objetivo de introduzir conceitos gerais sobre a arquitetura de DRFPGAs, nesta seção será apresentada a arquitetura do DRFPGA Xilinx Virtex-II. Nos concentramos na família Virtex-II por ela ser a primeira de grande porte com capacidade de reconfiguração dinâmica e ter uma arquitetura razoavelmente simples de ser entendida. A família Virtex-4 apresenta várias evoluções, mas a arquitetura básica é ainda a mesma.

O DRFPGA Virtex-II baseia-se em uma arquitetura simétrica composta pelos seguintes elementos ilustrados na Figura 5 (XILINX, 2007):

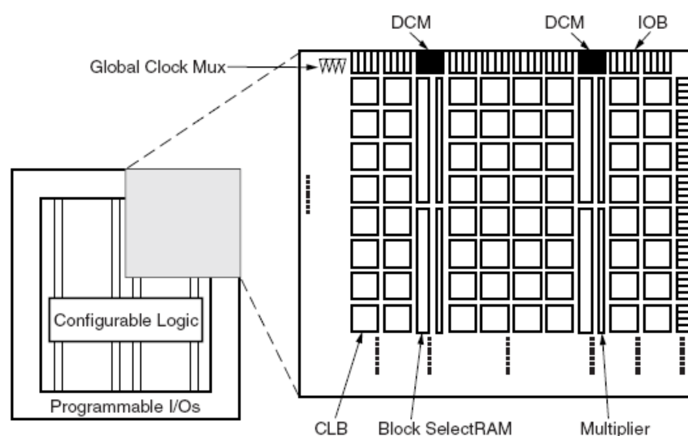


Figura 5 - Arquitetura do Virtex-II

1) Os blocos de entrada e saída (*Input Output Blocks*, IOBs) são blocos responsáveis pela interface entre os CLBs e os pinos do componente. Os IOBs também incluem elementos de armazenamento, os quais podem ser configurados como *flip-flops* tipo D ou *latches*, de acordo com a conveniência. Os IOBs ficam localizados à margem do layout.
2) Os blocos lógicos configuráveis (*Configurable Logic Blocks*, CLBs), ilustrado na Figura 6, são os principais elementos de um DRFPGA. Estes tornam possível a

implementação e o roteamento de uma função lógica. Eles são compostos por uma matriz de chaves (*switch matrix*) que acessam caminhos de roteamento, e por quatro fatias (*slices*).

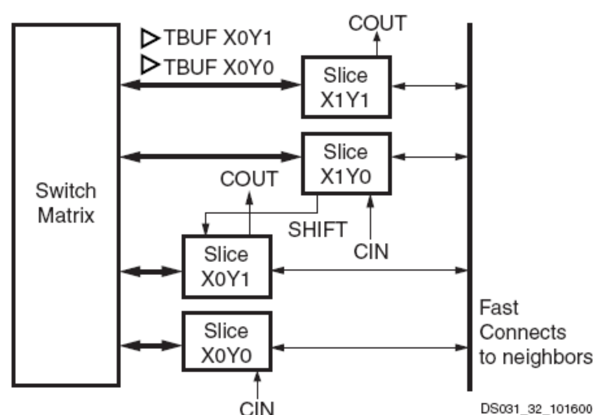


Figura 6 - Bloco lógico configurável.

Adjacentes aos CLB estão alocadas matrizes gerais de roteamento (*General routing matrix*, GRM), que têm por função proporcionar aos CLBs o acesso aos caminhos de roteamento de propósito geral (*general-purpose routing*).

Os *slices* são compostos por dois geradores de funções, conhecidos como *Look up Tables* (LUTs) que são componentes de 4 bits que podem gerar funções lógicas, trabalhar como registradores de deslocamento ou como blocos de memória. Além das LUTs, o *slice* possui dois blocos de lógica *carry*, dois registradores, dois multiplexadores e elementos de geração de lógica aritmética. A estrutura de um *slice* está ilustrada na Figura 7.

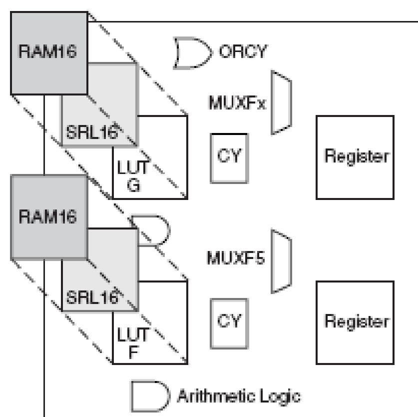


Figura 7 - Slice do Virtex-II.

3) O Virtex-II também inclui na sua arquitetura elementos de memória RAM, multiplicadores otimizados (geralmente utilizados para processamento digital de sinais) e gerenciadores digitais de clock (Digital Clock Managers, DCMs), utilizados para gerar diferentes freqüências e deslocamentos de fase a partir do clock de entrada.

A configuração do dispositivo é realizada através dos portos de reconfiguração. Existem vários portos de reconfiguração no Virtex-II tais como *Boundary Scan* ou JTAG, o *SelectMap*, o *Master-Serial* e o *Slave-Serial*. E um porto para auto-reconfiguração[5], o ICAP.

Desde a comercialização da família Virtex II, outras gerações importantes de DRFPGAS surgiram. O dispositivo Virtex-II Pro utiliza a mesma estrutura básica do Virtex-II, mas introduz núcleos de hardware do processador *Power-PC*. Em 2004 foi lançada a família Virtex-IV, tendo como principal diferença com a Virtex-II a possibilidade de reconfigurar blocos bem menores que uma coluna inteira (menor unidade reconfigurável nos DRFPGAs anteriores) chamados de *frames*. Esta abordagem reduziu sensivelmente o tempo de reconfiguração, já que este é diretamente proporcional ao tamanho do *bitstream*. Outra importante inovação do Virtex-IV é a possibilidade de efetuar a comunicação entre CLBs tanto na horizontal quanto na vertical.

---

[5] Reconfiguração gerada no dispositivo e aplicada ao próprio dispositivo. Para tanto é necessário um controlador de reconfiguração dentro do próprio sistema.

Com a evolução dos dispositivos, as metodologias de projeto de SDRs em FPGAs também evoluíram. O método *Modular Design Flow* (XILINX, 2004) foi a primeira metodologia proposta pela Xilinx para projetos de SDRs. Este método está baseado nas tecnologias das famílias Virtex, Virtex II e Virtex II Pro. O método *Modular Design Flow* pode ser adotado, baseado em módulo (module-based) ou baseado em diferença (difference-based). No método baseado em módulo, são criadas configurações parciais das descrições das partições de um sistema e a comunicação entre elas é feita por uma entidade chamada *bus-macro*, que garante canais fixos de roteamento entre os módulos reconfiguráveis. Procura-se garantir assim uma transição segura dos sinais dos módulos que irão se (des)conectar no tempo, ou seja, estejam isolados através de buffer controlados por enable. Este conceito é muito similar ao conceito de "*gaskets*" proposto por (HORTA, 2002). A estrutura básica de um DRFPGA com *bus-macros* está representada na Figura 8.

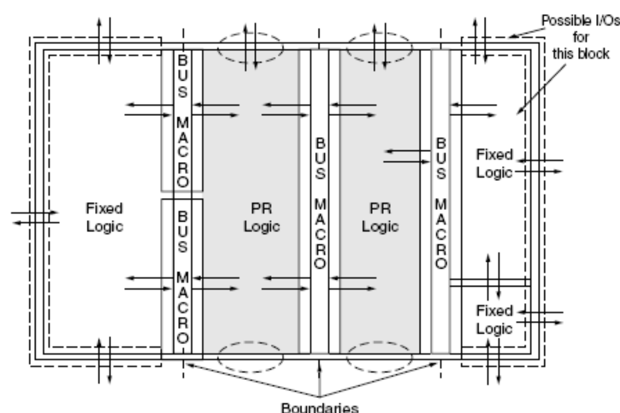


Figura 8 - Estrutura de bus-macros em um DRFPGA

Nesta figura, estão representadas quatro partições, sendo duas delas fixas e duas reconfiguráveis. Os três bus-macros executam a comunicação entre as partições. As conexões fixas alocadas nestes bus-macros garantem que os canais de comunicação funcionem mesmo com a alteração da arquitetura, tanto de partição fixa para partição reconfigurável quanto de partição reconfigurável para partição reconfigurável.

Para a implementação *difference-based*, um *bitstream* é gerado a partir das diferenças entre as partições reconfiguráveis que ocupem a mesma partição. Como

resultado, o *bitstream* é menor, podendo ser carregado mais rapidamente e ocupar menos área para o seu armazenamento.

O método *Partial Reconfiguration Early Access* (XILINX, 2008), foi proposto em 2005 em substituição aos métodos anteriores. Com a tecnologia disponibilizada pelo DRFPGA Virtex-IV, tornou-se necessário uma metodologia que viabilizasse as inovações tecnológicas deste dispositivo, reduzindo sensivelmente o tamanho do *bitstream* e por conseqüência o tempo de reconfiguração. O Early Access tornou desnecessária a utilização de *bus-macros* para canais que atravessam regiões reconfiguráveis, mas que não possuem conexões com estas, tornando a metodologia mais simples e eficiente. Além disso, a conexão entre módulos adjacentes passa a ser feita pela lógica dos CLBs.

A nova metodologia também apresenta melhor manipulação dos CLBs, sendo possível fazer a diferenciação entre *bus-macros* extensos e curtos. *Bus-macros* curtos são conexões de CLBs vizinhos, enquanto que em *bus-macros* extensos, dois CLBs são conectados com um intervalo de outros dois CLBs entre eles (BOBDA, 2007). A Figura 9(a) e Figura 9(b) ilustram um *bus-macro* curto e um extenso respectivamente. Com o uso dos *bus-macros* extensos fica possível aproveitar *bus-macros* não utilizados entre estes como ilustrado na Figura 10. Esse tipo de arranjo foi denominado aninhamento.

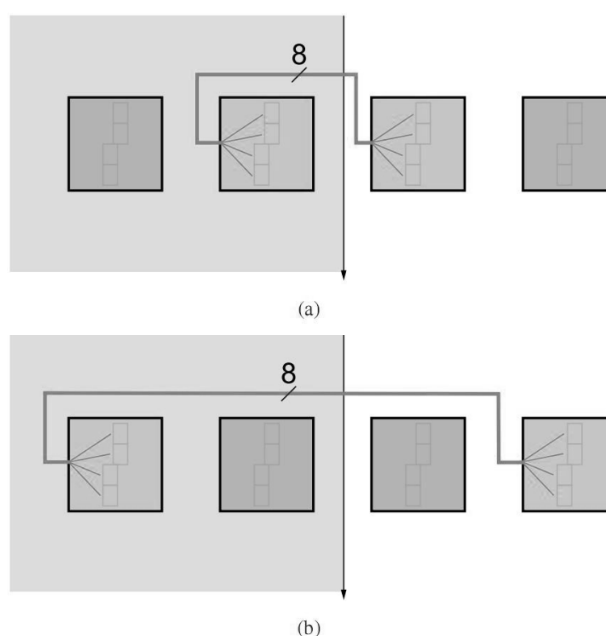


Figura 9 - Bus-macros curtos x extensos

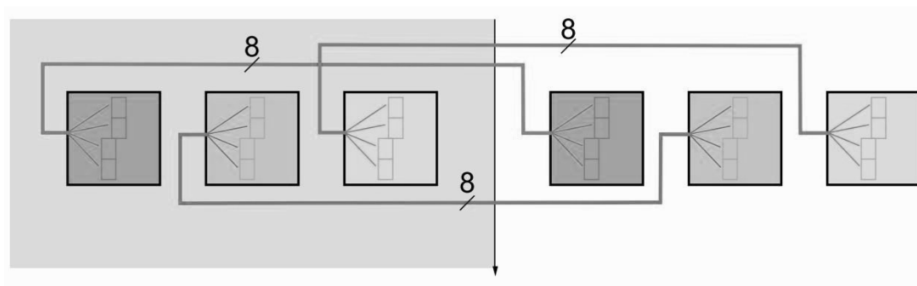


Figura 10 - Bus-macros aninhadas

Além destas vantagens, a metodologia Early Access tornou possível a reconfiguração de apenas parte do circuito; antes, a reconfiguração era possível apenas em colunas inteiras.

Os *bus-macros* também ficaram mais flexíveis nos sentidos de conexão. Anteriormente, as transmissões de dados eram possíveis apenas horizontalmente (da direita para esquerda e da esquerda para direita), com a nova metodologia a transmissão vertical se tornou possível (de cima para baixo e de baixo para cima).

2.2. Support Vector Machine (Ou Máquina de Vetor de Suporte)

Máquinas de aprendizado tais como Redes Neurais, Redes Bayesianas, Discriminante Linear de Fisher (LERNER; LAWRENCE, 2001) e *Support Vector Machines* (CORTES; VAPNIK, 1995; VAPNIK, 1998) são aplicadas em tarefas de classificação de padrões tais como reconhecimento de imagens, classificação de textos, seqüenciamento genético, entre outros. Após uma avaliação preliminar, os algoritmos de SVM mostraram eficiência superior quando comparados com outras Redes Neurais Artificiais tradicionais. Outros pontos positivos do SVM também merecem ser destacados: ausência de mínimos locais, redução de *overfitting*, bom desempenho de generalização, solução ótima encontrada em tempo polinomial, entre outros.

2.2.1. Definição Matemática e Algorítmica da SVM

Nesta seção serão introduzidos os conceitos matemáticos mais importantes da SVM tendo em vista o caso mais genérico, o conjunto de amostras não-separável para classificação e plano de separação não-linear (ou, simplesmente caso não-separável e não-linear). Outros dois casos: o separável e linear, e não-separável e linear, podem ser considerados condições especiais do caso genérico. O caso não-separável e linear é especialmente relevante para este estudo dado que o objetivo da arquitetura criada é permitir o treinamento de diversos tipos de amostras de entrada, como será descrito no capítulo 5.

O objetivo do aprendizado da SVM é obter uma função $\widehat{\Phi}: X \rightarrow Y$, baseada em uma série de exemplos de treinamento $\{\vec{x}_i, y_i\}_{i=1}^{m}$, onde m é o número de amostras de treinamento, $\vec{x}_i \in X \subseteq \Re^d$ são padrões de entrada e $y_i \in Y = \{-1, +1\}$ são as saídas desejadas. Considerando que não há um plano linear de separação satisfatório para os exemplos de entrada, os vetores $\vec{x}$ do espaço de entrada (*input space*) são mapeados em um espaço de características (*feature space*) $Z \subseteq \Re^n$ com $n \gg d$ através de um mapeamento não linear $\varphi: X \rightarrow Z$. A aplicação da transformação em cada entrada $\vec{x}_i$ gera um vetor correspondente $\vec{z}_i$ no espaço Z como ilustrado na Figura 11. Deste modo, um hiperplano linear pode ser utilizado para separar os pontos no espaço de características. Assim, a função $\widehat{\Phi}$ é dada por:

$$\widehat{\Phi} = \left(\vec{w} \cdot \varphi(\vec{x})\right) + b \qquad (1)$$

onde $\vec{w}$ é um vetor normal ao hiperplano de separação e b é um parâmetro limiar (bias).

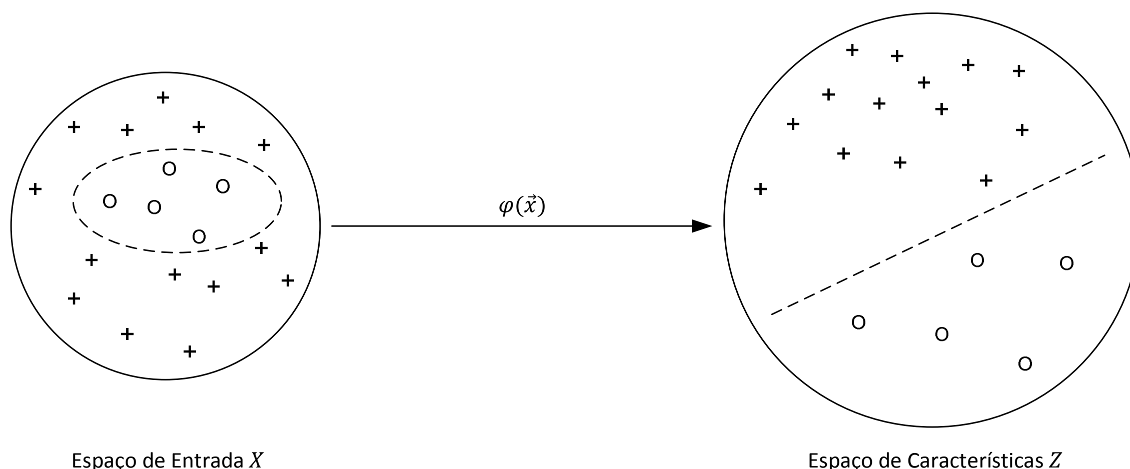


Figura 11 - Mapeamento não linear em SVM

É possível separar linearmente os vetores de entrada no espaço de características mesmo que eles não sejam separáveis no espaço de entrada. Isto é possível através de uma propriedade matemática conhecida como *kernel*, que nada mais é do que o produto interno da transformada de dois vetores de entrada. Graças a esta propriedade, é possível computar o mapeamento implicitamente, sem a necessidade do conhecimento explicito de uma função φ. O *kernel* é matematicamente definido por:

$$K(\vec{x}_i, \vec{x}_j) = \varphi(\vec{x}_i) \cdot \varphi(\vec{x}_j) \quad (2)$$

O processo de mapeamento do espaço de entrada para o espaço de características e a geração do hiperplano linear são características da primeira fase do algoritmo SVM, conhecida como fase de treinamento. Na segunda fase, conhecida como fase de teste, os vetores de entrada são classificados em um dos dois lados do hiperplano.

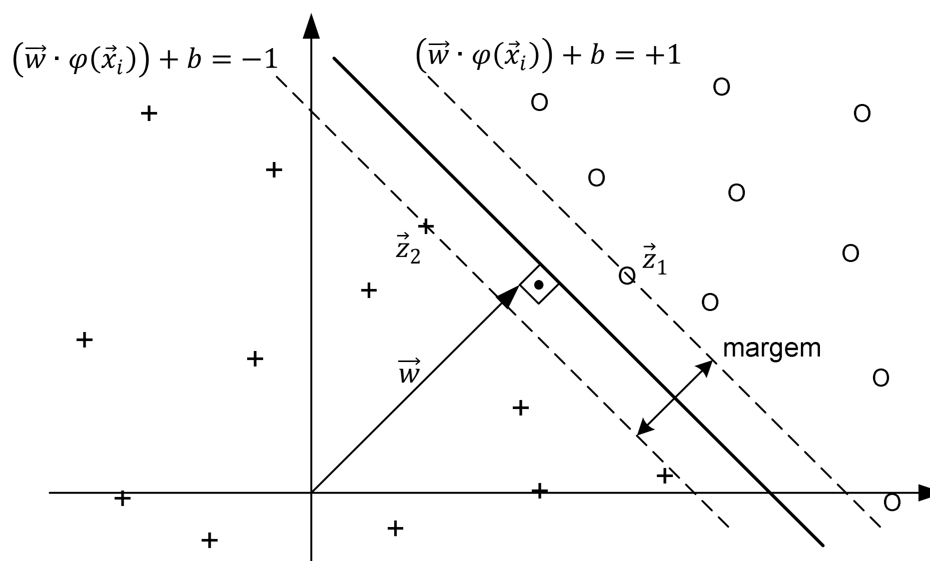


Figura 12 - Hiperplano ótimo

Na Figura 12 verificam-se dois tipos de amostras no espaço de características Z separados por um hiperplano[6], cuja distância à origem $|b|/\|\vec{w}\|$, sendo o vetor $\vec{w}$ um vetor normal ao hiperplano. Assim sendo, é possível determinar a equação do hiperplano que separa os exemplos positivos dos negativos. Os vetores $\vec{z}_1$ e $\vec{z}_2$ são vetores de suporte (i.e., aqueles mais próximos do hiperplano). Entre os possíveis hiperplanos de separação, aquele que melhor classifica os vetores de entrada é o hiperplano com a maior margem de separação entre os dois conjuntos de treinamento. Este hiperplano é conhecido como hiperplano ótimo.

O objetivo da fase de treinamento da SVM, portanto, é a obtenção do vetor $\vec{w}$ e do escalar b do hiperplano que maximiza a margem.

Para descrever um método com o fim de alcançar o objetivo acima descrito, serão adotadas as seguintes considerações (BURGES, 1998):

$$y_i(\vec{w} \cdot \varphi(\vec{x}_i) + b) - 1 \geq 0; \;\; \forall i = 1, \dots, m \qquad (3)$$

[6] A representação em duas dimensões é meramente ilustrativa, dada a impossibilidade de representar o alto número de dimensões utilizadas em SVM.

A equação (3) é válida caso os exemplos sejam separáveis (i.e., não exista nenhuma amostra de um tipo “+” alocada erroneamente no outro lado do hiperplano, dentro domínio do tipo “O”. Ou vice-versa). No entanto é comum que nos treinamentos reais não haja separação ótima sendo necessário adicionar um fator de erro para cada exemplo dado por ξ_i. Introduzindo o fator de erro em (3) temos:

$$y_i(\vec{w} \cdot \varphi(\vec{x}_i) + b) - 1 \geq 1 - \xi_i; \ \ \forall i = 1, \dots, m \tag{4}$$

Dadas as considerações, calculemos o valor da margem. Analisando a Figura 12, temos que:

$$\vec{w} \cdot (\vec{z}_1 - \vec{z}_2) = 2 \quad \Rightarrow \quad \frac{\vec{w}\,(\vec{z}_1 - \vec{z}_2)}{||\vec{w}||} = \frac{2}{||\vec{w}||} \tag{5}$$

Considerando f_m como a função dos componentes de $\vec{w} = w_1, \dots w_d$ a margem máxima é obtida maximizando a função f_m, o equivalente a minimização de $||\vec{w}||$. Algebricamente este problema é formulado com as ferramentas da programação quadrática (PQ), devido à conveniência do desenvolvimento matemático das equações. Deste modo a formulação do problema é equivalente a[7]:

$$\min_{\vec{w},b} \left\{ f_m(\vec{w}) = \frac{1}{2} ||\vec{w}||^2 + C \sum_{i=1}^{m} \xi_i \right\} \tag{6}$$

Com as restrições dadas por (4). O primeiro termo da equação maximiza a margem enquanto o segundo penaliza a presença de pontos classificados de forma errada. C é uma constante que estabelece o equilíbrio entre os dois termos.

---

[7] O fator multiplicativo $\frac{1}{2}$ é aplicado ao termo quadrático para evitar posteriormente a multiplicação por 2 em suas derivadas.

Note que a função f_m em (6) é diferenciável para qualquer $\vec{w}$, enquanto que para $f_m = 2/\|\vec{w}\|$, como na equação (5), não é possível diferenciar quando os componentes de $\vec{w}$ são iguais a zero.

A equação (6) apresenta grande dificuldade de resolução dado o grande número de variáveis $\vec{x}_i$, de (4) que devem ser computadas. Uma possível forma de encontrar o mínimo de uma função convexa é através da teoria do multiplicador de Lagrange (VAPNIK, 1998), que introduz uma variável dual para cada variável original da restrição. A lagrangeana do problema acima descrito pode ser calculada como:

$$L(\vec{w}, b, \vec{\alpha}) = \frac{1}{2}\|\vec{w}\|^2 - \sum_{i=1}^{m} \alpha_i (y_i(\vec{w} \cdot \varphi(\vec{x}_i) + b) - 1) \qquad (7)$$

E a equação (6) pode ser reformulada como um problema dual de minimização de α, como demonstrado na seguinte equação:

$$\min_\alpha \left\{ L(\alpha) = \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j \, y_i y_j \cdot \varphi(\vec{x}_i) \cdot \varphi(\vec{x}_j) - \sum_{i=1}^{m} \alpha_i \right\} \qquad (8)$$

Restrito a:

$$\sum_{i=1}^{m} \alpha_i y_i = 0 \qquad (9)$$

e

$$0 \leq \alpha_i \leq C \qquad (10)$$

A equação (8) constitui um problema de programação quadrática (PQ) devido à sua dependência quadrática de α_i. A resolução deste problema será abordada na seção 2.2.3.

A minimização de L contém restrições sobre os multiplicadores de Lagrange, sendo estas mais simples que as originais sobre $\vec{w}$ e b. Com a obtenção dos multiplicadores de Lagrange, é possível calcular $\vec{w}$ e b com as equações (11) e (12):

$$\vec{w} = \sum_{i=1}^{\boldsymbol{m}} \alpha_i y_i \varphi(\vec{x}_i) \tag{11}$$

$$b = \sum_{i=1}^{\boldsymbol{m}} \alpha_i y_i \cdot \varphi(\vec{x}_i) \cdot \varphi(\vec{x}_k) - y_k\,; \quad 0 < \alpha_k < C \tag{12}$$

Uma vantagem de se usar a expressão (8) é que os exemplos com multiplicadores de Lagrange α_i resultantes que são nulos podem ser eliminados da computação de $\vec{w}$ e b. Os multiplicadores de Lagrange α_i não nulos correspondem a exemplos alocados a uma distância mínima do hiperplano de separação e os $\vec{x}_i$ associados a eles são denominados vetores de suporte.

Após solucionar a equação (8) e obter todos os dados de treinamento, será possível executar a fase de teste. A equação para qualificar novos exemplos pode ser obtida substituindo o resultado de (11) em (1).

$$\operatorname{sgn}\left\{\widehat{\Phi}(\vec{x}) = \sum_{i=1}^{m} \alpha_i y_i \cdot K(\vec{x}_i, \vec{x}) + b\right\} \tag{13}$$

Cujo sinal determinará a qual classe a amostra $\vec{x}$ pertencerá.

2.2.2. Condições Karush-Kuhn-Tucker

As condições de Karush-Kuhn-Tucker (KKT) são necessárias para um problema de programação matemática em geral dada por:

$$\min\{f(x_1, \cdots, x_r)\} \tag{14}$$

Restrito a:

$$g_i(x_1, \cdots, x_r) \geq 0;\ \forall i = 1, \cdots, m \tag{15}$$

$$h_j(x_1, \cdots, x_r) = 0;\ \forall i = 1, \cdots, p \tag{16}$$

Transportando as condições KKT à equação (8) e realizando o desenvolvimento matemático necessário, estas são reduzidas (PLATT, 1999) a:

$$\alpha_i = 0 \Leftrightarrow y_i u_i \geq 1 \tag{17}$$

$$0 < \alpha_i < C \Leftrightarrow y_i u_i = 1 \tag{18}$$

$$\alpha_i = C \Leftrightarrow y_i u_i \leq 1 \tag{19}$$

Sendo α_i o multiplicador de Lagrange associado ao exemplo de treinamento dado pela entrada $\vec{x}_i$ e a saída y_i, C o fator definido na seção 2.2.1, e u_i a função:

$$u_i(\vec{x}) = \widehat{\Phi}(\vec{x}) = \sum_{i=1}^{m} \alpha_i y_i \cdot K(\vec{x}_i, \vec{x}) + b \tag{20}$$

Avaliada em $\vec{x} = \vec{x}_i$.

Além de serem úteis para determinar se a solução aproximada resultante de uma iteração em um algoritmo é a solução final, as condições KKT são úteis na construção de novos algoritmos para o problema de programação quadrática de *SVM*.

2.2.3. O algoritmo SMO, *Sequential Minimal Optimization*

Desde a formalização do problema do SVM, alguns métodos de resolução do problema da PQ em SVM foram propostos (VAPNIK, 1982; OSUNA; FREUND; GIROSI, 1997; ANGUITA; BONI; RIDELLA, 2003), entre outros. Porém, estes apresentam uma resolução do problema PQ-SVM baseada em métodos numéricos, o que resulta em problemas de armazenamento de matrizes também implicando um alto número de operações lógico-aritméticas.

O algoritmo Sequential Minimal Optimization proposto por Platt (PLATT, 1999) resolve o problema PQ-SVM de maneira analítica, com um algoritmo de decomposição que a cada passo soluciona o problema de otimização SVM de menor tamanho possível. Conseqüentemente, a quantidade de dados armazenados assim como a quantidade de operações aritméticas é consideravelmente reduzida. Isso resulta em um tempo menor de execução com economia de recursos aritméticos.

Uma iteração do algoritmo *SMO* é composta por três partes: seleção de um par de coeficientes, otimização dos coeficientes selecionados e atualização de dados. Estas etapas serão detalhadas a seguir.

Para encontrar o menor número possível de coeficientes a se otimizar por iteração, devemos considerar a restrição dada por (9). Dada esta restrição, temos que, para $0 < p < m + 1$:

$$\alpha_p y_p = -\sum_{i=1}^{p-1} \alpha_i y_i - \sum_{i=p+1}^{m} \alpha_i y_i = \text{constant} \quad (21)$$

Portanto α_p é constante e não pode ser modificado. Assim sendo, o menor subconjunto que pode ser otimizado contém dois multiplicadores de Lagrange, com a formulação, para $0 < \{p, q\} < m + 1$:

$$\alpha_p y_p + \alpha_q y_q = -\sum_{i=1}^{p-1} \alpha_i y_i - \sum_{i=p+1}^{q-1} \alpha_i y_i - \sum_{i=q+1}^{m} \alpha_i y_i = \gamma \quad (22)$$

Onde γ é constante.

A otimização é feita analiticamente e a velocidade da convergência é sujeita à escolha dos Multiplicadores Langrangeanos, α_p e α_q a serem otimizados em cada iteração, assim como a velocidade de execução do algoritmo.

A equação (22) pode ser representada por uma reta com inclinação de 135 ou 45 graus em um plano de coordenadas (y_p e y_q podem ser +1 ou –1). Considerando também a restrição dada por $0 \leq \alpha_i \leq C$, os possíveis valores de α_p e α_q durante as iterações de otimização deverão estar contidos em uma superfície quadrada delimitada por 0 e C. A delimitação precisa pode ser obtida, para um par de valores α_p e α_q, pela análise em diferentes situações:

a) Se $y_p \neq y_q$

No intuito de obter os valores mínimos e os valores máximos para α partiremos das seguintes considerações: $y_p = +1$ e $y_q = -1$. Assim a equação (22) pode ser reescrita como:

$$\alpha_p - \alpha_q = \gamma \tag{23}$$

Portanto, γ variando entre $-C \leq \gamma \leq C$, como mostrado na Figura 13:

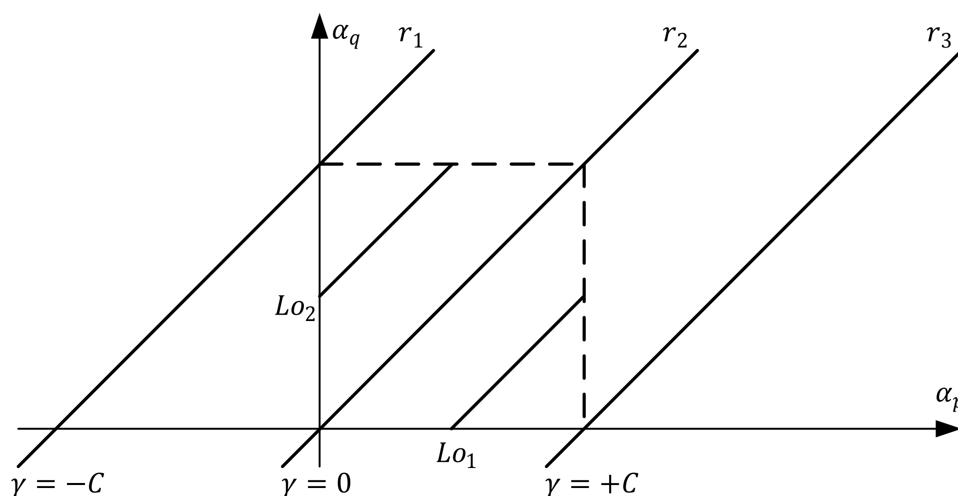


Figura 13 - Representação dos possíveis valores de α_p e α_q para $y_p \neq y_q$.

Entre r_1 e r_2 estão as retas para os valores de γ negativos permitidos e entre r_2 e r_3 estão as retas para os valores de γ positivos permitidos. Entre r_2 e r_3 o valor mínimo de α_2 é $Lo_1 = 0$. Entre r_1 e r_2 o valor mínimo de α_2 ocorre quando $\alpha_1 = 0$, portanto $Lo_p = -\gamma = \alpha_q - \alpha_p$. Portanto o valor mínimo de α_q é:

$$Lo = \max(0, \alpha_q - \alpha_p) \tag{24}$$

Seguindo o mesmo raciocínio para o valor máximo, conclui-se que:

$$Hi = \min(C, C + \alpha_q - \alpha_p) \tag{25}$$

Pode ser demonstrado que os mesmos resultados serão obtidos quando os valores de y adotados forem $y_p = -1$ e $y_q = +1$.

b) Se $y_p = y_q$

Para este segundo caso teríamos a situação indicada na Figura 14:

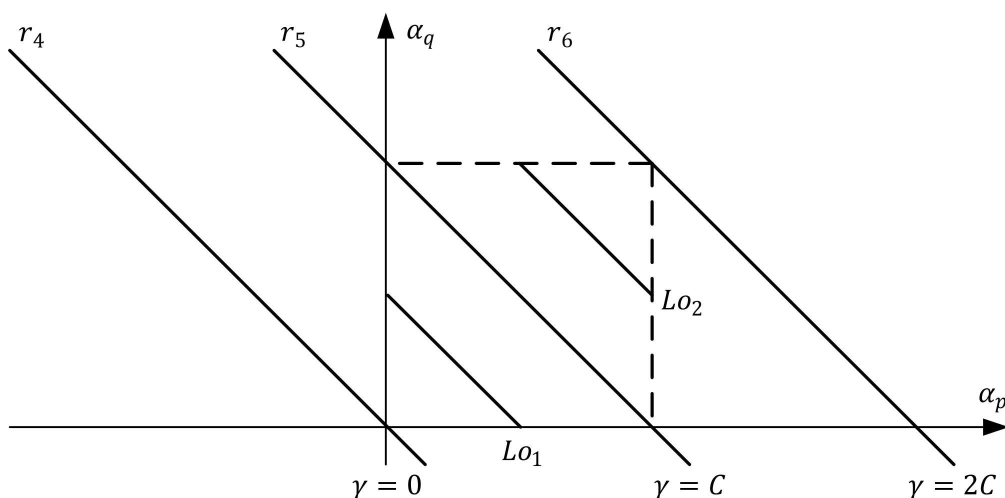


Figura 14 - Representação dos possíveis valores de α_p e α_q para $y_p = y_q$.

Aplicando novamente o raciocínio desenvolvido no item “a” obtêm-se as seguintes equações:

$$Lo = \max(0, \alpha_q + \alpha_p - C) \quad (26)$$

$$Hi = \min(C, \alpha_q + \alpha_p) \quad (27)$$

A partir das equações acima descritas é possível demonstrar através de desenvolvimento matemático (PLATT, 1999), (GONZALEZ, 2006), (KEERTHI et al., 2001) uma expressão para um novo valor de α_q no final de uma iteração:

$$\alpha_q^{new} = \alpha_q^{old} + \frac{y_q(b_{low} - b_{up})}{\eta} \quad (28)$$

Onde α_q^{old} é o valor do coeficiente α_q antes da otimização e η é equivalente a derivada segunda de $L(\alpha)$ em relação a α_q:

$$\eta = \frac{\partial^2 L(\alpha)}{\partial \alpha_2} = K(\vec{x}_p, \vec{x}_p) + K(\vec{x}_q, \vec{x}_q) - 2K(\vec{x}_q, \vec{x}_p) \quad (29)$$

Após o cálculo de α_q^{new} este deve ser corrigido de acordo com os limites estabelecidos por Hi e Lo, como definido em (30).

$$\alpha_q^{new} = \begin{cases} Hi \text{ se } \alpha_q^{new} \geq Hi \\ \alpha_q^{new} \text{ se } Lo < \alpha_q^{new} < Hi \\ Lo \text{ se } \alpha_q^{new} \leq Lo \end{cases} \quad (30)$$

E, b_{up} e b_{low} são dados por

$$b_{up} = \max\{F_i\}, i \in I_{up} \quad (31)$$

$$b_{low} = \min\{F_i\}, i \in I_{low} \quad (32)$$

Com:

$$I_{up} = \{t \mid \alpha_t < C, y_t = +1 \text{ ou } \alpha_t > 0, y_t = -1\} \quad (33)$$

$$I_{low} = \{t \mid \alpha_t < C, y_t = -1 \text{ ou } \alpha_t > 0, y_t = +1\} \quad (34)$$

e

$$F_i = -y_i \frac{\partial L(\alpha)}{\partial \alpha_i} = y_i - \sum_{j=1}^{m} \alpha_j y_j K(\vec{x}_j, \vec{x}_i) \quad (35)$$

Já o cálculo de um novo valor de α_p após uma iteração α_p^{new} pode ser definido utilizando a restrição (22):

$$\alpha_p^{new} = \alpha_p^{old} + y_p y_q\left(\alpha_q^{old} - \alpha_q^{new}\right) \tag{36}$$

O algoritmo SMO baseia-se em uma heurística para a seleção dos pares α_p e α_q a serem otimizados. Os multiplicadores selecionados ao final são aqueles que correspondem a b_{up} e b_{low}. Pode ser matematicamente provado que ao menos um dos coeficientes escolhidos não satisfaz as condições KKT se a condição $b_{up} > b_{low}$ for verdadeira (KEERTHI et al., 2001). Se $b_{up} \leq b_{low}$, então todos os coeficientes satisfazem as condições KKT e o algoritmo pode ser finalizado. Portanto, a computação de b_{up} e b_{low} serve tanto para a escolha do par a ser otimizado, assim como para critério de parada. Essa verificação é realizada com uma tolerância ε.

A cada iteração, dados são atualizados e um novo par é selecionado. Ao final de uma iteração, α_p^{new} é definido segundo as condições de (36) e α_q^{new} é computado de acordo com equações (28), (29) e (30). Os valores de I_{up} e I_{low} devem ser atualizados, assim como o valor do novo F_i :[8]

$$F_i^{new} = F_i^{old} + y_p\left(\alpha_p^{old} - \alpha_p^{new}\right)K\left(\vec{x}_p, \vec{x}_i\right) + y_q\left(\alpha_q^{old} - \alpha_q^{new}\right)K\left(\vec{x}_q, \vec{x}_i\right) \tag{37}$$

Desta forma, b_{up} e b_{low} podem também ser atualizados para uma nova iteração se a minimização geral não tiver sido alcançada.

Como a última etapa do algoritmo, o valor de b pode ser calculado como:

$$b = \frac{b_{up} + b_{low}}{2} \tag{38}$$

[8] Os multiplicadores lagrangeanos são iniciados com valor zero, enquanto que a função F_i é iniciada com os valores de y_i.

Os desenvolvimentos acima descritos podem ser resumidos em três tarefas sequenciais principais, que estarão associadas a blocos de arquitetura a serem apresentados na seção 4.4. As tarefas são descritas a seguir :

a) Seleção: procedimento para a escolha do par de coeficientes a ser otimizado, caracterizado pelas equações (31), (32), (33), (34) e (35).
b) Otimização: procedimento para otimizar os coeficientes escolhidos na tarefa a), caracterizado pelas equações (28), (29), (30) e (36).
c) Atualização: com os novos coeficientes otimizados, fazer a atualização dos parâmetros de computação de cada exemplo, de acordo com a equação (37).

3. TRABALHOS CORRELATOS

O assunto de implementação em hardware de algoritmos SVM é relativamente novo até mesmo para implementações em ASICs e em FPGAs sem reconfiguração dinâmica. E até o presente momento apenas uma solução analítica do treinamento SVM foi desenvolvida em hardware (ACOSTA HERNANDES et al, 2009) e apenas um estudo sobre metodologia de reconfiguração dinâmica aplicado a SVM foi feito (GONZALEZ, 2006). Este capítulo irá apresentar uma breve descrição destes e de outros trabalhos relacionados a esta pesquisa.

3.1. Implementações em Hardware da Fase de Teste SVM

A implementação em hardware da fase de teste é muito mais simples do que a fase de treinamento. Isso é devido à natureza da classificação, que matematicamente nada mais é do que a função sinal de uma soma de produtos.

(GENOV; CAUWENBERGHS, 2003) introduziram uma arquitetura denominada Kerneltron, uma implementação híbrida analógica digital para a fase de teste da *SVM*. Esta realiza operações de multiplicação e acumulação (MAC) em paralelo. A arquitetura foi implementada em ASIC, onde o processador era composto por células internamente analógicas, porém com interface digital, onde era possível realizar cálculos paralelos através de soma de correntes. Assim a arquitetura ocupava uma área muito menor que o correspondente a um processador completamente digital.

Em 2005 foi proposta uma arquitetura paralela reconfigurável em FPGA (BIASI; BONI; ZORAT, 2005), denominada KBLAZE. A classificação da SVM neste trabalho é executada de acordo com a equação (13). A solução consiste em um SoC, composto por duas unidades MAC, uma unidade que executa o produto

escalar e um módulo kernel integrado ao *soft-processor* core da Xilinx Virtex II, o MicroBlaze. Este também é utilizado para supervisionar o sistema. Os dados que circulam entre estes módulos foram codificados no formato ponto fixo de 18 bits. A reconfiguração neste caso é estática e utilizada para explorar o paralelismo inerente à multiplicação de vetores.

Com o objetivo de reduzir o hardware necessário para a SVM foram propostas arquiteturas com sistemas numéricos logarítmicos, nas quais, multiplicadores e divisores são substituíveis por somadores e subtratores (KHAN; ARNOLD; POTTENGER, 2004; KHAN; ARNOLD; POTTENGER, 2005; BONI, ZORAT, 2006). Outro importante trabalho visando a redução de hardware foi proposto por meio de um algoritmo que evita o uso de multiplicadores (ANGUITA et al, 2006).

3.2. Implementações em Hardware da Fase de Treinamento SVM

A primeira proposta de arquitetura *hardware* para implementar a fase de aprendizado de uma SVM foi realizada em 1998 (ANGUITA; BONI; ROVETTA, 1998). Foi proposta uma versão analógica de SVM linear onde o problema da programação quadrática era resolvido por uma rede neural do tipo Hopfield. Em um trabalho subseqüente foi proposta uma solução digital para o problema (ANGUITA; BONI; RIDELLA, 1999), onde os autores mostraram que em condições específicas, multiplicadores podem ser substituídos por *shifters*, além de mostrar que as variáveis podem ser codificadas no formato ponto fixo. Entretanto, tiveram que contornar o problema da precisão dos cálculos: a resolução do problema da programação quadrática é resolvida por multiplicações matriciais e como o número de operações envolvidas é muito alto, o algoritmo que, na teoria, deveria convergir para a solução não o faz na prática por problemas de precisão. Os trabalhos que se seguiram (ANGUITA; BONI, 2001), (ANGUITA; BONI, 2002) e (ANGUITA; BONI; RIDELLA, 2002) apresentaram sucessivas melhoras na arquitetura digital e na convergência do algoritmo, através de simulações de arquiteturas VLSI.

Finalmente, em 2003, é proposta uma arquitetura digital implementada em FPGA (ANGUITA; BONI; RIDELLA, 2003). O trabalho resolve o problema da programação quadrática através de um algoritmo desenvolvido pelos autores, chamado FIBS, e traz também uma análise dos efeitos da quantização sobre o desempenho de uma SVM. A arquitetura é implementada em blocos executados paralelamente, no intuito de obter maior velocidade de execução ao custo de maior área ocupada. Dois circuitos foram implementados para análise no FPGA da XILINX Virtex II, dispositivo XC2V8000. O primeiro paraleliza 8 módulos, enquanto o segundo paraleliza 32 módulos ao custo de 1% e 6% de área ocupada no FPGA, respectivamente. Para validar o algoritmo foram realizadas simulações com o padrão de teste Sonar com 104 exemplos de treinamento e outros 104 exemplos para a classificação.

Todas as propostas acima apresentadas são baseadas em resoluções numéricas do problema da programação quadrática, portanto, utilizam multiplicação de matrizes. Por este motivo, as soluções acima citadas são inviáveis para casos com um grande número de vetores.

Ao contrário das propostas até aqui apresentadas (que utilizam resolução da PQ por métodos numéricos), uma resolução analítica foi realizada por (PEDERSEN; SCHOEBERL, 2006). Os autores aplicam o algoritmo SMO para a resolução da SVM em uma estrutura *hardware-software*. Na partição *hardware* do sistema foram incluídas as operações aritméticas de multiplicação e acumulação, que são consideradas temporalmente críticas para execução do algoritmo. A arquitetura foi baseada em um sistema de pipeline e multiplicadores de 64 bits. O restante do algoritmo é executado na partição software do sistema que faz uso de um processador Java otimizado (JOP) com freqüência de 100MHz. Comparações com a implementação inteiramente em *software* (executada no mesmo processador) conseguiram redução de tempo de até 60% (PEDERSEN; SCHOEBERL, 2006) O sistema foi treinado com uma aplicação com 60 exemplos, somente, utilizando *benchmarks* gerados pelos próprios autores.

As propostas até agora descritas são limitadas a poucos exemplos. Considerando que na maioria dos casos, os treinamentos de SVM utilizam grande número de exemplos, a substituição dos algoritmos implementados em software pelos algoritmos baseados em hardware não se torna atrativa.

O trabalho de (ACOSTA HERNANDEZ et al, 2009) é o primeiro a implementar o algoritmo SMO inteiramente em *hardware*. A arquitetura proposta utiliza um algoritmo SMO modificado para trabalhar com paralelismo, chamado *Multiple Pairs Sequential Minimal Optimization* (MP-SMO). A Figura 15 mostra a arquitetura VLSI utilizada no trabalho, que é composta por: a) um seletor que executa a tarefa de seleção dos pares a serem otimizados a cada iteração; b) "N"[9] otimizadores que executam a otimização dos coeficientes paralelamente (cada bloco executa a otimização de um par de multiplicadores lagrangeanos a cada iteração); c) "2N" blocos de kernel, que calculam as operações de kernel necessárias para a atualização global de dados (esta atualização é feita de modo similar ao descrito pela equação (37)); d) quatro blocos de memória que contêm os exemplos de entrada, as saídas do treinamento, os multiplicadores lagrangeanos e os valores da função auxiliar usada na seleção dos pares.

O projeto do bloco dos kernels neste trabalho foi baseada no kernel proposto por (ANGUITA et al, 2006) formado apenas por somadores e *shifters*, resultando em uma implementação mais simples e eficiente.

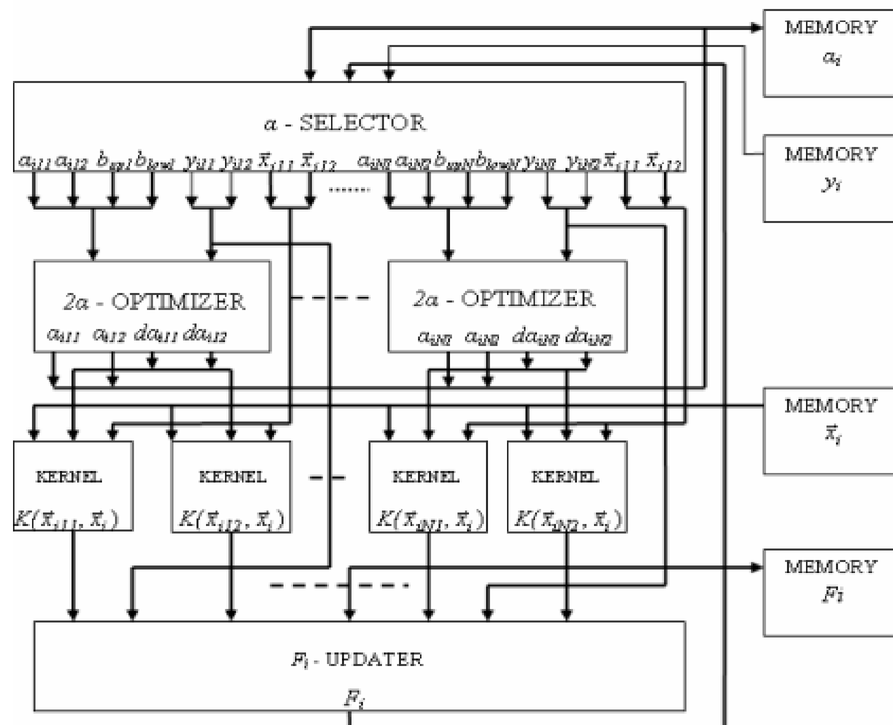


Figura 15 - Arquitetura VLSI da implementação MP-SMO

[9] "N" indica o número de pares de coeficientes de lagrange a serem otimizados paralelamente.

A arquitetura foi testada com o *benchmark* “Tic-Tac-Toe endgame”, um conjunto de treinamento contendo 958 exemplos, um conjunto consideravelmente maior do que os que foram utilizados anteriormente, apesar de que os dados considerados não serem em ponto fixo (tipo e tamanho dos dados deste problema consistem de 1 bit). A arquitetura visa, principalmente, aumentar o desempenho da solução de SVM, obtendo uma redução de até 76% do tempo de treinamento (utilizando N=4), comparada à solução de otimização de um único par, como foi apresentado na seção 2.2.3.

3.3. Proposta de metodologia para um SDR aplicado a SVM.

O único trabalho até agora desenvolvido, encontrado na literatura, tendo como objetivo a implementação de um algoritmo SVM em um SDR, é uma tese de doutorado escrita por (GONZALEZ, 2006). Nele, o foco principal foi a proposição de uma metodologia de particionamento do algoritmo, dentro de uma abordagem *top-down.* O trabalho organiza a proposta da metodologia em seis recomendações:

Gonzalez propõe uma metodologia de projeto voltada para a reconfiguração dinâmica do tipo *top-down.* O trabalho organiza a proposta da metodologia em seis recomendações:

a) Dividir as etapas de computação de um algoritmo em tarefas, avaliando o tempo e a forma de execução utilizando técnicas de *profiling* e *coverage*. Estimar a área de *hardware* baseando-se no número de operadores deste algoritmo.
b) Estabelecer um conjunto de tarefas com tempos de execução e suas interdependências, para assim determinar através de grafos ASAP e ALAP, o tempo mínimo de execução do conjunto e a mobilidade de cada tarefa.

c) Elaborar um esquema de reconfiguração preliminar[10] (prefetching) para otimizar o tempo mínimo de execução.
d) Definir duas áreas lógicas concorrentes e fisicamente intercambiáveis: a área de execução e a área de reconfiguração.
e) Os critérios de partição de uma FSMD devem ser baseados na redução do custo de comunicação, que pode ser definido como o número de conexões intrapartições e o número de vezes que o fluxo de dados do algoritmo atravessa essas conexões, previamente calculado através do perfil obtido por *profiling* e *coverage*.
f) Obtidas as partições temporais são definidas as partições espaciais (ou físicas). Caso a partição for maior do que a área disponível, esta será divida em subpartições (sendo permitido, inclusive, a migração de micropartições entre partições).

Essa metodologia já foi proposta tendo como base o modelo de solução analítica do algoritmo SVM, porém a análise foi feita a partir de uma descrição em alto nível, não tendo atingido uma implementação em nível RTL.

[10] Para isso seria necessário conhecer o tempo de reconfiguração de cada tarefa.

4. METODOLOGIA DE PROJETO

Neste capítulo serão apresentadas as arquiteturas digitais para a implementação do algoritmo SMO, com as adaptações necessárias ao algoritmo para a definição dos módulos do sistema.

4.1. Arquitetura básica do algoritmo SMO

A idéia básica para a implementação do algoritmo SMO em hardware é transformar as tarefas definidas na Seção 2.2.3 em blocos que executem estas tarefas. É importante para o projeto ser implementado de forma modular, tendo como objetivo o posterior particionamento temporal para a reconfiguração dinâmica. Este tema será abordado na seção 6.1.

O trabalho de (PLATT, 1998) apresenta o primeiro algoritmo do SMO encontrado na literatura, porém a heurística utilizada pelo autor é pouco eficiente e difícil de ser implementada em hardware. A heurística apresentada posteriormente por (KEERTHI, 2001), previamente explicada na Seção 2.2.3, acelera o processo de treinamento, além de facilitar a implementação em hardware.

As três tarefas principais executadas pelo algoritmo são seleção, otimização e atualização. Três blocos serão construídos para executar estas tarefas. Dentro das tarefas de otimização e atualização existe a função matemática do *kernel* que dentro do projeto possui a maior complexidade computacional e computa todos os dados de entrada. Enquanto que no primeiro a função *kernel* é chamada apenas uma vez para a computação do parâmetro η, na tarefa de atualização ela é chamada o número de vezes correspondente ao número m de exemplos de treinamento.

Devido a importância do *kernel* para o projeto, um bloco dedicado foi desenvolvido[11] para essa tarefa.

O cálculo do *kernel* dentro do bloco otimizador, caracterizado pelas equações (28) e (29), torna-se um empecilho para a execução do projeto. Primeiramente porque é executada através de divisão de ponto fixo por ponto fixo, o que demanda muitos recursos do ponto de vista de hardware. Adicionalmente, como explicado na Seção 6.1 o bloco *kernel* não estará disponível ao mesmo tempo que o bloco otimizador na arquitetura reconfigurável.

Para solucionar estes problemas, foi adotada a simplificação de substituir a variável η pela constante 2. Essa substituição foi possível, levando-se em conta os seguintes fatores:

a) A divisão por 2 é executada com um simples deslocamento para a direita, enquanto que no algoritmo original deveria ser executa uma divisão de ponto fixo por ponto fixo, que é uma operação complexa.
b) A substituição não irá afetar a taxa de erro, tendo em conta que o algoritmo sempre converge para uma solução ótima baseada na variável de tolerância ε. A única conseqüência será um pequeno aumento do número de iterações como será verificado na Seção 5.1.

Figura 16 apresenta de maneira simplificada a execução do algoritmo SMO com a divisão das tarefas mencionadas anteriormente representadas pelos retângulos cinzas. Dentro destes, as ações principais são apresentadas, com referência às equações descritas na Seção 2.2.3, com as adaptações mencionadas anteriormente.

Uma iteração SMO corresponde a uma seqüência das tarefas da figura. Pode-se notar que as computações dentro das tarefas envolvem alguns laços. Nos módulos Seletor e Atualizador, as computações são repetidas para cada um dos exemplos de treinamento, levando a m execuções. O laço do Kernel envolve d execuções uma vez que todos os d elementos de cada exemplo precisam ser processados.

---

[11] Explicação detalhada desta decisão de projeto na próxima seção.

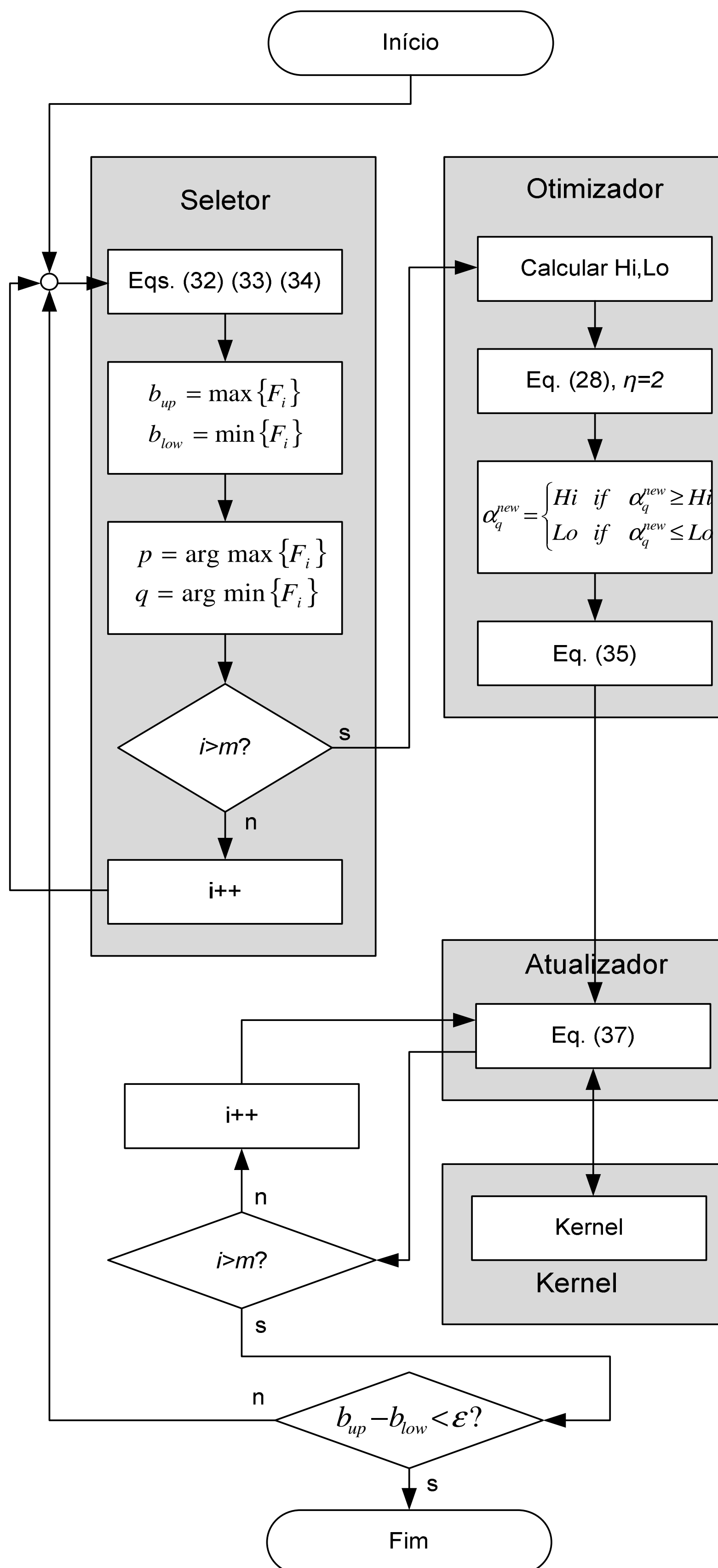


Figura 16 - Arquitetura modular

4.2. Aspectos de Implementação da função de kernel

A definição de um algoritmo claro, preciso e focado na implementação em hardware é fundamental para o desenvolvimento do projeto. O *kernel* é outro ponto fundamental a ser cuidado, pois é responsável por grande parte do processamento. Em geral, o *kernel* mais adotado nos trabalhos atuais é o Gaussiano, pois este é capaz de mapear as amostras de entrada em um espaço de características com dimensão muito maior, gerando assim uma alta capacidade de separabilidade. O *kernel* Gaussiano é definido pela equação (39).

$$K(\vec{x}_i, \vec{x}_j) = e^{-\frac{\|\vec{x}_i - \vec{x}_j\|^2}{2\sigma^2}} \tag{39}$$

Este *kernel*, apesar de ser o mais geral, traz dificuldades importantes para a sua implementação em hardware, pois a operação de exponenciação na base natural necessita de técnicas especiais, assim como as operações vetoriais envolvidas. Além disso, tal função *kernel* tenderia a consumir uma alta proporção do tempo e teria um custo de área elevado comparado com o resto da implementação.

Uma alternativa mais viável foi proposta em (ANGUITA et al, 2006), onde os autores apresentam um *kernel* semelhante ao Gaussiano, porém transposto para a base 2, tornando possível uma implementação com um algoritmo CORDIC (Coordinate Rotation Digital Computer). O *kernel* proposto é dado por:

$$K(\vec{x}_i, \vec{x}_j) = 2^{-\gamma\|\vec{x}_i - \vec{x}_j\|_1} \tag{40}$$

onde

$$\|\vec{x}_i - \vec{x}_j\|_1 = \sum_{s=1}^{d} |\vec{x}_{is} - \vec{x}_{js}| \tag{41}$$

lembrando que o cálculo deve ser feito para cada elemento de um dado exemplo de treinamento. O trabalho de Anguita et al. prova que este *kernel* é tão eficiente e preciso quanto o *kernel* Gaussiano original, através da realização de diversos testes com as duas versões.

Para a implementação em hardware neste trabalho, o algoritmo SMO foi reconstruído com estas considerações, cujo pseudo-código correspondente pode ser visto no Algoritmo 1. Se houver interesse por mais detalhes, o leitor poderá ver o pseudo-código do algoritmo todo no Apêndice A.

Algoritmo 1 - Pseudo-código da função kernel adotada

```
Para (s=1; s<=d; s++)   //Norma 1  // Número de elementos d
   nm1 = |x_is − x_js|;
   E1=E1+nm1;
Fim Para

E1=-E1* γ;  //γ = 2^p, p ϵ ℤ
F=fix(E1);
E1=E1-F;
B1=Δα;  //Δα = α_x^old − α_x^new

Se (E1 − log2(1 − 2^−1)) > 0
   E2 = E1;
   B2 = B1;
Do Contrário
   E2 = E1 − log2(1 − 2^−1);
   B2 = (1 − 2^−1) ∗ B1;
Fim Se

Se (E2 − log2(1 − 2^−2)) > 0
   E3 = E2;
   B3 = B2;
Do Contrário
   E3 = E2 − log2(1 − 2^−2);
   B3 = (1 − 2^−2) ∗ B2;
Fim Se

Se (E3 − log2(1 − 2^−3)) > 0
   E4 = E3;
   B4 = B3;
Do Contrário
   E4 = E3 − log2(1 − 2^−3);
   B4 = (1 − 2^−3) ∗ B3;
Fim Se

                ⋮
```

```
Se (En − log₂(1 − 2^(−n))) > 0 //n define o número de ciclos do algoritmo CORDIC.
   En = E(n-1);
   Bn = B(n-1);
Do Contrário
   En = E(n-1) − log₂(1 − 2^(−n));
   Bn = (1 − 2^(−n)) ∗ B(n-1);
Fim Se

k = Bn ∗ 2^F;
Retorna(k);
```

4.3. Definição de uma solução SVM genérica.

Os trabalhos correlatos apresentados na seção 3.2 apresentam arquiteturas bem adequadas aos problemas específicos que se desejam resolver (em termos da quantidade de elementos em cada exemplo e do tipo usado em cada elemento). O objetivo da arquitetura nesta dissertação, ao contrário das anteriormente desenvolvidas, é executar um treinamento genérico, ou seja, a arquitetura será capaz de treinar diferentes tipos de conjuntos de treinamento com um amplo número de elementos. Além disso, os elementos poderão ser tanto reais quanto binários.

Um exemplo de estratégia *ad-hoc* é o trabalho de (ACOSTA HERNANDES et al, 2009), que aborda uma solução específica, baseada no conjunto de exemplos a serem treinados. No *benchmark* utilizado pelos autores, denominado *tic-tac-toe*, foram utilizados $m = 958$ vetores de entrada (958 exemplos) que possuem dimensão $d = 27$ (27 elementos), com dados do tipo binário, por exemplo:

$\vec{x}_1$ 1 0 0 1 0 0 1 0 0 1 0 0 0 1 0 0 1 0 1 0 0 0 1 0 0 1 0

$\vec{x}_2$ 1 0 0 1 0 0 1 0 0 1 0 0 0 1 0 0 1 0 0 1 0 1 0 0 0 1 0

$\vdots$ $\vdots$

$\vec{x}_{958}$ 0 1 0 0 1 0 1 0 0 1 0 0 1 0 0 0 1 0 0 1 0 1 0 0 1 0 0

Com o hardware projetado para computar especificamente esse tipo de vetor de entrada, os registradores internos foram implementados com largura de 27 bits, assim como o tamanho de palavra dos somadores, multiplicadores, etc.

Considerando que deseja-se agora uma arquitetura que suporte o treinamento genérico, as seguintes medidas foram adotadas:

a) Codificação dos vetores de entrada em ponto fixo.

É um requisito necessário para a generalização da implementação, pois muitos conjuntos de treinamento apresentam dados decimais mais complexos. Os vetores são normalizados para que os valores de entrada variem entre 0 e 1. O número de bits para codificação será definido por um estudo apresentado posteriormente neste capítulo.

b) Serialização da operação de *kernel* para o cálculo com diferentes números de elementos.

Para lidar com exemplos de variadas dimensões *d*, uma solução é calcular as operações vetoriais em um laço interno. Com isso, o valor da dimensão do vetor deixa de ser um fator limitante para o treinamento.

O *kernel* utilizado neste trabalho é como descrito na Seção 4.2. Considerando as equações 40 e 41, a operação em hardware pode ser serializada, permitindo o cálculo do kernel para qualquer valor de d, conforme fluxograma da Figura 17. Alguns aspectos de implementação das operações dentro do *kernel* serão mostrados no próximo capítulo.

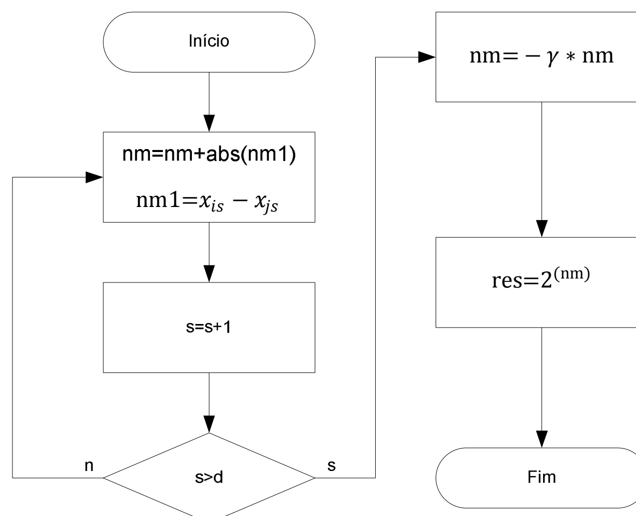


Figura 17 - Algoritmo para serialização do cálculo do *kernel*

c) **Desenvolvimento da arquitetura para realização das operações aritméticas em ponto fixo.**

As arquiteturas propostas em trabalhos anteriores contavam com uma estrutura específica para a o treinamento a ser executado. Já este projeto, conta com uma estrutura mais complexa, com todas as operações executadas em ponto fixo, porém sem grandes penalidades de área utilizada.

5. Implementação Plana

Neste capítulo apresentamos vários aspectos de implementação em hardware do algoritmo SMO. O foco neste capítulo é para uma implementação plana, isto é, sem uso de técnicas de reconfiguração dinâmica; as diversas decisões de projeto apresentadas aqui serão diretamente utilizadas no projeto com reconfiguração dinâmica, a ser apresentada no próximo capítulo. De início um estudo para a definição do tamanho da palavra a ser adotada no projeto é apresentado; a seguir são fornecidos alguns detalhes da implementação da função e módulo de cálculo do kernel. Na parte final do capítulo, a metodologia para a síntese RTL do circuito é descrita assim como os resultados de simulação com o algoritmo sintetizado.

5.1. Análise dos dados em ponto fixo

Um dos principais problemas para a implementação de SVM em hardware é o tipo e tamanho da representação numérica, que pode afetar a precisão dos cálculos. Na computação de propósito geral em software, as variáveis são armazenadas com precisão de 32 ou 64 bits em ponto flutuante, que é suficiente para evitar a maioria dos problemas de quantização.

Hardware dedicado, por outro lado, usa registradores de ponto fixo com precisão limitada. Nesta implementação, onde o erro é propagado no número de vezes correspondente ao número de exemplos a cada iteração, é de suma importância analisar os efeitos da quantização dos dados. Caso a quantização utilizada não seja adequada, o comportamento do algoritmo pode ser negativamente afetado.

A literatura técnica apresenta, até o momento, poucos trabalhos realizados sobre o assunto de quantização em SVM. Em (ANGUITA, BONI, RIDELLA, 2003), uma análise teórica foi feita para a fase de teste, que, infelizmente, não se aplica

bem à fase de treinamento que é o alvo deste trabalho. Posteriormente, um estudo com uma abordagem probabilística foi feito para o kernel Gaussiano (ANGUITA, BOZZA, 2005), que novamente não se aplica totalmente a arquitetura projetada neste trabalho.

Para estudar os efeitos específicos de quantização neste trabalho, algumas implementações foram testadas utilizando-se uma representação em ponto fixo de [6.k] bits onde o tamanho da palavra é de 6+k bits. A quantidade limitada de bits na parte inteira foi pré-definida de forma a não causar problemas, pois todos os dados de entrada são normalizados como descrito na Seção 4.3. Esta representação presta-se também ao armazenamento e computação de C e γ, considerando-se que, com os seus valores usuais, definidos pelo usuário, as operações dificilmente irão ultrapassar uma parte inteira maior do que a suportada pela codificação de 6 bits.

Para realizar os testes com diversas alternativas, foi feita uma implementação em MatLab baseando-se no algoritmo descrito no Apêndice A. Foram testados os diversos parâmetros arbitrários alterando valores mais comuns obtidos de trabalhos correlatos. Para comparação, os resultados obtidos são relacionados com a simulação em ponto-flutuante (solução em software), que seria a comparação ótima, se tratando de precisão.

Para testar os diferentes casos a variável de precisão ε foi fixada em um valor bastante utilizado, o de 0,001 (ACOSTA HERNANDEZ et al., 2009). A Tabela 1 apresenta os resultados do treinamento realizado em três benchmarks (coluna 1), cujo número de amostras (ou exemplos) e elementos é mostrado na coluna 2. Três representações em ponto fixo foram testadas: a representação de [6.14] bits nas colunas três e quatro, a [6.18] bits nas colunas cinco e seis e a [6.26] bits nas colunas sete e oito. Nas duas colunas seguintes, os resultados com ponto-flutuante são apresentados e, finalmente, as duas últimas colunas apresentam os parâmetros definidos pelo usuário para o treinamento executado. Estes parâmetros foram definidos em software por tentativa e erro, para determinar os melhores valores, ou seja os valores que minimizam as taxas de erro na fase de teste.

Tabela 1 - Erro de teste e número de iterações para um conjunto de benchmarks.

Benchmarks	m/d	**Hw 20-bit**		**Hw 24-bit**		**Hw 32-bit**		**Software**		**Parâmetros**	
		Erro	**#Itr.**	**Erro**	**#Itr.**	**Erro**	**#Itr.**	**Erro**	**#Itr.**	C	γ
Breast Cancer	569/30	2,86%	393	2,86%	379	2,86%	381	2,86%	340	10	0,125
Dermatology	358/132	0,86%	515	0,86%	472	0,86%	472	0,86%	466	1	1
Tic Tac Toe	958/27	N/C	N/C	5,68%	2572	5,68%	2568	5,68%	2481	3	0,5

Para cada caso, os resultados de erro da fase teste de classificação e o número de iterações necessárias para o treinamento foram obtidos. Para medir com mais precisão os valores de taxa de erro e de número de iterações, foi utilizado um método denominado *ten-fold cross validation* onde o conjunto de treinamento é divido em 10 partes, sendo 9 delas utilizadas para o treinamento e uma para a fase de teste de classificação; o processo é repetido 10 vezes de forma que todas as 10 partes sirvam para o teste. Este método é ilustrado na Figura 18, onde os retângulos brancos representam os conjuntos para o treinamento e os retângulos cinzas representam os conjuntos separados para teste. O valor final é definido pela média aritmética dos dez treinamentos, isso vale tanto para o valor de erro quanto para o número de iterações.

Analisando a Tabela 1, pode-se observar resultados interessantes. O primeiro destes é sobre a convergência. Para $0 < k < 14$ o algoritmo não converge para ε. Se tal parâmetro não for atingido, não é possível executar a fase de teste de classificação. Para os casos testados, somente o *benchmark tic-tac-toe* não convergiu para a codificação de 20 bits. Isso pode ser explicado pelo grande número de exemplos e iterações, que amplificam o erro em cascata.

Observe que para as implementações em hardware, o erro de teste de classificação não é afetado pela precisão da representação em ponto fixo, mesmo quando comparada com a solução utilizando ponto flutuante (software). Pode-se também observar que conforme a precisão aumenta, o número de iterações diminui, porém esse processo se estabiliza depois de 24 bits de codificação.

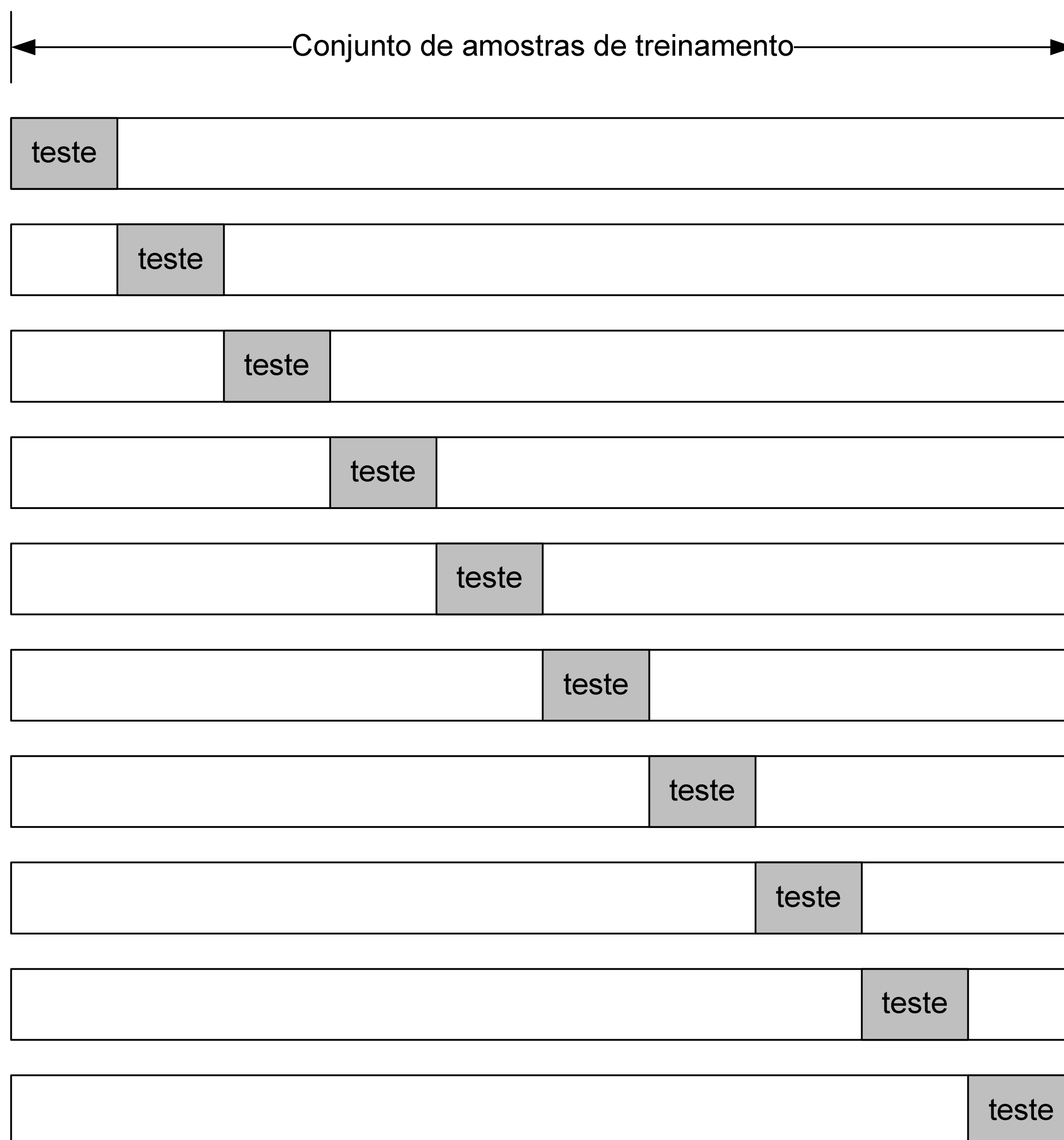


Figura 18 - *Ten-fold cross validation*.

Analisando os resultados, pode-se concluir que a codificação em 24 bits é a mais adequada para essa implementação, considerando-se que a codificação em 32 bits não traz nenhuma vantagem real para a execução do algoritmo, além de consumir mais recursos de área e memória do dispositivo.

5.2. A implementação da exponenciação

Como já exposto na Seção 4.2, o *kernel* utilizado neste trabalho foi inicialmente proposto por (ANGUITA et al., 2006). Para a exponenciação em hardware, um fator multiplicativo β foi adicionado e uma versão do seguinte algoritmo CORDIC foi proposta: Primeiramente a função do *kernel* é avaliada e reescrita como uma exponenciação em base dois de uma parte inteira mais uma parte fracionada:

$$K(\vec{x}_i, \vec{x}_j) = 2^{-\gamma\|\vec{x}_i - \vec{x}_j\|_1} = 2^{I+F} = 2^I 2^F \tag{42}$$

Para calcular a equação acima, basta, primeiramente, calcular o valor de $\beta 2^F$ e deslocar o resultado $-I$ vezes para a direita. O cálculo de $\beta 2^F$ é executado através de um algoritmo iterativo: Primeiramente B_1 e E_1 são inicializados com os valores de β e F, respectivamente. Em seguida na iteração $i+1$, B_{i+1} e E_{i+1} são definidos como:

$$B_{i+1} = B_i\left(1 + d_i 2^{-i}\right) \tag{43}$$

$$E_{i+1} = E_i - \log_2\left(1 + d_i 2^{-i}\right) \tag{44}$$

onde $d_i \in -1{,}0$ tal que $|E_{i+1}| \leq |E_i|$, garantindo assim a convergência do algoritmo. Pode-se demonstrar que se $E_i \to 0$ então $B_i \to 2^F$ (ANGUITA et al., 2006).

Este algoritmo pode ser facilmente implementado em hardware usando subtrações, um registrador de deslocamento e um comparador para calcular a equação (43), enquanto que a equação (44) pode ser implementada com o armazenamento dos valores pré-calculados de $\left|\log_2\left(1 + d_i 2^{-i}\right)\right|$ em registradores internos.

Anguita et al. projetaram a arquitetura para este *kernel* como exibido na Figura 19. Entretanto, com o objetivo de melhorar ainda mais a arquitetura

previamente proposta, uma pequena modificação foi executada: na arquitetura original a subtração que determina o valor de B_{i+1} é realizada somente após o cálculo de d_i. Para retirar esta dependência, a operação de multiplexação é executada somente após a subtração, paralelizando as execuções, como mostra a Figura 20.

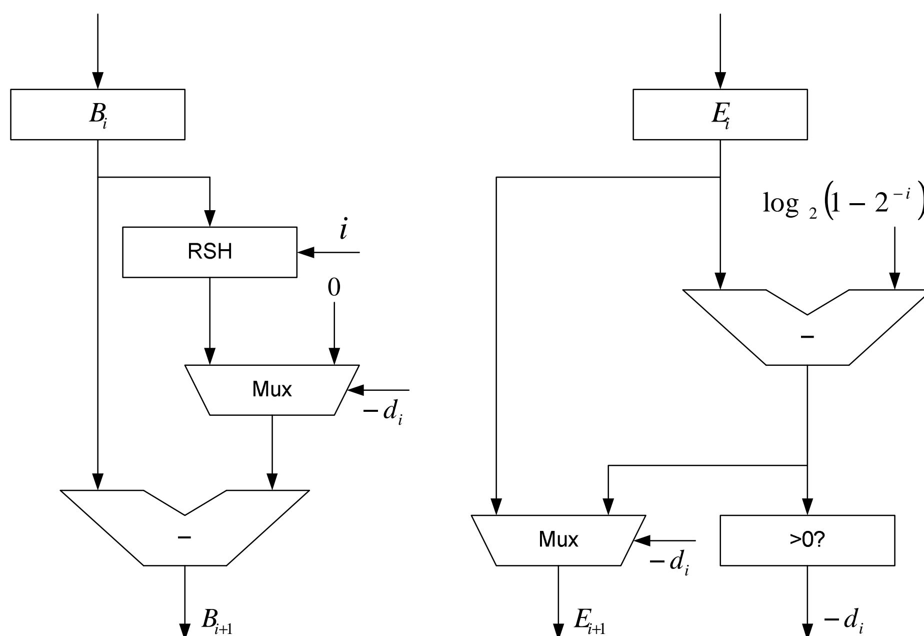


Figura 19 - Arquitetura de *kernel* proposta por Anguita et al.

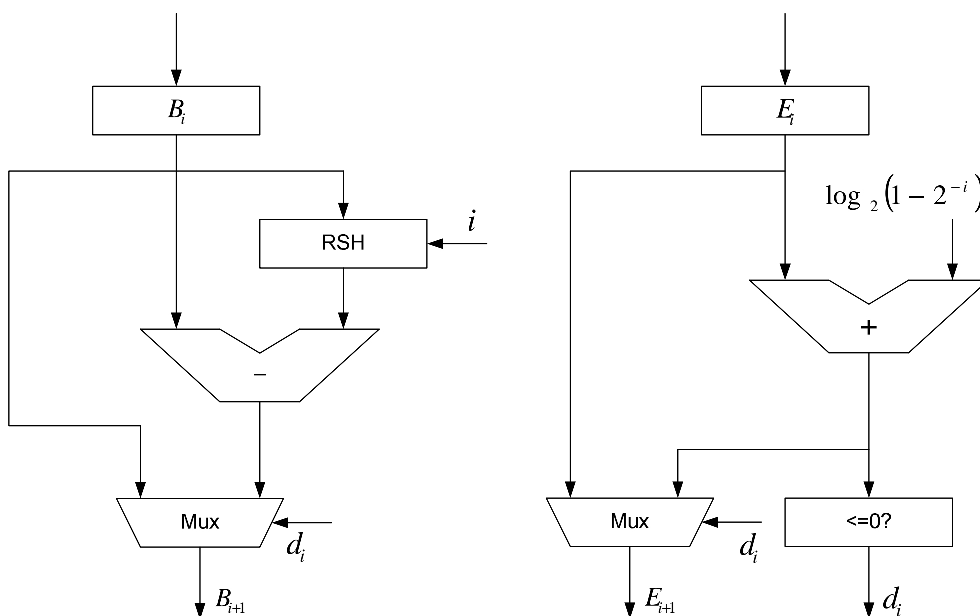


Figura 20 - Arquitetura de *kernel* modificada.

Os componentes de entrada foram codificados em ponto fixo, como mencionado anteriormente. Com 6 *bits* representando a parte inteira e k bits representando a parte fracionária, onde o peso do *bit* menos significativo é 2^{-k}. Experimentos independentes da arquitetura geral foram feitos com o objetivo de mensurar apenas o efeito de quantização do *kernel*. A arquitetura de kernel foi simulada em RTL e sintetizada, sendo os seguintes números de iterações $i = 8{,}16$ e 24 testados. A codificação k para cada caso adotado foi de $k = i$, pois a precisão requerida pela i-ésima iteração é $\left|\log_2\left(1 - 2^{-i}\right)\right|$. Pode-se notar que há uma conexão entre o número de iterações e a precisão da codificação. Analisando-se essa relação, conclui-se que valores de i maiores que valores de k não irão melhorar a precisão do kernel como demonstrado na equação (45):

$$\left|\log_2\left(1 - 2^{-(k+1)}\right)\right| \approx \left|-2^{-(k+1)}\right| < \left|-2^{-k}\right| \tag{45}$$

Os resultados da simulação e síntese podem ser observados na Figura 21 (dados de precisão) e na Tabela 2 (dados de área ocupada no FPGA). A arquitetura é duplicada, pois o *kernel* sempre computa um par de valores por vez na resolução do algoritmo SMO, como descrito na Seção 2.2.3.

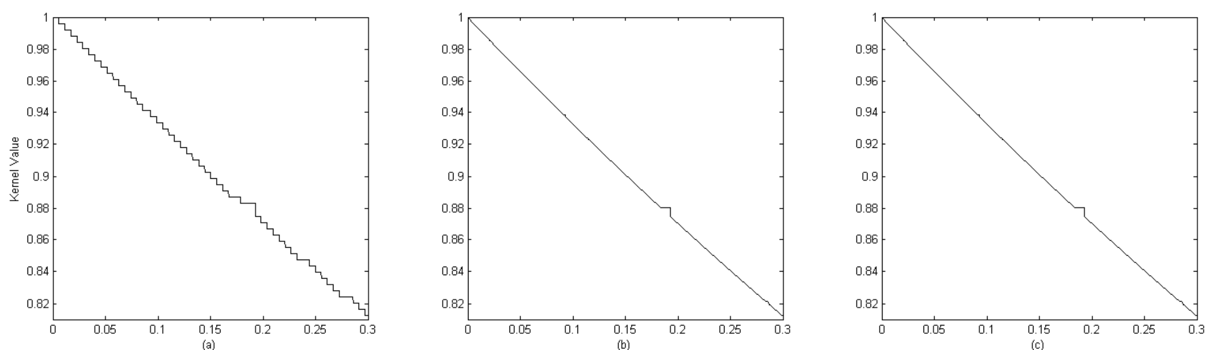


Figura 21 - Resultados de exponenciação para 8 (a), 16 (b) e 24 (c) bits e iterações no intervalo $0 \leq F \leq 0{,}3$.

Na prática, a precisão obtida com $i, k = 8$ é suficiente para gerar taxas de erros baixas. O efeito de serra obtido na Figura 21(a) é desprezível, tendo em conta

que os erros em cascata, descritos na seção anterior são maiores dos que os exibidos nesta figura.

Tabela 2 - Número de *Slices* ocupados no FPGA de acordo com a precisão do *kernel*.

i, k	**# Slices**
8	402
16	668
24	884

Considerando-se os efeitos de precisão e a área ocupada no FPGA, a solução com 8 iterações foi adotada. A codificação de ponto fixo fica mantida como descrito na seção anterior.

É importante considerar que a codificação adotada é muito interessante do ponto de vista da análise de efeito do erro, pois o menor valor possível para o *kernel* é 2^{-31} que é muito próximo de zero e não afeta significativamente o resultado final, assim como o maior valor possível é $2^{-(2^{-18})}$, que é muito próximo de 1, também não gera grandes problemas[12].

5.3. Fluxo de Projeto

Para a construção dos blocos de hardware, foi utilizada uma metodologia baseada em uma linguagem de alto nível, utilizando uma ferramenta denominada Bluespec SystemVerilog (BSV) (NIKHIL, 2004). O BSV é uma ferramenta para o projeto de circuitos digitais em ASICs e FPGAs que propõe uma abordagem particular para a síntese de alto nível. Ao invés de utilizar uma especificação baseada em linguagens de alto nível tradicionais, incluindo-se C, SystemC, etc. Esta

[12] Os valores mencionados são obtidos da codificação em ponto fixo [6.18]. Como a parte inteira é composta de 6 *bits* (1 *bit* para o *flag* de sinal e 5 *bits* para o valor em si) o maior número possível é $2^5 - 1 = 31$, assim como a parte fracionária é de 18 *bits*, sendo o menor valor possível $2^{-18} \cong 3{,}8147 \cdot 10^{-6}$.

ferramenta é baseada na semântica tradicional de descrição de hardware, mais especificamente, estruturada em SystemVerilog. Deste modo, o BSV explora conceitos avançados de software, traduzindo o comportamento do sistema para máquinas de estados finitos concorrentes, que é muito mais próximo do projeto de hardware. Além disso, o BSV aproveita o modelo de interfaces e instâncias do SystemVerilog para desenvolver o projeto de comunicação entre os módulos.

A descrição no BSV é desenvolvida através de FSMDs (máquinas de estados finitos com datapath) contendo operações complexas que o sintetizador posteriormente transforma em descrição de hardware. Ao final, o programa gera um arquivo Verilog, com descrição do circuito no nível RTL, que pode ser sintetizado em sintetizadores RTL como o programa ISE da Xilinx.

Na abordagem modular utilizada neste trabalho, cada bloco descrito no capítulo anterior foi concebido no BSV em System Verilog, para posterior síntese em blocos independentes em arquivos Verilog, como ilustrado na Figura 22.

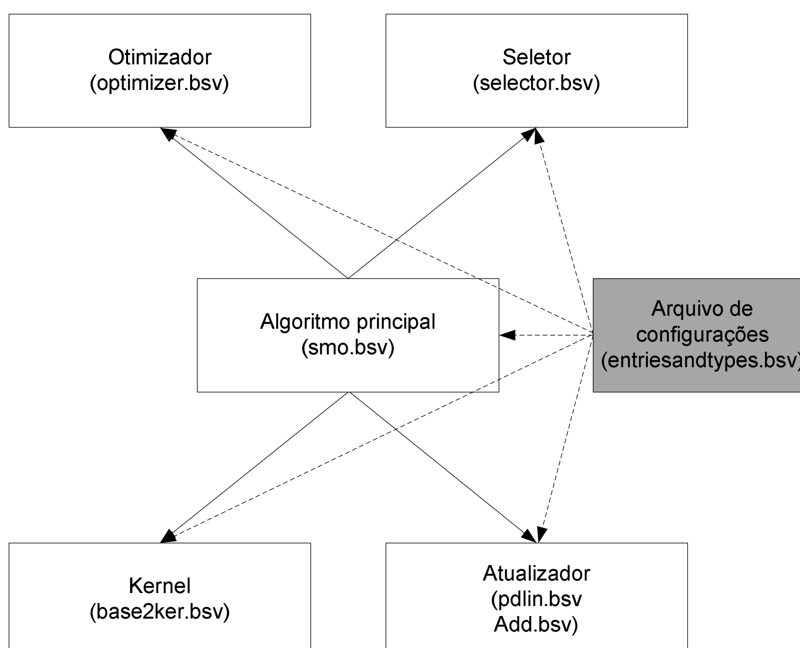


Figura 22 - Estrutura de arquivos adotada no BSV.

O algoritmo principal se comunica com todos os outros blocos, organizando a seqüência de tarefas, verificando as condições de execução dos demais blocos e controlando entrada, saída e armazenamento de dados das memórias. Os arquivos

conectados ao algoritmo principal executam as tarefas do algoritmo SMO como anteriormente explicado[13].

O arquivo de configurações contém todos os valores arbitrários, tais como os fatores γ e ε, além das constantes como o valor C. Este arquivo também determina o tipo de codificação que será utilizado em todos os outros blocos (ponto-fixo, números inteiros com ou sem sinal, etc.), além do tamanho das palavras utilizadas. Esta estrutura facilita os testes com diversos tipos de variáveis e codificações, e foi essencial para a execução de todos os testes em alto-nível para a escolha da codificação mais adequada, assim como os valores dos fatores e constantes. É importante ressaltar que o arquivo de configurações, apesar de ter a mesma extensão, não irá gerar código HDL, este simplesmente fornece as condições para a compilação dos outros blocos.

5.4. Simulações e análise de tempo

Para a análise de tempos, foram feitas simulações em nível RTL de algumas iterações, visando uma análise geral de tempo e área ocupada no FPGA. Para ambos os casos as simulações foram feitas tendo como dispositivo alvo o FPGA Xilinx Virtex-IV XC4VLX25, caracterizado por 10752 slices e 1296 Kbits de memória RAM.

As simulações foram executadas no programa ISE 9.2, desenvolvido pela Xilinx. Os blocos de memória foram carregados com os valores de entrada dos benchmarks. Como resultado da simulação, foi possível definir o número de ciclos necessários para execução de cada tarefa, e, conseqüentemente, o tempo total de execução real do algoritmo. A área total ocupada pela arquitetura plana foi de 1439 slices (13,4% da área deste FPGA), enquanto que a freqüência máxima de operação para este caso, definida pela síntese RTL, foi de 186,905 MHz.

---

[13] O modelo do atualizador foi subdividido em dois blocos para facilitar a posterior alocação de blocos fixos no FPGA.

A Tabela 3 mostra a forma de se estimar o número de ciclos para execução total de cada tarefa dentro de uma iteração do algoritmo SMO de acordo com o número de iterações associado. Pode-se notar que a atualização é a responsável por uma grande parte do processamento uma vez que depende do produto entre o número de exemplos e elementos. Os dados foram obtidos da simulação RTL.

Tabela 3 - Número de ciclos necessários para computar uma iteração SMO.

Tarefa	**Número de ciclos**
Seleção	$5m$
Otimização	17
Atualização	$m(9d + 13)$

Assim, calculando o tempo de execução do algoritmo em hardware, em função de *m* e *d*:

$$ti = \frac{m(9d + 18) + 17}{f} \quad (46)$$

Onde *ti* é o tempo de iteração e *f* é a freqüência de operação do circuito. Essa equação é importante, pois define o tempo de treinamento para qualquer conjunto de amostras.

A Tabela 4 mostra o tempo total de treinamento comparando a solução em FPGA em uma arquitetura plana, rodando sob o valor máximo de freqüência, com a solução em software, rodando em linguagem MatLab. A solução software foi obtido, rodando-se o programa em um computador pessoal com um processador Intel Core 2 Duo com um clock de 2.2 GHz e 4 GB de memória RAM.

Tabela 4 - Tempo total de treinamento na arquitetura plana.

	Breast Cancer	**Dermatology**	**Tic-Tac-Toe**
Software	10,44 s	13,10 s	109,2 s
Hardware	0,299 s	0.981 s	3.417 s
Aceleração	34,9	13,3	31,9

Como pode-se notar e era de esperar, o tempo de treinamento para implementação em hardware foi melhor em, pelo menos, uma ordem de grandeza.

6. Proposta de arquitetura e implementação para o SDR

Neste capítulo apresentamos a proposta de uma arquitetura dinamicamente reconfigurável para o algoritmo SMO, baseada nos resultados obtidos nos capítulos 4 e 5. De início, um estudo realizado para a definição das partições fixas e reconfiguráveis é apresentado; depois, a metodologia de síntese é apresentada, assim como os resultados de sua aplicação. Por fim, as metodologias de validação e de simulação são explicadas, assim como os resultados de implementação são apresentados.

6.1. Proposta de particionamento para a reconfiguração dinâmica

Para se analisar as possibilidades da implementação do algoritmo SMO em uma arquitetura dinâmica reconfigurável, um estudo preliminar de seus efeitos foi realizado. Neste estudo, aspectos de ocupação de área e penalidade de tempo foram computados com o dispositivo Xilinx Virtex-IV XC4VLX25.

Procurou-se neste estudo verificar a melhor configuração para a partição temporal do sistema, na distribuição entre partições fixas e reconfiguráveis. Assumiu-se que o particionamento já estaria feito e partições corresponderiam às tarefas definida nas seções 2.2.3 e 4.1. Na Tabela 5 são apresentados os números de CLBs por bloco e a porcentagem destes em relação ao total da área utilizada no FPGA. Estes dados foram obtidos a partir da síntese RTL do programa ISE da Xilinx.

Tabela 5 - Dados de área ocupada no FPGA

	Seletor	**Otimizador**	**Atualizador**	**Kernel**	**Outros**
Slices	103	322	196	402	416
% do total	7,16%	22,38%	13,62%	27,94%	28,91%

Analisando a Tabela 5 nota-se que os blocos que mais ocupam espaço no dispositivo são o Otimizador e o *kernel*. Baseando-se nisso, é interessante um particionamento temporal para que área do otimizador seja reaproveitada para implementar o *kernel* utilizando a mesma área. As tarefas continuam sendo executadas como ilustrado na

Figura 16. A execução das três tarefas caracteriza uma iteração do algoritmo SMO, como descrito na Seção 2.2.3, e o treinamento é finalizado ao final de todas as iterações.

Em uma implementação plana, com já visto anteriormente, as tarefas de seleção, otimização e atualização seriam executadas seqüencialmente dentro de uma iteração. Tal configuração é ilustrada novamente na Figura 23, para facilitar a compreensão das figuras posteriores.

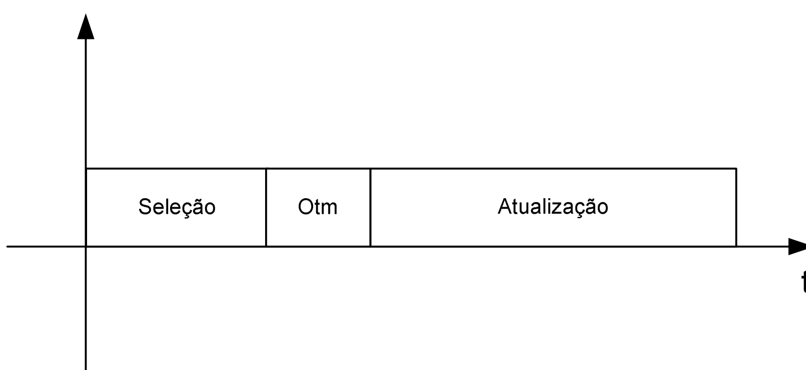


Figura 23 - Uma iteração da arquitetura plana.

Já na estratégia para uma arquitetura reconfigurável, o particionamento seria feito e executado como descrito na Figura 24, onde a seleção é executada em paralelo com a configuração do bloco otimizador; em seguida a atividade do FPGA passaria para a reconfiguração da área anteriormente utilizada pelo otimizador para implementar o bloco do cálculo do *kernel*. Com o bloco de *kernel* disponível, é possível executar a tarefa de atualização, e assim uma iteração é completada.

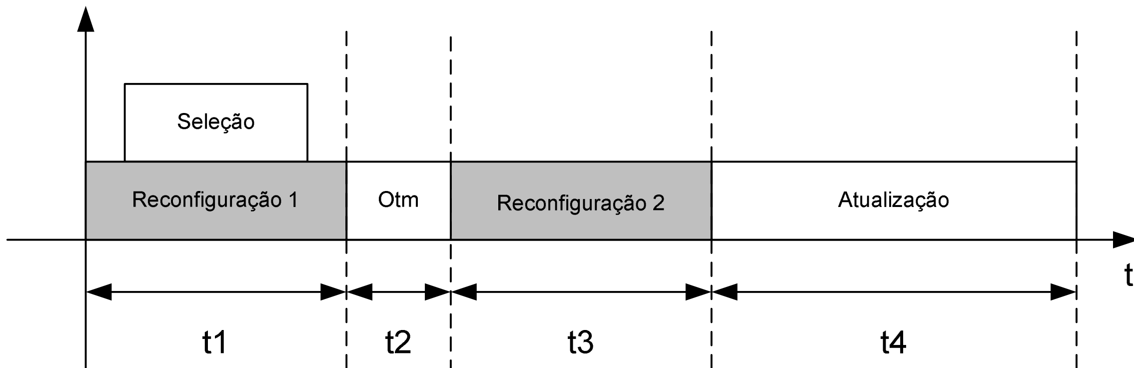


Figura 24 - Uma iteração da arquitetura reconfigurável.

A Figura 25 ilustra a alocação espacial das partições correspondentes aos instantes t1, t2, t3 e t4 da Figura 24. As diversas partições incluem as lógicas que executam as tarefas na forma indicada na Figura 16.

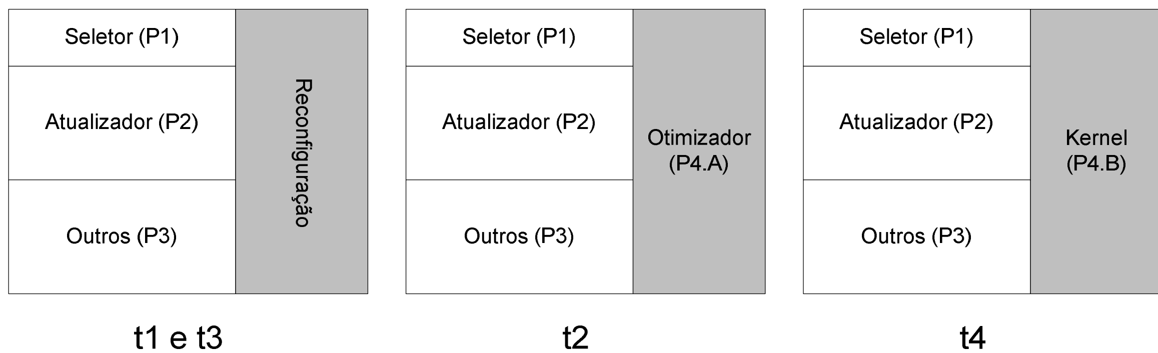


Figura 25 - Partições no DRFPGA ao longo do tempo.

Um problema que surge neste modelo de particionamento é que o módulo otimizador precisa do bloco de *kernel* para calcular o valor de α_q^{new} como descrito nas equações 28 e 29. Porém os dois blocos nunca estarão disponíveis ao mesmo tempo. Este problema foi solucionado utilizando-se a simplificação do *kernel* dentro do otimizador, como descrito na Seção 4.1.

Este esquema de reconfiguração permitirá uma economia de área de até 22,38% (área ocupada pelo bloco otimizador), dependendo da metodologia de síntese e implementação.

6.2. Análise da Viabilidade da Arquitetura Reconfigurável

Na seção 5.4, apresentamos a equação (46) para a estimativa de tempo de processamento de uma iteração da implementação plana do SVM. Na a análise da arquitetura reconfigurável, deve-se considerar que o processo de seleção ocorre paralelamente ao processo de reconfiguração, devendo-se descontar esse tempo do cálculo final. Assim o tempo de computação de uma iteração é reduzido em $5m$ (onde m é o número de exemplos do problema), com :

$$tir = \frac{m(9d + 13) + 17}{f} + 2tr \tag{47}$$

e

$$tr = \frac{\text{Bitstream}}{\#\text{Bytes}_{Prec} \cdot f_{Prec}} \tag{48}$$

onde tir é o tempo da iteração reconfigurável, tr é o tempo de reconfiguração, f é a freqüência de operação do circuito, Bitstream é a seqüência de bits usada na reconfiguração parcial, #Colunas relaciona a quantidade de colunas a serem reconfiguradas e por fim, $\#\text{Bytes}_{Prec}$ e f_{Prec} são o número de *bytes* e a freqüência do porto de reconfiguração, respectivamente. Na realidade, esta equação refere-se ao caso para o qual uma coluna inteira é reconfigurada, porém, os dispositivos Virtex-4 aceitam ser reconfigurados por frames de reconfiguração.

Pode-se achar o valor aproximado do *bitstream* desejado utilizando os dados do dispositivo obtidos do fabricante. No caso deste projeto, os dados específicos do

FPGA Virtex-IV XC4VLX25 foram obtidos de (XILINX, 2009). Os cálculos foram feitos considerando $f = 50MHz$ (para ambas arquiteturas), $f_{Prec} = 100MHz$, e levando em conta que a área a ser reconfigurada ocupa 412 slices (tamanho do kernel - maior bloco a ser reconfigurado) que ocupam 2 colunas deste dispositivo (cada coluna possui 384 slices), porém como explicado na próxima seção, os *bus-macros* ocupam um lugar considerável e o roteamento não é possível em um espaço menor do que 3 colunas neste caso.

Com os dados disponíveis, é possível calcular o valor adotando-se os seguintes procedimentos: primeiramente calcular a quantidade de *frames* de memória de configuração necessária para se reconfigurar uma coluna de CLBs, dado por $\text{Frame}_{\text{Coluna}}$ na equação abaixo (com arredondamento) [14]:

$$\text{Frame}_{\text{Coluna}} = \frac{\text{Frames}_{\text{rec}}}{\#\text{Colunas}} = \frac{5928}{28} \approx 212 \tag{49}$$

Na equação acima, $\text{Frames}_{\text{rec}}$ representam o número total de frames de reconfiguração do dispositivo. A seguir, procura-se obter a quantidade de *frames* de memória de configuração da região parcialmente reconfigurável (Partial Reconfigurable Region, PPR). No caso deste projeto a partição reconfigurável é implementada em 3 colunas, e o frame da coluna do FPGA alvo é de 212:

$$\text{Frame}_{\text{PRR}} = \text{Frame}_{\text{Coluna}} \cdot \#\text{Colunas} = 212 \cdot 3 = 636 \tag{50}$$

O próximo passo é calcular o tamanho do *bitstream* para reconfigurar a coluna da PRR, dada pela equação a seguir:

---

[14] Não confundir frames de memória de configuração com *frames* de reconfiguração. Os primeiros referem-se a blocos de memória interna no tecido do DRFPGA que são reconfigurados em uma única vez, enquanto, os últimos são os menores grupos de CLBs que devem ser reconfigurados na reconfiguração parcial.

$$\text{Bitstream}_{\text{FramePRR}} = \text{Frame}_{\text{PRR}} \cdot \text{Palavras}_{\text{Frame}} \cdot \text{Bytes}_{\text{Palavras}} = 636 \cdot 41 \cdot 4 = 104304 \text{ Bytes} \quad (51)$$

Para um cálculo mais preciso, deve-se considerar o *overhead* de configuração, que é um cabeçalho que sempre acompanha o *bitstream* quando a reconfiguração é executada. O overhead é dado em palavras e sua inclusão no bitstream é expresso pela equação (52):

$$\text{Bitstream} = \text{Bitstream}_{\text{Frame-PRR}} + \text{Palavras}_{\text{Overhead}} \cdot \text{Bytes}_{\text{Palavras}} = 104304 + 1312 \cdot 4 = 109552 \text{ Bytes} \quad (52)$$

Com o *bitstream* calculado, é possível obter o tempo de reconfiguração:

$$tr = \frac{109552 \text{ Bytes}}{4\,Bytes * 100\,MHz} \cong 274\ \mu s \quad (53)$$

A equação (47) implica que haverá uma penalidade de tempo em relação a arquitetura plana. Essa penalidade de tempo pode ser facilmente calculada como tir/ti (ti dado pela equação (46)).

Testes foram feitos com os três *benchmarks* relacionados anteriormente. A Figura 26 apresenta a penalidade de tempo versus o tempo de reconfiguração de FPGAs, variando de 0 a 548 µs. A linha tracejada no meio da figura indica o tempo de reconfiguração para o dispositivo alvo.

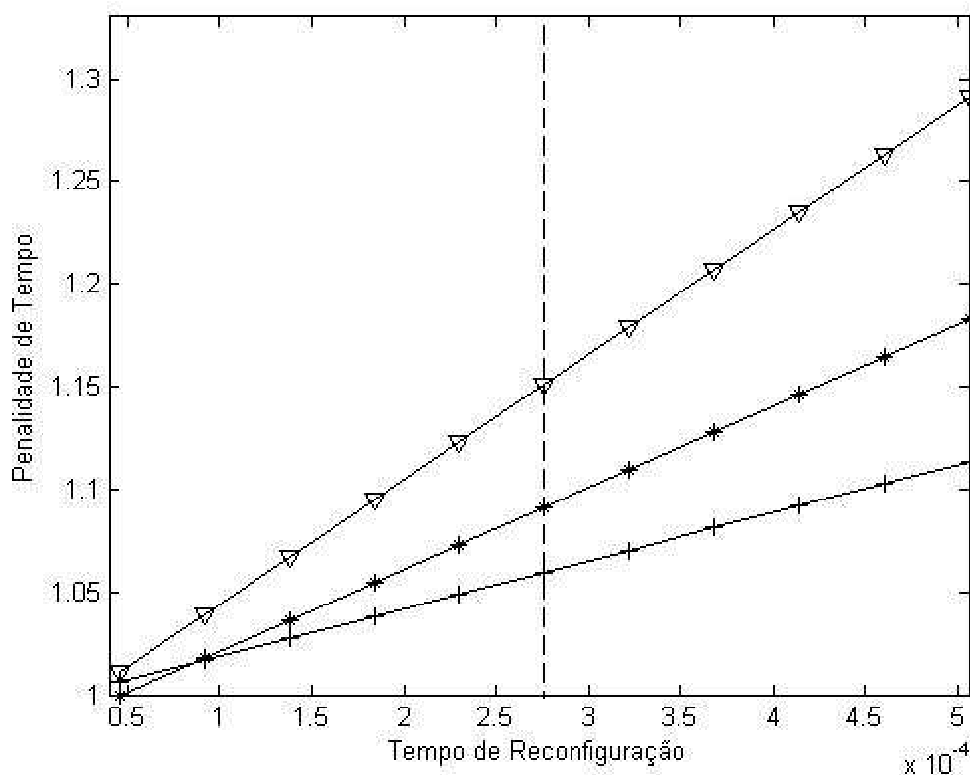


Figura 26 - Penalidade de tempo para $0 \leq tr \leq 502\mu s$.

Pode-se observar que a penalidade de tempo varia de acordo com o *benchmark*, tendo-se 16,57%, 6,53% e 10,08% de penalidade de tempo para os *benchmarks* Breast Cancer, Dermatology e Tic-tac-toe, respectivamente. A penalidade é aceitável para um ganho de área, e a execução do algoritmo continua sendo muito mais rápida quando comparada com a solução em software. A Tabela 6 mostra os tempos de execução na solução em sofware (primeira linha), o tempo na execução no hardware reconfigurável incluindo a penalidade de tempo, e a aceleração comparando as duas soluções para os três *benchmarks* testados.

Tabela 6 - Tempo total de treinamento na arquitetura reconfigurável.

	Breast Cancer	**Dermatology**	**Tic-Tac-Toe**
Software	10,44 s	13,10 s	109,2 s
Hardware	0,348 s	1,045 s	3,761 s
Aceleração	30	12,53	29,03

6.3. Metodologia de Síntese

A Metodologia de síntese utilizada neste projeto baseia-se no método Early Access. A Figura 27 descreve cada procedimento da metodologia.

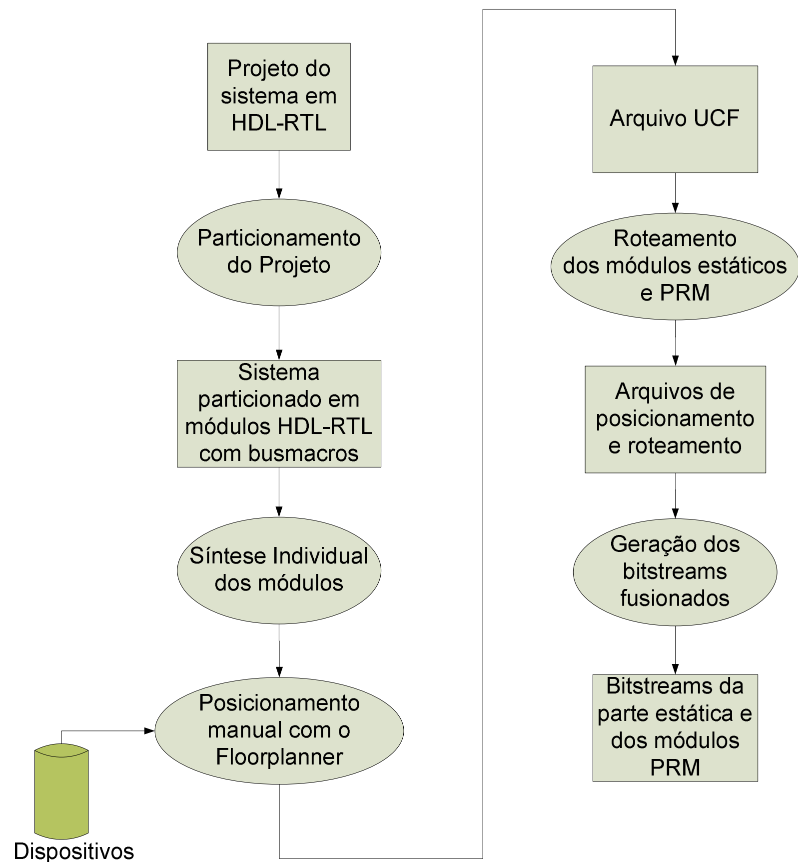


Figura 27 - Etapas da Metodologia Early Access

A seguir será detalhado cada etapa da metodologia ilustrada, aplicada ao projeto SVM:

- **Projeto em HDL-RTL:** Primeiramente os circuitos foram descritos em BSV como descrito na Seção 5.3, gerando as descrições HDL, preparando assim o material necessário para o primeiro passo da metodologia: o particionamento. Já pensando no circuito reconfigurável, a própria descrição em alto-nível foi desenvolvida para gerar os arquivos separadamente, como ilustrado anteriormente na Figura 22. Cada arquivo bsv gerou um arquivo verilog correspondente. O módulo *top* teve que ser modificado manualmente, pois o

BSV gera lógica no bloco principal, impedindo que se aplique a metodologia modular necessária ao Early Access (BOBDA, 2007), assim foi gerado o bloco adicional (bloco outros) com a lógica extra, com o módulo *top* descrevendo apenas os mapeamentos dos portos (*port maps*).

- **Particionamento do projeto e síntese individual:** Foi basicamente executado como descrito na seção 6.1. Os arquivos em *verilog* foram separados e sintetizados no programa Xilinx ISE individualmente. Os resultados de área resultantes foram apresentados na Tabela 5.
- **Posicionamento manual no *Floorplanner*:** O posicionamento foi feito utilizando o *Floorplanner* com a biblioteca do dispositivo Virtex-IV XC4VLX25. As áreas fixas e reconfiguráveis foram posicionadas, o pino de relógio, o DCM e o Buffer geral BUFG (que sincroniza o sinal externo de clock) foram também alocados. Também foram definidos os pinos de entrada e saída para os sinais correspondentes do DRFPGA. A alocação dos módulos foi feita tendo em consideração as conexões entre os módulos e as necessidades de memória de cada um. Com o arquivo UCF gerado, foi feita a edição do mesmo para inserir os bus-macros.
- **Roteamento dos módulos estáticos e reconfiguráveis**: O posicionamento e roteamento dos elementos posicionados são feitos nas regiões fixas e reconfiguráveis respectivamente com as funções dos scripts da Xilinx. Nesta seção o projeto foi verificado várias vezes e a cada erro o posicionamento manual era refeito, pois em algumas condições o roteamento não era possível. No caso da alocação reconfigurável, por exemplo, mesmo tendo necessidade de apenas 2 colunas para a implementação, a ferramenta não era capaz de implementar as conexões. O problema foi resolvido quando se aumentou a alocação para 3 colunas inteiras.
- **Geração dos *bitstreams***: Finalmente, após executar o roteamento, os *scripts* da Metodologia Early Access foram aplicados, e as partes estáticas foram fundidas com cada um dos módulos a serem incluídos inicialmente nas PRRs para gerar o *bitstream* de configuração inicial do DRFPGA.

O aumento do número de colunas para implementar as partições reconfiguráveis, descrito no item acima de roteamente, foi um problema imprevisto, gerado basicamente devido aos seguintes fatores:

- No SDR, os sinais dos módulos reconfiguráveis, otimizador e *kernel*, têm diferentes destinos e origens para a parte fixa do circuito. Nesta primeira versão da SVM com reconfiguração dinâmica foi tomada a decisão de realizar as conexões ponto-a-ponto por simplicidade de projeto, porém, isto implicou duplicação do número de *bus-macros*. O alto número de *bus-macros,* por sua vez, exigiu que se implementasse a partição reconfigurável em colunas inteiras (o único modo de conectar todas as *bus-macros* e ainda se avizinhar aos blocos com conexões).
- O fato de a ferramenta não executar o roteamento (por problemas práticos de roteamento com os *bus-macros*) em apenas 2 colunas (o que já seria um pouco maior do que a área estimada para o caso ideal, sem ser ponto-a-ponto) e executá-lo com 3. Isso aumentou em 50% a área necessária para a alocação das partições reconfiguráveis.

Estes fatores resultaram nos roteamentos ilustrados na Figura 28, onde pode-se perceber que não se aproveita totalmente o potencial de redução de área, pois apenas uma pequena área é compartilhada pelos dois módulos reconfiguráveis. Todavia, o potencial de economia de área ainda existe e pode ser aproveitado se uma redução do número de *bus-macros* for executada; com isso, tornaria-se possível implementar a PRR em um espaço que não ocupe a coluna inteira, e, consequentemente, aumentar a área compartilhada pelas duas partições reconfiguráveis. Isto poderia ser feito em futuras versões da implementação do SDR, através de circuitos de multiplexação no tempo ou de um projeto de bus, que coletem e direcionem sinais vindos da parte fixa do SDR.

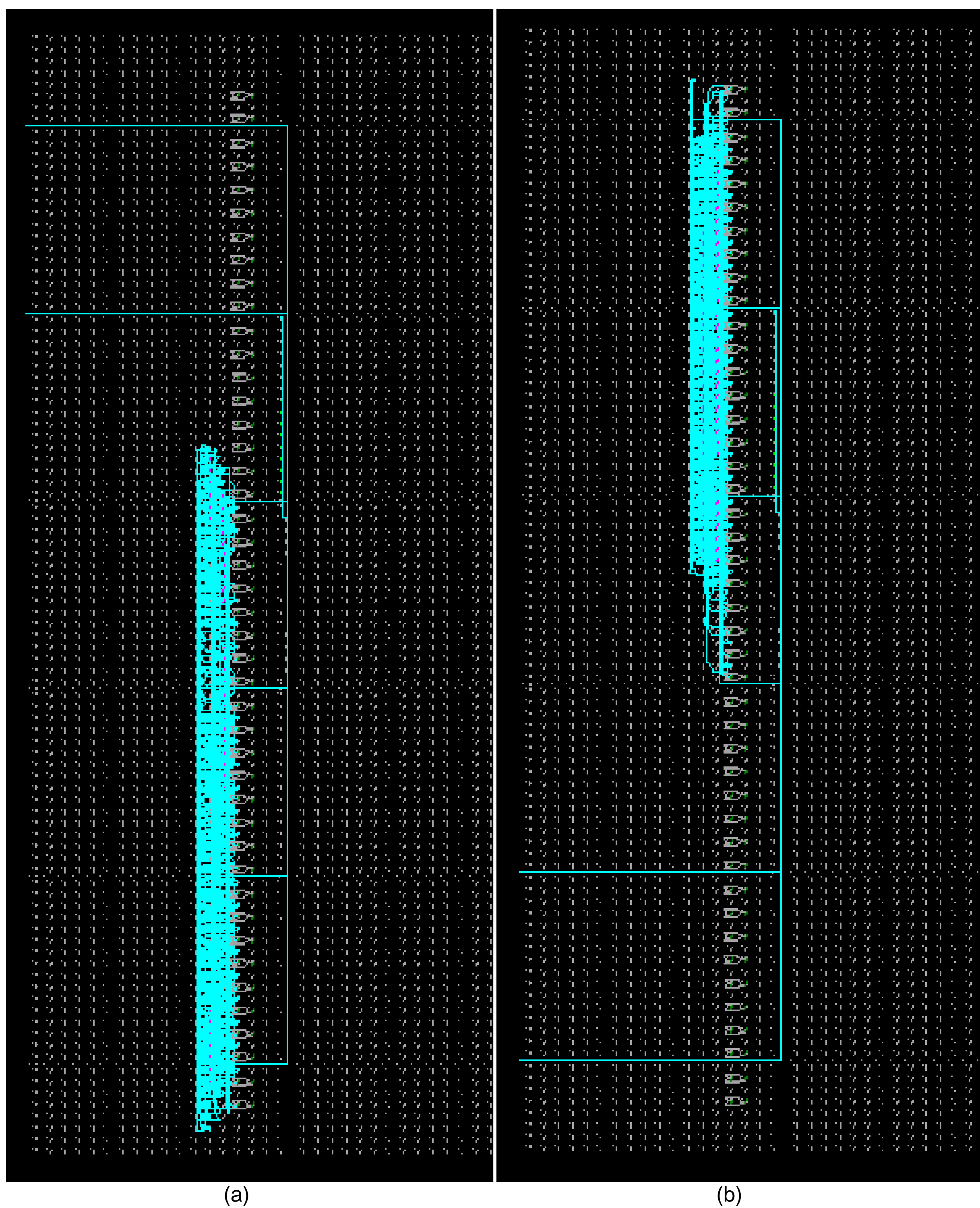


Figura 28 - Roteamento dos módulos reconfiguráveis: kernel (a) e otimizador (b)

Como resultado do resto da síntese, a Figura 29 mostra o roteamento de todos os módulos estáticos:

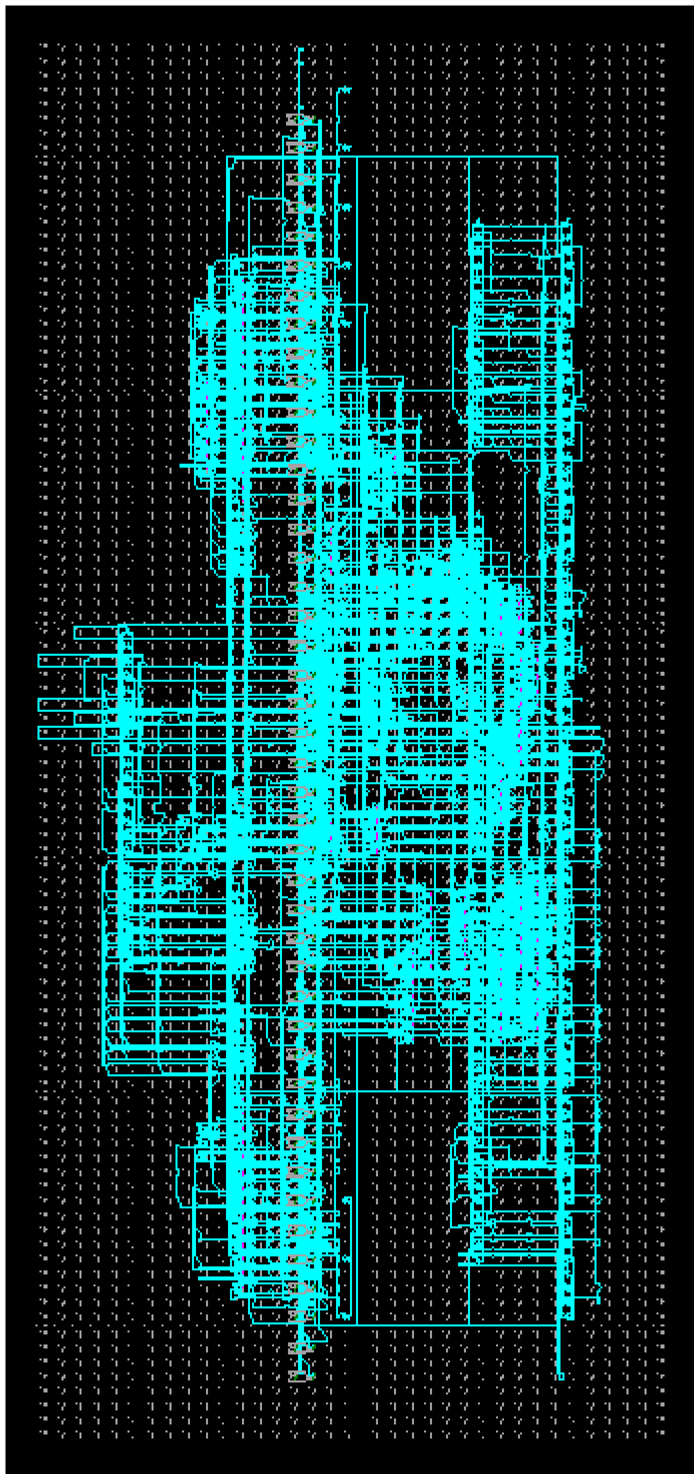


Figura 29 - Roteamento dos módulos estáticos.

6.4. Metodologia de Simulação por DCS.

Uma das técnicas mais eficiente para simulações de SDRs é denominada chaveamento dinâmico de circuitos (dynamic circuit switching, DCS) (LYSAGHT; STOCKWOOD, 1996; BRUNELLI, 2005; KOJIMA, 2007). Esta técnica emprega chaves de isolação entre os módulos fixos e reconfiguráveis, permitindo que os módulos reconfiguráveis fiquem ativos e em contato com a estrutura estática nos momentos adequados. A técnica também emula os tempos de reconfiguração do dispositivo, permitindo assim simular com exatidão os tempos, as dependências e a estrutura do SDR.

O funcionamento destas chaves virtuais pode ser melhor explicado quando se observa a Figura 30, que mostra quatro tarefas e a inclusão de quatro chaves de reconfiguração. No circuito, são adicionados dois módulos de controle de escalonamento que detectam as condições de reconfiguração, acionando ou não, as chaves.

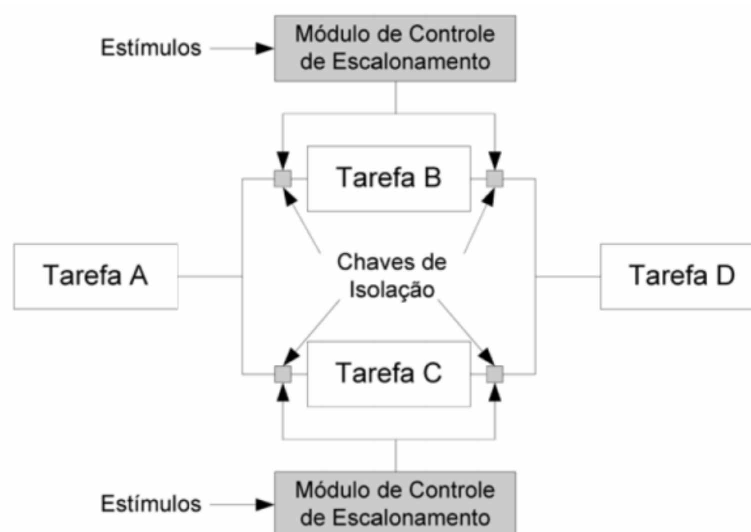


Figura 30 - Exemplo de simulação DCS.

As tarefas dinâmicas são agrupadas em grupos ativos e inativos, e conectadas as tarefas estáticas por meio das chaves de isolação. Estas podem assumir um dos seguintes estados: ativa, inativa ou em transição, como descrito na

Tabela 7, Onde "0" representa o zero lógico, "1" representa o um lógico, "X" representa uma valor desconhecido e "Z" representa estado de alta impedância.

Tabela 7 - Tabela da verdade das chaves de isolação.

Variável de Controle	Saída	Estado do conjunto
0	Entrada	Ativo
1	X	Transição
Z	Z	Inativo
X	Z	Inativo (adotado)

O tempo de transição é dado na simulação, e este deve ser baseado em cálculos a partir dos dados oferecidos pelo fabricante do dispositivo. Basicamente, o tempo de reconfiguração pode ser calculado como explicado na Seção 6.2 (equações (48) a (53)).

A Figura 31 exemplifica o funcionamento da técnica DCS (KOJIMA, 2007). Nele, dois módulos reconfiguráveis (porta E e porta OU) se alternam controlados por seis chaves de isolação (is1-is6), isolando todos os portos de entrada e saída dos módulos reconfiguráveis. A unidade de controle (*unit control*) ativa ou desativa as chaves de isolação, enquanto o módulo de teste simula e verifica os sinais de entrada e saída para verificar a funcionalidade do circuito.

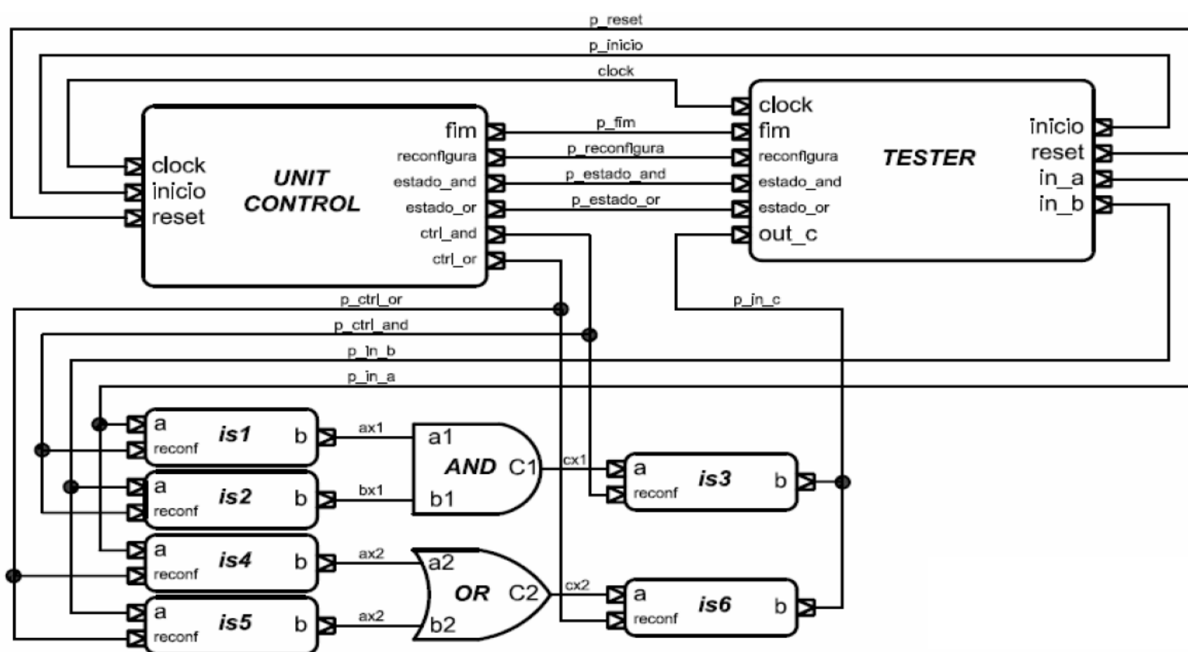


Figura 31 - Simulação DCS para portas lógicas E/OU.

As formas de onda deste circuito são exibidas na Figura 32 (KOJIMA, 2007), usada para ilustrar o funcionamento das chaves. Quando o sinal "reconfigura" assume nível lógico 1 os módulos são reconfigurados e a saída c assume estado de transição. Após a primeira reconfiguração o módulo OU fica ativo (controle OU em nível lógico 1), enquanto o módulo E fica inativo (controle E em alta impedância). Após a segunda reconfiguração o processo contrário ocorre. O sinal fim indica o término do processo de reconfiguração.

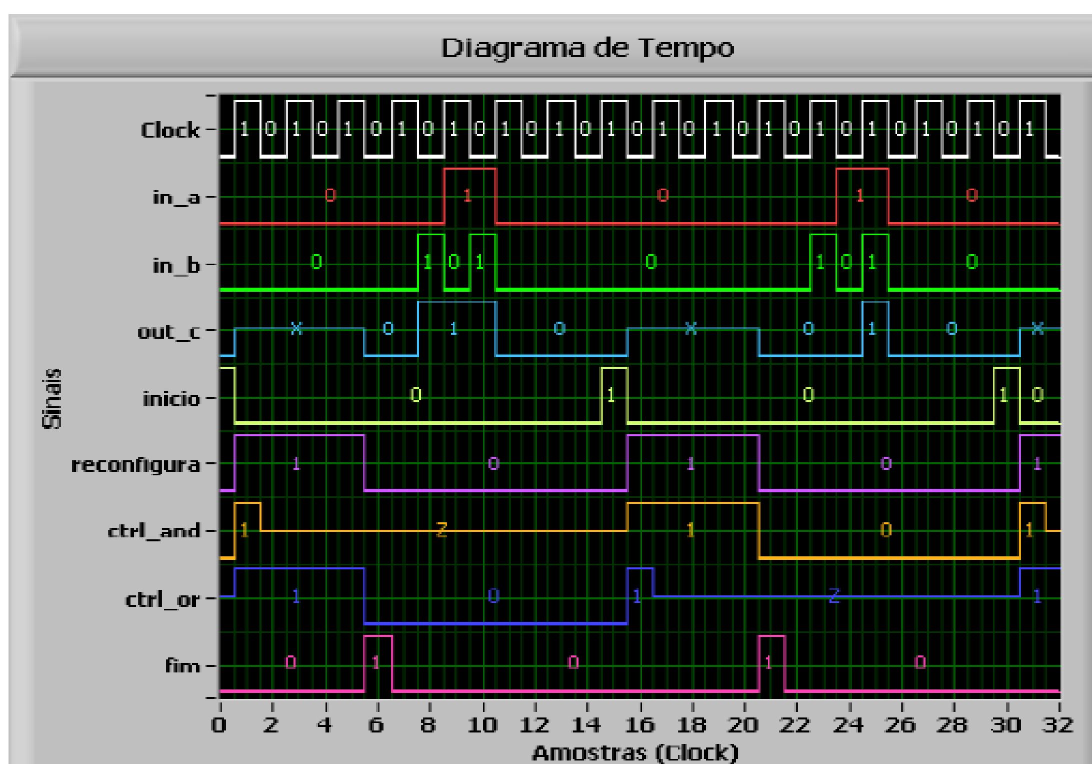


Figura 32 - Formas de onda da simulação DCS.

6.5. Simulação DCS aplicada a arquitetura SVM.

Para a aplicação da metodologia de simulação DCS a este circuito, as chaves de isolação devem ser adicionadas ao circuito, e a unidade de controle projetada. A unidade de controle foi baseada no particionamento temporal ilustrado na Figura 24. Sinais internos de controle foram utilizados para construir uma máquina de estados que irá controlar o funcionamento das chaves de isolação de acordo com a seqüência de execução dos blocos. A Figura 33 mostra a máquina de estados do controlador.

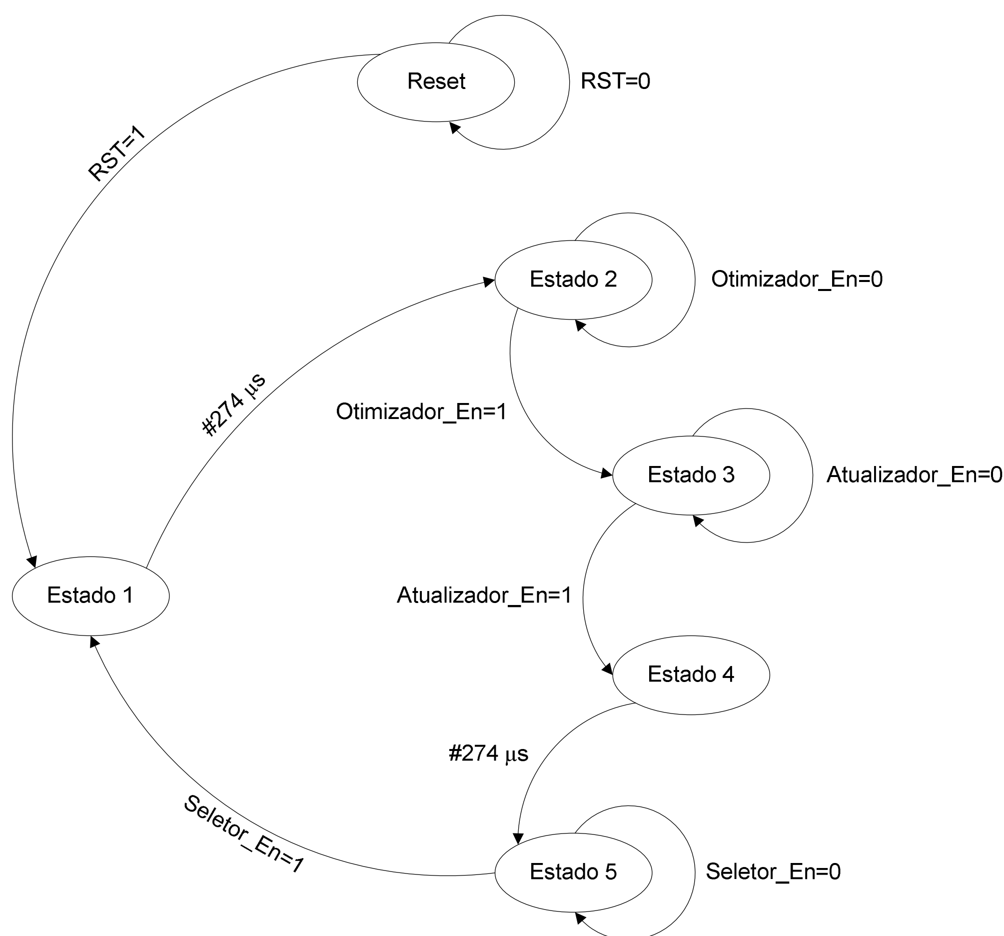


Figura 33 - Máquina de estados da unidade de controle.

Os sinais de *enable* foram utilizados como informação para saber que uma tarefa começa a ser executada, e conseqüentemente a outra foi finalizada. Por exemplo, o estado 3 ocorre enquanto se executa a tarefa de otimização, e só passa ao próximo estado quando a tarefa seguinte (atualização) ativa seu sinal de *enable*.

Os estados 1 e 4 definem o funcionamento das chaves de isolamento. No primeiro caso o controlador fecha as chaves do bloco otimizador e abre as chaves do bloco kernel depois de esperar o tempo de reconfiguração (transição do estado 1 para o 2), enquanto que no segundo caso a lógica das chaves é invertida após aguardar o *delay* de reconfiguração novamente (transição do estado 4 para o 5). A Tabela 8 descreve os eventos que ocorrem em cada estado, a segunda coluna descreve os acontecimentos que ocorrerão no estado relacionado na primeira coluna.

Tabela 8 - Eventos da máquina de estado da unidade de controle.

Estado	**Evento**
Estado 1	Alocação do módulo otimizador; Execução da tarefa de seleção.
Estado 2	Verifica se a tarefa de seleção foi executada para seguir ao próximo estado.
Estado 3	Execução da tarefa de otimização.
Estado 4	Alocação do módulo *kernel*.
Estado 5	Execução da tarefa de atualização.

Ao contrário do modelo apresentado na seção anterior, os blocos kernel e otimizador não têm entradas e saídas em comum, cada bloco tem seu conjunto de conexões com a lógica fixa e, por este motivo, ambas as conexões serão mantidas, como ilustrado na Figura 34. Esta decisão não afeta o funcionamento do circuito, pois mesmo que o nível lógico dos sinais conectados fiquem indefinidos enquanto as chaves ficam abertas, os blocos não estarão ativos no momento. Todavia, foram inseridos multiplexadores para cada sinal que se conecta a um *enable*, para que não se ativem tarefas em momentos inadequados. Em cada multiplexador se encontram o sinal normal de operação e um sinal 0 - a unidade de controle seleciona o sinal 0 do multiplexador sempre que o módulo em questão estiver inativo.

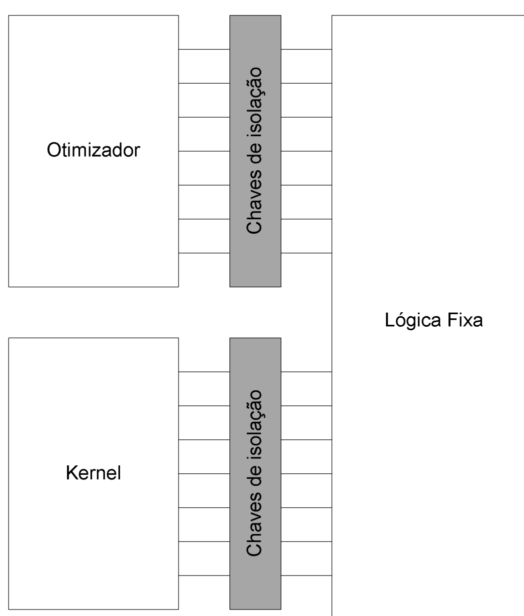


Figura 34 - Configuração das chaves de isolação.

6.6. Resultados da Simulação DCS

Nesta seção, são apresentados resultados referentes à simulação baseada no DCS. A Figura 35 mostra a evolução do algoritmo SMO nas quatro primeiras iterações. A primeira faixa superior verde representa o sinal de clock; em seguida, o sinal de iterações é exibido (variando de 1 a 4). Abaixo do indicador do número de iterações, está representado o estado da unidade de controle (variando nos padrões 1,4 e 5). Abaixo destes, sinais normais do circuito são exibidos para ilustrar o funcionamento da função DCS. Pode-se reparar que nos estados 1 e 4 da unidade de controle, os sinais ficam indefinidos (linhas vermelhas), pois os módulos estão sendo alocados (reconfigurados) de acordo com a sequência descrita na Tabela 8. Os estados 2 e 3 ocorrem em um intervalo muito curto de tempo e não são visíveis nesta figura. Estes estados são detalhados na Figura 36. Observa-se que os sinais normais entre as linhas vermelhas mostram a tarefa de otimização sendo executada (lembrando que a tarefa de otimização dura apenas 17 ciclos).

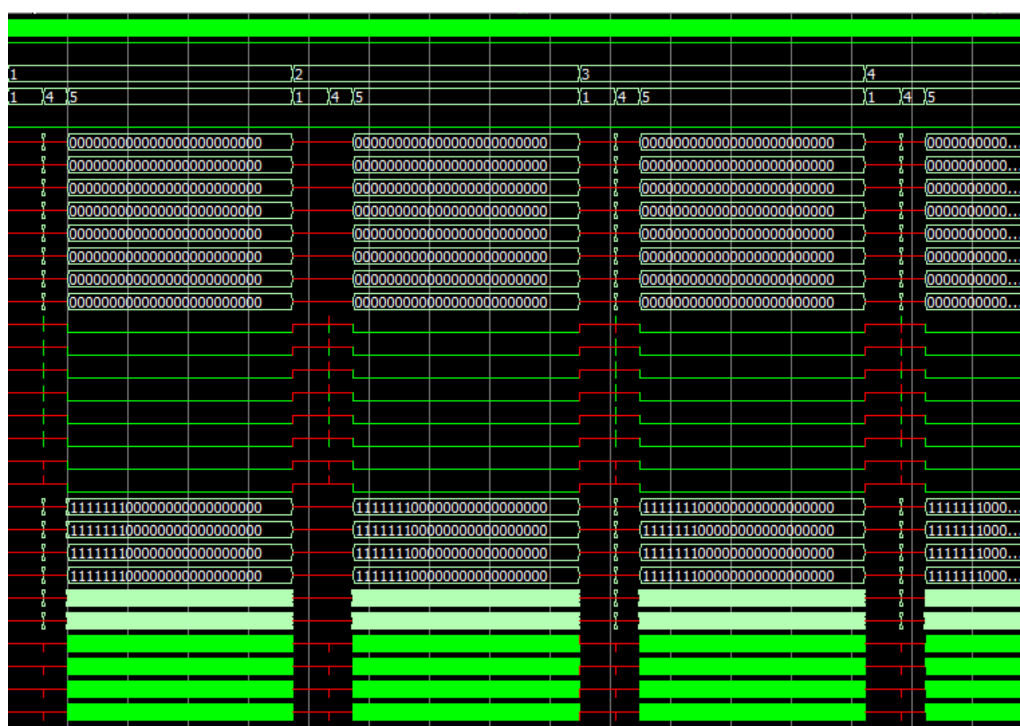


Figura 35 - Simulação DCS para arquitetura do SDR.

Figura 36 - Detalhe dos estados 2 e 3 da unidade de controle.

Para confirmar o correto funcionamento do algoritmo, a memória α foi verificada durante certos momentos da simulação, e os valores correspondem com os valores simulados em software, com um certo desvio, devido a precisão limitada de hardware.

A Tabela 9 mostra os primeiros 114 valores de α, obtidos de simulação, após a 359ª iteração do algoritmo para o *benchmark Breast Cancer.* A primeira, quarta e sétima colunas indicam o índice do alpha. A segunda, quinta e sétima colunas apresentam os valores de alpha obtidos em software através do *software* Matlab, e, por fim, a terceira, sexta e nona colunas mostram o valores de alpha obtidos em simulação de hardware.

É possível observar que os valores estão bem próximos, com poucas exceções. Os valores limiares (0 e 1) têm peso maior para o efeito final, e também se pode observar que todos os limiares estão iguais nas duas simulações.

Tabela 9 - Valores da memória α depois da 359ª iteração.

alpha	software	hardware	alpha	software	hardware	alpha	software	hardware
alpha 1	0,485226	0,485214	**alpha 39**	0	0	**alpha 77**	0,039133	0,039967
alpha 2	0	0	**alpha 40**	0	0	**alpha 78**	0,077399	0,078812
alpha 3	0	0	**alpha 41**	0	0	**alpha 79**	0,0028	0,003597
alpha 4	0	0	**alpha 42**	0,376384	0,376236	**alpha 80**	0	0
alpha 5	0	0	**alpha 43**	0	0	**alpha 81**	0	0
alpha 6	0	0	**alpha 44**	0	0	**alpha 82**	0,77104	0,770195
alpha 7	1	1	**alpha 45**	1	1	**alpha 83**	0	0
alpha 8	0	0	**alpha 46**	0	0	**alpha 84**	0	0
alpha 9	0	0	**alpha 47**	0	0	**alpha 85**	0,451418	0,452122
alpha 10	0	0	**alpha 48**	0	0	**alpha 86**	0	0
alpha 11	0	0	**alpha 49**	0,105389	0,105087	**alpha 87**	0	0
alpha 12	0	0	**alpha 50**	0	0	**alpha 88**	0	0
alpha 13	0,06502	0,064938	**alpha 51**	0	0	**alpha 89**	0,34595	0,34539
alpha 14	0,205712	0,205975	**alpha 52**	0	0	**alpha 90**	0	0
alpha 15	0,001118	0,001564	**alpha 53**	0	0	**alpha 91**	0	0
alpha 16	0	0	**alpha 54**	0,886479	0,88752	**alpha 92**	0	0
alpha 17	0	0	**alpha 55**	0	0	**alpha 93**	0	0
alpha 18	0	0	**alpha 56**	0,072168	0,070332	**alpha 94**	0	0
alpha 19	1	1	**alpha 57**	0	0	**alpha 95**	0	0
alpha 20	0	0	**alpha 58**	0	0	**alpha 96**	0	0
alpha 21	0,398285	0,398773	**alpha 59**	0	0	**alpha 97**	0,165778	0,165195
alpha 22	0	0	**alpha 60**	0,048211	0,048203	**alpha 98**	0	0
alpha 23	0	0	**alpha 61**	0	0	**alpha 99**	0	0
alpha 24	0	0	**alpha 62**	0	0	**alpha 100**	0,374351	0,375423
alpha 25	0	0	**alpha 63**	0	0	**alpha 101**	0	0
alpha 26	1	1	**alpha 64**	0	0	**alpha 102**	0	0
alpha 27	0	0	**alpha 65**	0,063225	0,064301	**alpha 103**	0	0
alpha 28	0	0	**alpha 66**	0,195533	0,198727	**alpha 104**	0,032075	0,032078
alpha 29	0,608089	0,605774	**alpha 67**	0	0	**alpha 105**	0	0
alpha 30	0,187572	0,18774	**alpha 68**	0,395877	0,395214	**alpha 106**	0	0
alpha 31	0	0	**alpha 69**	0,585773	0,585674	**alpha 107**	0	0
alpha 32	0	0	**alpha 70**	1	1	**alpha 108**	0	0
alpha 33	0	0	**alpha 71**	0	0	**alpha 109**	0,391686	0,392094
alpha 34	0	0	**alpha 72**	0,082307	0,082283	**alpha 110**	0	0
alpha 35	0	0	**alpha 73**	0	0	**alpha 111**	0,921529	0,921501
alpha 36	0,231055	0,230743	**alpha 74**	1	1	**alpha 112**	0,85913	0,857994
alpha 37	0	0	**alpha 75**	0	0	**alpha 113**	0	0
alpha 38	0	0	**alpha 76**	0	0	**alpha 114**	0	0

7. CONCLUSÃO DO TRABALHO

Esta dissertação apresentou uma nova arquitetura genérica para o treinamento SVM, assim como uma arquitetura reconfigurável. A seguir listamos as principais contribuições realizadas neste trabalho, as conclusões que pudemos obter após todo o desenvolvimento do trabalho e sugestões para a continuação do trabalho.

7.1. Contribuições

Entendemos que o trabalho contribuiu para aumentar a compreensão das atividades envolvidas no projeto de SDRs, assim como para a implementação em hardware de SVMs. Listamos as principais a seguir:

- A criação de uma arquitetura plana genérica, e implementação RTL, do algoritmo SVM para o treinamento de diversos tipos de conjuntos de amostras de entrada e para qualquer número de elementos do conjunto de entrada. A generalização do número de elementos foi possível através da criação de um laço interno para o cálculo da norma-1 dentro do bloco de *kernel*.
- O estudo sobre os efeitos de codificação em ponto-fixo para a implementação do algoritmo SMO em hardware. O estudo foi feito em alto-nível, e provou qual seria a codificação mais eficiente para desenvolver a arquitetura.
- A Modificação e implementação da arquitetura de *kernel* proposta por Anguita et al. Com este trabalho se definiu o número de iterações

necessário para executar a exponenciação dentro do bloco de kernel e a modificação contribuiu para acelerar esta operação que é a mais executada dentro do algoritmo.

- Desenvolvimento de uma arquitetura reconfigurável. Partindo da implementação plana, foi feito o particionamento, o estudo da viabilidade, calculando os tempos e a penalidade de reconfiguração. Depois disto a metodologia Early Access foi seguida para gerar os *bitstreams* de configuração/reconfiguração do FPGA.
- Validação da arquitetura reconfigurável, através de simulação DCS. Foi desenvolvida uma unidade de controle para administrar as chaves de isolação, e os demais sinais de controle.
- Na solução aqui descrita, o treinamento é realizado com conjuntos de amostras muito maiores que nos trabalhos anteriores, e é a primeira em hardware em que pode treinar diversos tipos de *benchmarks*. É também a primeira implementação SVM com arquitetura reconfigurável.
- A dissertação resultou em três trabalhos submetidos, aprovados e a serem publicados nos anais do XVI Iberchip e no VI Southern Programmable Logic (ambos em 2010). Os trabalhos intitulados "A General-Purpose Hardware Implementation of a SVM-SMO Algorithm" e "An FPGA Implementation of a Kernel Function for Support Vector Machines" tratam da implementação genérica SMO e da modificação e implementação do *kernel* proposto por Anguita et al. Ambos foram aceitos para apresentação oral no XVI Workshop Iberchip. Já o trabalho "A General-Purpose Dynammically Reconfigurable Svm" trata da arquitetura reconfigurável e foi aceito para apresentação oral na SPL 2010.

7.2. Conclusões

Após a realização desta dissertação, pudemos nos debruçar sobre a metodologia utilizada e os resultados obtidos, concluindo-se o seguinte:

- A aplicação de técnicas de reconfiguração dinâmica ao exemplo SVM, mostra que a implementação de SDRs continua sendo uma estratégia atrativa, por prover ganho de área, desde que a penalidade de tempo seja absorvível. O projeto que realizamos mostra, porém que existem ainda muitas tarefas manuais no fluxo de projeto de SDRs, como o particionamento e o *floorplanning*, que o torna longo e cansativo.
- Na implementação de SDRs, deve-se ter como foco minimizar o número de linhas utilizadas no *bus-macro*. Na versão da arquitetura SDR para o algoritmo SVM desta dissertação este objetivo não foi colocado, criando uma área extra que retirou algumas vantagens da estratégia de reconfiguração dinâmica.
- Para a implementação de SDRs, deve-se analisar a penalidade de tempo no ambiente de aplicação. A implementação de SVM pôde ser realizada com sucesso para reconfiguração dinâmica, mesmo considerando-se que a cada iteração do algoritmo havia uma penalidade pelo tempo de reconfiguração (quando não havia processamento efetivo de outras tarefas), uma vez que havia uma seqüencialidade favorável, além de uma grande quantidade de laços internos.
- A simulação por DCS continua sendo uma estratégia fundamental para validar o comportamento do circuito sob reconfiguração dinâmica, porém, requer muito trabalho manual por parte do projetista.

7.3. Trabalhos Futuros

Atividades futuras sugeridas para a arquitetura SMO em geral:

- Desenvolver uma estrutura de comunicação com memória externa, utilizando ferramentas como o EDK. Isso irá permitir que se possa processar muito mais amostras de treinamento e o fator de memória não será mais um limitante de generalização. Além disso, o fato de não se utilizar as memórias internas facilitará a alocação de partições no *FloorPlanner*, facilitando, inclusive, o projeto de arquitetura reconfigurável.
- Modificar a estrutura do atualizador, que é a parte que mais consome tempo na execução do algoritmo. O laço interno do kernel poderia ser paralelizado.

Atividades futuras sugeridas para arquitetura SMO reconfigurável:

- Projetar um *bus* de comunicação para que não seja mais necessário duplicar as saídas na parte reconfigurável do circuito. Isso irá otimizar a área utilizada e reduzir o número de *bus-macros*, o que irá permitir que se implemente a arquitetura reconfigurável em frames, e não somente em colunas inteiras.
- Implementar a arquitetura reconfigurável em hardware, utilizando o Microblaze para carregar os *bitstreams*, armazenados na memória externa da placa de desenvolvimento a ser utilizada, no DRFPGA para fazer a reconfiguração e utilizar o porto ICAP.

Atividades futuras sugeridas para projeto de SDRs em geral:

- Efetivo desenvolvimento de ferramentas de particionamento que facilitem a análise do overhead de tempo e de área.

- Ferramentas de posicionamento que evite ao projetista elaborar a tarefa manual de montar toda a planta-baixa do projeto dentro da metodologia *Early Access*.
- Desenvolver ferramentas de apoio que gerem automaticamente estruturas para simulação baseadas na técnica DCS.

REFERÊNCIAS

ACOSTA HERNANDEZ, R.; STRUM, M.; WANG, J. C.; GONZALEZ, J. A. Q. **A VLSI implementation of the the SVM training,** XV Workshop Iberchip, Buenos Aires, Argentina, pp. 204-209, 2009.

ANGUITA, D.; RIDELLA, S.; ROVETTA, S. **Circuital implementation of support vector machines.** Electronics Letters, 34**,** pp. 1596-1597, 1998.

ANGUITA, D.; BONI, A; RIDELLA, S. **Learning algorithm for nonlinear support vector machines suited for digital** VLSI. Electronics Letters, Vol. 35, No. 16, Agosto, 1999.

ANGUITA, D.; BONI, A. **Toward analog and digital hardware for support vector machines**. Proceedings of the International Joint Conference on Neural Networks, Vol. 4, pp. 2422-2426, Julho, 2001.

ANGUITA, D.; BONI, A. **Improved neural network for SVM learning**. IEEE Transactions on Neural Networks, Vol. 13, No. 5, pp. 1243-1244, Setembro, 2002.

ANGUITA, D.; BONI, A.; RIDELLA, S. **Digital kernel perceptron**. Electronics Letters, Vol. 38, No. 10, Maio, 2002.

ANGUITA, D.; BONI, A.; RIDELLA, S.; A Digital Architecture for Support Vector Machines: Theory, Algorithm and FPGA implementation, **IEEE Trans. on Neural Networks**, Vol. 14, No.5, pp. 993-1009, 2003.

ANGUITA, D.; BOZZA, G., **The effect of quantization on support vector machines with Gaussian kernel**, Neural Networks, 2005. IJCNN ’05. Proceedings. 2005 IEEE International Joint Conference on, vol.2, no., pp. 681-684 vol. 2, 31 Agosto, 2005.

ANGUITA, D. et al Feed-Forward Support Vector Machine Without Multipliers. **IEEE Transactions On Neural Networks**, Vol. 17, No. 5, 2006.

BECKER, J. et al. **Dynamic partial FPGA explotation.** In: Proceedings of the IEEE, v. n. 2, p. 438-452, 2007.

BIASI, I; BONI, A; ZORAT, A **reconfigurable parallel architecture for SVM classification** in IEEE Proc. IJCNN, 2005, pp. 2867–2872.

BOBDA, C. **Introduction to Reconfigurable Computing.** Dordrecht: Springer, 2007. 359 p.

BONI, A.; ZORAT, A. **FPGA Implementation of Support Vector Machine with Pseudo-Logarithmic Number Representation.** International Joint Conference on Neural Networks, 2006.

BRUNELLI, L. **Abordagem Para Redução De Complexidade De RNA Usando Reconfiguração Dinâmica.** 169 p. Tese (Doutorado) - Universidade Federal de Campina Grande, Campina Grande, 2005.

BURGES, C.J.C.; **A Tutorial on Support Vector Machines for Pattern Recognition.** Data Mining and Knowledge Discovery 2:121 - 167, 1998.

CORTES, C.; VAPNIK, V.; **Support-Vector Networks** Machine Learning, vol. 20, pp. 273–297, 1995.

DITTMANN, F; GÖTZ, M. **Reconfiguration Time Aware Processing on FPGAs**. In: Dagstuhl Seminar N. 06141 on Dynamically Reconfigurable Architectures, 2006.

ESKINAZI, R. **Uma Metodologia Para Escalonamento De Tarefas De Tempo Real Em Arquiteturas Dinamicamente Reconfiguráveis**, Tese (Doutorado) - Universidade Federal de Pernambuco, Recife, 2005.

ESQUIAGOLA, J. et al. **A dynamically reconfigurable Bluetooth Base Band Unit**. 2005. International Conference on Field Programmable Logic and Applications, 2005. pp. 148- 152.

ESQUIAGOLA, J. **Reconfiguração dinâmica de um controlador Bluetooth banda base**, Dissertação (Mestrado) - Escola politécnica, Universidade de São Paulo. São Paulo, 2006.

GENOV, R.; CAUWENBERGHS, G. **Kerneltron: Support Vector Machines in Silicon.** IEEE Transactions on Neural Networks, vol. 14 pp. 1426-1134, 2003.

GONZALEZ, J.A.Q. **Uma Metodologia de Projetos para Circuitos com Reconfiguração Dinâmica de Hardware Aplicada a Support Vector Machine**. Tese (Doutorado) - Escola politécnica, Universidade de São Paulo, 2006.

HAUCK, S. **The Roles of FPGAs in Reprogrammable Systems**. In: Proceedings of the IEEE, Vol. 86, No. 4, pp. 615-639, April, 1998.

HENKEL, J.; PARAMESWARAN, S. (Ed.). **Basics of Reconfigurable Computing.** Dordrecht: Springer, 2007. 525 p.

HORTA, E. L. **Comutador ATM reconfigurável dinamicamente através de hardware.** 2002. 94p. + 2 apêndices. Tese (Doutorado) – Escola politécnica, Universidade de São Paulo, São Paulo, 2002.

JOACHIMS, T. **Making large-scale SVM learning practical**. In Schöllkopf, B., et al. editor, Advances in Kernel Methods – Support Vector Learning, MIT Press, 1999

KEERTHI, S.S. et al **Improvements to Platt's Algorithm for SVM Classifier Design.** Neural Computation, 13:637–649, 2001.

KHAN, F. M.; ARNOLD, M. G.; POTTENGER, W. M. **Finite precision analysis of support vector machine classification in logarithmic number systems.** Proceedings of the Euromicro Symposium on Digital System Design, pp. 254-261, Sept, 2004.

KHAN, F. M.; ARNOLD, M. G.; POTTENGER, W. M.; **Hardware-based support vector machine classification in logarithmic number systems.** Proceedings of the International Symposium on Circuits and Systems, Vol. 5, pp. 5154-5157, May, 2005.

KOJIMA, L **Metodologia de Projeto de Sistemas Dinamicamente Reconfiguráveis.** Dissertação (Mestrado) - Escola politécnica, Universidade de São Paulo. São Paulo, 2007.

LERNER, B.; LAWRENCE, N. D. **A comparison of state-of-the-art classification techniques with application to cytogenetics**. Neural Computing & Applications, vol. 10(1), pp. 39-47, 2001.

LIU, H.; WONG, D. F. **Circuit partitioning for dynamically reconfigurable FPGAs**. Proceeding of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays, Monterey, CA, pp. 187-194, February, 1999.

LU, S. L.; YIANNACOURAS, P.; SUH, T.; KASSA, R.; KONOW, M. A Desktop Computer with a Reconfigurable Pentium® **ACM Transactions on Reconfigurable Technology and Systems (TRETS)**, vol. 1, No 1, Março, 2008.

LYSAGHT, P.; DUNLOP, J. **Dynamic reconfiguration of field programmable gate arrays.** In: The 1993 International Workshop on Field-Programmable Logic and Applications, 1993, pp 82-94.

LYSAGHT, P.; STOCKWOOD, J. A Simulation Tool for Dynamically Reconfigurable Field Programmable Gate Arrays. In: **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, v. 4, n. 3, p. 381-390, 1996.

MAJER, M. et al. **The Erlagen Slot Machine: A Dynamically Reconfigurable FPGA-based computer.** Journal of VLSI Signal Processing. 47, 2007.

MAK, T. S. T.; LAM, K. P. **Embedded computation of maximum-likelihood phylogeny inference using platform FPGA.** Proceedings of the 2004 IEEE Computational Systems Bioinformatics Conference, IEEE Computer Society, pp. 512-514, 2004.

NIKHIL, R., **Bluespec System Verilog: efficient, correct RTL from high level specifications** Formal Methods and Models for Co-Design, 2004.

OSUNA, E.; FREUND, R.; GIROSI, F. **Improved Training Algorithm for Support Vector Machines** Proc. IEEE NNSP ’97, (1997).

PEDERSEN, R.; SCHOEBERL, M. **An Embedded Support Vector Machine**. IEEE Conference Proceedings. Intelligent Solutions in Embedded Systems, 2006 International Workshop on, 2006.

PEDREGAL, P. **Introduction to Optimization**. Texts in Applied Mathematics, 46. Springer-Verlag New York Inc., 2004.

PLATT, J. C. **Fast Training Of Support Vector Machines Using Sequential Minimal Optimization**. In SCHOLKOPF, B; BURGES, C. J. C.; SMOLA, A. J., eds., “Advances in Kernel Methods - Support Vector Learning”, MIT Press, Cambridge, MA, pp. 185-208, 1999.

SOARES, R. I. **Infra-estrutura e Implementação de Controle de Configurações em Software para Hardware Confiurável**, dissertação (Mestrado) - Pontifícia Universidade Católica do Rio Grande do Sul, 2005.

VAPNIK, V. **Estimation of Dependences Based on Empirical Data** Springer-Verlag, (1982).

VAPNIK V. **Statistical Learning Theory**. New York: Wiley, 1998. 768 p.

XILINX. **Two Flows for Partial Reconfiguration***:* Module Based or Difference Based (v1.2). setembro, 2004. XAPP290.

XILINX. **Virtex-II Platform FPGAs***: Complete Data Sheet.* s.l. : DS03,1 v3.5, Novembro, 2007.

XILINX. **Early Access Partial Reconfiguration User Guide***, UG208 (v1.2)* Setembro, 2008.

XILINX, I. **Virtex-4 Configuration User Guide. .** UG071, v. 1.11, 9 de Junho 2009.

ZHANG, X.; NG, K. W. **A review of high-level synthesis for dynamically reconfigurable FPGAs.** In: Microprocessors and Microsystems, 24. 2000, pp 199-211.

APÊNDICE A - Algoritmo SMO

O pseudo-código a seguir descreve todo o algoritmo SMO. A matrizes α, y e F são variáveis globais e podem ser acessadas em qualquer função do algoritmo.

Algoritmo Principal

Iteração=1;

Para (i=1; i<=m ; i++) //Inicialização dos valores.

$\alpha_i = 0$;

$F_i = y_i$;

Fim Para

Enquanto $\left(b_{up} - b_{low} > \varepsilon\right)$

$b_{low} = 10$; //Valor arbitrário (contanto que $b_{low} > b_{up}$).

$b_{up} = -10$; //Valor arbitrário (contanto que $b_{low} > b_{up}$).

Seleção (C, m);

Otimização $(\alpha_p, \alpha_q, C, b_{low}, b_{up}, y_p, y_q)$;

Atualização $\left(\alpha_p^{old}, \alpha_p^{new}, \alpha_q^{old}, \alpha_q^{new}, y_p, y_q, x_p, x_q, m\right)$;

Iteração++;

Fim Enquanto

Seleção $(\boldsymbol{C}, \boldsymbol{m})$ // m exemplos

Para (t=1; t<=m ; t++)

Se $\left(\left(\alpha_t < C \;\&\; y_t = +1 \;\&\; F_t > b_{up}\right) \,||\, \left(\alpha_t > 0 \;\&\; y_t = -1 \;\&\; F_t > b_{up}\right)\right)$

$b_{up} = F_t$;

$p = t$;

Do contrário Se $\left(\left(\alpha_t < C \;\&\; y_t = -1 \;\&\; F_t < b_{low}\right) \,||\, \left(\alpha_t > 0 \;\&\; y_t = +1 \;\&\; F_t < b_{low}\right)\right)$

$b_{low} = F_t$;

$q = t$;

Fim se

Fim para

Atualização $(\boldsymbol{\alpha_p^{old}}, \boldsymbol{\alpha_p^{new}}, \boldsymbol{\alpha_q^{old}}, \boldsymbol{\alpha_q^{new}}, \boldsymbol{y_p}, \boldsymbol{y_q}, \boldsymbol{x_p}, \boldsymbol{x_q}, \boldsymbol{m})$;

Para (i=1; i<=m ; i++)

$F_i^{new} = F_i^{old} + y_p(\alpha_p^{old} - \alpha_p^{new})K(\vec{x}_p, \vec{x}_i) + y_q(\alpha_q^{old} - \alpha_q^{new})K(\vec{x}_q, \vec{x}_i)$;

$F_i^{old} = F_i^{new}$;

Fim Para

$\alpha_p^{old} = \alpha_p^{new}$;

$\alpha_q^{old} = \alpha_q^{new}$;

Otimização $(\boldsymbol{\alpha_p}, \boldsymbol{\alpha_q}, \boldsymbol{C}, \boldsymbol{b_{low}}, \boldsymbol{b_{up}}, \boldsymbol{y_p}, \boldsymbol{y_q})$

Se $(y_p \neq y_q)$

$Lo = \max(0, \alpha_q - \alpha_p)$;

$Hi = \min(C, C - \alpha_p + \alpha_q)$;

Do contrário

$Lo = \max(0, \alpha_q - C + \alpha_p)$;

$Hi = \min(C, \alpha_q + \alpha_p)$;

Fim Se

$\eta = K(\vec{x}_p, \vec{x}_p) + K(\vec{x}_q, \vec{x}_q) - 2K(\vec{x}_q, \vec{x}_p)$;

$\alpha_q^{new} = \alpha_q^{old} + y_q(b_{low} - b_{up})/\eta$;

Se $(\alpha_q^{new} > Hi)$

$\alpha_q^{new} = Hi$;

Do contrário Se $(\alpha_q^{new} < Lo)$

$\alpha_q^{new} = Lo$;

Fim Se

Se $(\alpha_q^{new} = \alpha_q^{old} - \alpha_p^{old})$ & $(y_p \neq y_q)$

$\alpha_p^{new} = 0$;

Do contrário Se $(\alpha_q^{new} = \alpha_q^{old} + \alpha_p^{old})$ & $(y_p = y_q)$

$\alpha_p^{new} = 0;$

Do contrário Se $(\alpha_q^{new} = C - \alpha_p^{old} + \alpha_q^{old})$ & $(y_p \neq y_q)$

$\alpha_p^{new} = C;$

Do contrário Se $(\alpha_q^{new} = \alpha_q^{old} - C + \alpha_p^{old})$ & $(y_p = y_q)$

$\alpha_p^{new} = C;$

Do contrário

$\alpha_p^{new} = \alpha_p^{old} + y_p y_q (\alpha_q^{old} - \alpha_q^{new});$

Fim Se

Kernel

Para (s=1; s<=d; s++) //Norma 1 // numero de elementos d

nm1 = $|x_{is} - x_{js}|$;

E1=E1+nm1;

Fim Para

E1=-E1* γ; //$\gamma = 2^p,\ p \in \mathbb{Z}$

F=fix(E1);

E1=E1-F;

B1=$\Delta\alpha$; //$\Delta\alpha = \alpha_x^{old} - \alpha_x^{new}$

Se (E1 – $\log_2(1 - 2^{-1})) > 0$

E2 = E1;

B2 = B1;

Do Contrário

E2 = E1 – $\log_2(1 - 2^{-1})$;

B2 = $(1 - 2^{-1}) *$ B1;

Fim Se

Se (E2 – $\log_2(1 - 2^{-2})) > 0$

E3 = E2;

B3 = B2;

Do Contrário

E3 = E2 – $\log_2(1-2^{-2})$;

B3 = $(1-2^{-2}) *$ B2;

Fim Se

Se (E3 – $\log_2(1-2^{-3})) > 0$

E4 = E3;

B4 = B3;

Do Contrário

E4 = E3 – $\log_2(1-2^{-3})$;

B4 = $(1-2^{-3}) *$ B3;

Fim Se

⋮

Se (En – $\log_2(1-2^{-n})) > 0$ //n define o número de ciclos do algoritmo CORDIC.

En = E(n-1);

Bn = B(n-1);

Do Contrário

En = E(n-1) – $\log_2(1-2^{-n})$;

Bn = $(1-2^{-n}) *$ B(n-1);

Fim Se

$k =$ Bn $* 2^F$;

Retorna(k);