



Universidade Federal de Juiz de Fora
Faculdade de Engenharia
Programa de Pós-Graduação em Engenharia Elétrica

Melissa Santos Aguiar

**Implementação Embarcada em FPGA de Métodos Visando a Reconstrução
Online de Energia no Calorímetro Hadrônico do Experimento ATLAS**

Juiz de Fora

2023

Melissa Santos Aguiar

**Implementação Embarcada em FPGA de Métodos Visando a Reconstrução
Online de Energia no Calorímetro Hadrônico do Experimento ATLAS**

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Juiz de Fora, na área de concentração em Sistemas Eletrônicos, como requisito parcial para a obtenção do título de Mestre em Engenharia Elétrica.

Orientador: Luciano Manhães de Andrade Filho

Juiz de Fora

2023

Ficha catalográfica elaborada através do Modelo Latex do CDC da UFJF
com os dados fornecidos pelo(a) autor(a)

Aguiar, Melissa.

Implementação Embarcada em FPGA de Métodos Visando a Reconstrução Online de Energia no Calorímetro Hadrônico do Experimento ATLAS /
Melissa Santos Aguiar. – 2023.

99 f. : il.

Orientador: Luciano Manhães de Andrade Filho

Dissertação de Mestrado – Universidade Federal de Juiz de Fora, Faculdade de Engenharia. Programa de Pós-Graduação em Engenharia Elétrica, 2023.

1. Processamento Embarcado. 2. FPGA. 3. Calorimetria. I. Andrade Filho, Luciano, orient. II. Título.

Melissa Santos Aguiar

Implementação Embarcada em FPGA de Métodos Visando a Reconstrução Online de Energia no Calorímetro Hadrônico do Experimento ATLAS

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Juiz de Fora como requisito parcial à obtenção do título de Mestre em Engenharia Elétrica. Área de concentração: Sistemas Eletrônicos

Aprovada em 29 de setembro de 2023.

BANCA EXAMINADORA

Prof. Dr. Luciano Manhães de Andrade Filho - Orientador

Universidade Federal de Juiz de Fora

Prof. Dr. Augusto Santiago Cerqueira

Universidade Federal de Juiz de Fora

Prof. Dr. Victor Araujo Ferraz

Universidade Federal do Rio Grande do Norte

Prof. Dr. Rafael Antunes Nóbrega

Universidade Federal de Juiz de Fora

Juiz de Fora, 13/09/2023.



Documento assinado eletronicamente por **Luciano Manhaes de Andrade Filho, Professor(a)**, em 02/10/2023, às 15:23, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do [Decreto nº 10.543, de 13 de novembro de 2020](#).



Documento assinado eletronicamente por **Augusto Santiago Cerqueira, Professor(a)**, em 02/10/2023, às 15:26, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do [Decreto nº 10.543, de 13 de novembro de 2020](#).



Documento assinado eletronicamente por **Victor Araujo Ferraz, Usuário Externo**, em 02/10/2023, às 16:07, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do [Decreto nº 10.543, de 13 de novembro de 2020](#).



Documento assinado eletronicamente por **Rafael Antunes Nobrega, Professor(a)**, em 04/10/2023, às 13:45, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do [Decreto nº 10.543, de 13 de novembro de 2020](#).



A autenticidade deste documento pode ser conferida no Portal do SEI-Ufjf (www2.ufjf.br/SEI) através do ícone Conferência de Documentos, informando o código verificador **1466846** e o código CRC **84F58F5E**.

Dedico este trabalho ao Jake.

AGRADECIMENTOS

Eu gostaria de agradecer a todos que colaboraram de forma significativa para a concretização deste trabalho. Em especial ao Luciano, que me inspira por além de contribuir com a educação, sendo um pesquisador e professor muito presente, acessível e com alto padrão de ensino, ainda consegue se dedicar com muito talento à música. Sou muito grata e tenho sorte por você ter me orientado nesta jornada.

Agradeço aos meus amigos Camila, Dabson e Jhei, que se tornaram a minha segunda família e me proporcionaram os melhores e mais divertidos momentos desde que me mudei para Juiz de Fora. A minha vida não seria a mesma sem vocês nela.

Aos amigos que fiz no decorrer da vida acadêmica, em especial ao João, Letícia, Elidiane, Elisa e Nathânia, que foram ótimas companhias tanto nas aulas e trabalhos quanto nos momentos de lazer. Aos amigos e colegas de pesquisa do grupo NIPS/UFJF e do CERN Cluster Brazil, em especial à Dayane, Victor e Lucca.

Agradeço também à minha família, principalmente os meus pais Simone e Rogério, que batalharam muito para me proporcionarem a oportunidade de me dedicar exclusivamente aos estudos. Eu sei que não foi nada fácil e que muitas vezes vocês precisaram abrir mão de diversas coisas para colocarem os filhos em primeiro lugar, eu espero um dia poder retribuir tudo isso. Se não fosse pelo apoio de vocês, eu não chegaria aqui.

À colaboração ATLAS no CERN, em especial ao TileCal pela oportunidade de conhecer e colaborar neste complexo sistema pessoalmente, onde fui bem recebida e pude me especializar ainda mais nesta área que eu tanto me identifico.

Também gostaria de agradecer ao Laboratório Nacional de Luz Síncrotron, em especial ao Daniel, Augusto e Lucas, que são profissionais de excelência com quem eu tive o prazer de trabalhar junto no Grupo de Controles Avançados do Sirius, onde pude aprender muito com cada um, me tornando uma profissional melhor.

Por fim, agradeço a cada professor e a cada professora que eu tive a honra de ser aluna nesta longa jornada, e também a Universidade Federal de Juiz de Fora, pelas diversas oportunidades de ensino e inclusão, tais como o sistema de apoio estudantil e pelas bolsas de monitoria, iniciação científica e treinamento profissional.

Este trabalho recebeu suporte, em parte, do CNPq, CAPES e FAPEMIG (Agências brasileiras de fomento à pesquisa científica).

RESUMO

Os laboratórios de pesquisa em Física Experimental de Altas Energias vêm colaborando com avanços significativos na ciência e tecnologia. Neles, são construídos modernos aparatos para a detecção e estudo de partículas, onde ocorrem interações entre partículas subatômicas com altas energias, tendo como principal objetivo o estudo de diversas características das partículas elementares. O LHC, que é atualmente o maior acelerador de partículas do mundo, está passando por um processo gradual de atualização, em que a luminosidade das colisões está sendo aumentada, objetivando o aumento na probabilidade de ocorrerem eventos cada vez mais raros. Isto está impactando diretamente nos sistemas de instrumentação dos detectores, em especial no Calorímetro Hadrônico do Experimento ATLAS, onde o aumento na ocorrência de colisões adjacentes ocasiona o efeito de empilhamento nos sinais (*pile-up*). Devido ao fato de o algoritmo atualmente em uso na reconstrução dos sinais neste calorímetro advir de um método que não considera o empilhamento de sinais em sua formulação, diversas alternativas para realizar a estimação de energia estão sendo propostas, como os métodos iterativos de deconvolução de sinais baseados em Representação Esparsa de Dados e também métodos não lineares baseados em Redes Neurais Artificiais. Tais metodologias apresentam custo computacional alto, de modo que o principal desafio atualmente é o desenvolvimento de algoritmos capazes de operar de forma *online*. Neste contexto, o presente trabalho descreve a implementação de algoritmos em processadores dedicados, em FPGA, com uma arquitetura *multicore*, visando realizar a reconstrução *online* de energia em Calorímetros de Altas Energias, tendo como foco o Calorímetro Hadrônico do Experimento ATLAS, em cenários de empilhamento de sinais. Os resultados mostram a possibilidade da operação de tais processadores no primeiro nível de *trigger* do ATLAS, respeitando a taxa de colisões do LHC de 40 MHz, e com consumo de recursos lógicos dentro dos limites para implementação em FPGAs modernas.

Palavras-chave: Processamento Embarcado. Deconvolução de Sinais. Redes Neurais.

ABSTRACT

Research laboratories in Experimental High Energy Physics have been collaborating on significant advances in science and technology. Within these laboratories, modern apparatuses are constructed for the detection and study of particles, where interactions occur between subatomic particles with high energies. The primary objective is to investigate various characteristics of elementary particles. The Large Hadron Collider (LHC), currently the world's largest particle accelerator, is undergoing a gradual upgrade to increase the luminosity of the collisions, aiming to increase the probability of rarer events occurring. This directly impacts the instrumentation systems of detectors, particularly the Hadronic Calorimeter of the ATLAS Experiment, where the increased occurrence of adjacent collisions leads to the phenomenon of pile-up. Due to the current algorithm used in signal reconstruction in this calorimeter not considering the pile-up in its formulation, various alternatives for energy estimation are being proposed. These include iterative methods of signal deconvolution based on Sparse Data Representation and non-linear methods based on Artificial Neural Networks. These methodologies have a high computational cost, and the primary challenge is currently the development of algorithms capable of operating online. In this context, this work describes the implementation of algorithms on dedicated processors in FPGA with a multicore architecture, aiming to perform online energy reconstruction in High Energy Calorimeters, focusing on the Hadronic Calorimeter of the ATLAS Experiment in pile-up scenarios. The results demonstrate the possibility of operating such processors at the ATLAS first-level trigger, respecting the LHC collision rate of 40 MHz, and with logical resource consumption within the limits for implementation in modern FPGAs.

Key-words: Embedded Processing. Signal Deconvolution. Neural Networks.

LISTA DE ILUSTRAÇÕES

Figura 1 – Partículas elementares que compõem o Modelo Padrão [5].	14
Figura 2 – Visão geral do LHC [28].	19
Figura 3 – Complexo do LHC com todos detectores e sub-detectores [29].	20
Figura 4 – O detector ATLAS e seus sub-sistemas [38].	21
Figura 5 – Sistema de calorimetria do ATLAS [43].	22
Figura 6 – Partículas interagindo com os sub-detectores do ATLAS [45].	23
Figura 7 – Esquema de um módulo do TileCal [34].	24
Figura 8 – Formato de um pulso característico do TileCal [47].	24
Figura 9 – Efeito <i>pile-up</i> no TileCal [10].	26
Figura 10 – Sinais do TileCal simulados usando <i>Toy Monte Carlo</i>	27
Figura 11 – Função proposta para reconstrução de energia [63].	31
Figura 12 – Modelo de um neurônio [73].	33
Figura 13 – Filtro FIR em aplicação de fluxo contínuo [16].	34
Figura 14 – Rede <i>feedforward</i> em aplicação de fluxo contínuo [16].	35
Figura 15 – Processador SAPHO e seus blocos principais.	37
Figura 16 – Código em C_+^+ na IDE do SAPHO.	40
Figura 17 – Código em <i>Assembly</i> na IDE do SAPHO.	41
Figura 18 – Tela de configuração do processador na IDE do SAPHO.	41
Figura 19 – Código em C_+^+ e respectivo <i>Assembly</i> antes de implementar o PSET.	45
Figura 20 – Código em C_+^+ e respectivo <i>Assembly</i> com o PSET.	45
Figura 21 – Diagrama simplificado do SAPHO com o PSET.	46
Figura 22 – Código <i>Verilog</i> do PSET implementado em ponto-fixado.	46
Figura 23 – Código <i>Verilog</i> do PSET implementado em ponto-flutuante.	46
Figura 24 – Código em C_+^+ e respectivo <i>Assembly</i> antes de implementar o NORM.	47
Figura 25 – Código em C_+^+ e respectivo <i>Assembly</i> com o NORM.	47
Figura 26 – Código <i>Verilog</i> do NORM implementado na ULA.	47
Figura 27 – Código em C_+^+ e respectivo <i>Assembly</i> antes de implementar o ABS.	48
Figura 28 – Código em C_+^+ e respectivo <i>Assembly</i> com o ABS.	48
Figura 29 – Código <i>Verilog</i> do ABS implementado em ponto-fixado.	48
Figura 30 – Código em C_+^+ e respectivo <i>Assembly</i> antes de implementar o SIGN.	49
Figura 31 – Código em C_+^+ e respectivo <i>Assembly</i> com o SIGN.	49
Figura 32 – Código <i>Verilog</i> do ABS e SIGN implementados em ponto-flutuante.	50
Figura 33 – Inicialização de <i>array</i> no SAPHO antes da otimização.	50
Figura 34 – Inicialização de <i>array</i> no SAPHO com a otimização.	50
Figura 35 – Coeficientes da matriz de convolução.	51
Figura 36 – Coeficientes da matriz de autocorrelação.	53
Figura 37 – Coeficientes da matriz pseudo-inversa.	54
Figura 38 – Erro RMS da estimação de energia pelo método SSF [69].	55

Figura 39 – Relação entre o valor RMS do erro e o ganho.	56
Figura 40 – Diagrama da estrutura <i>multicore</i> implementada.	59
Figura 41 – Código para o preenchimento da janela de entrada de dados.	60
Figura 42 – Diagrama em ponto-fixa da implementação da RNA [65].	62
Figura 43 – Análises de quantização [65].	62
Figura 44 – Fluxograma da RNA em ponto-fixa.	63
Figura 45 – Erro obtido variando a ordem e o centro da Série de Taylor.	64
Figura 46 – Erro obtido variando número de bits do expoente e da mantissa.	65
Figura 47 – Fluxograma da RNA em ponto-flutuante.	65
Figura 48 – Quantidade de processadores variando com a frequência.	68
Figura 49 – Tempo de atraso variando com a frequência.	68
Figura 50 – Elementos lógicos variando com a frequência.	69
Figura 51 – Bits de memória variando com a frequência.	69
Figura 52 – Janela de entrada dos dados do método SSF no Modelsim.	70
Figura 53 – Comparação entre o sinal reconstruído, o alvo e o sinal ruidoso.	71
Figura 54 – IDE do SAPHO com a parte principal do código da FFT.	73
Figura 55 – Operações do Algoritmo 1.	88
Figura 56 – Operações do Algoritmo 2.	88
Figura 57 – Operações do Algoritmo 3.	88
Figura 58 – Operações do Algoritmo 4.	88
Figura 59 – Operações do Algoritmo 5.	88
Figura 60 – Tutorial SAPHO: criando um novo projeto.	94
Figura 61 – Tutorial SAPHO: configurando o diretório do projeto.	94
Figura 62 – Tutorial SAPHO: criando um processador.	94
Figura 63 – Tutorial SAPHO: configurando os parâmetros do processador.	95
Figura 64 – Tutorial SAPHO: interface de desenvolvimento em C^+	95
Figura 65 – Tutorial SAPHO: compilando o código.	96
Figura 66 – Tutorial SAPHO: código em <i>Assembly</i> gerado.	96
Figura 67 – Tutorial SAPHO: abrindo o projeto no Quartus.	97
Figura 68 – Tutorial SAPHO: selecionando os arquivos de parametrização.	97
Figura 69 – Tutorial SAPHO: resumo do projeto criado no Quartus.	98
Figura 70 – Tutorial SAPHO: selecionando o arquivo principal.	98
Figura 71 – Tutorial SAPHO: compilando o projeto.	99
Figura 72 – Tutorial SAPHO: visualizando o processador criado.	99
Figura 73 – Tutorial SAPHO: diagrama do <i>top level</i> do processador desenvolvido.	99

LISTA DE TABELAS

Tabela 1 – Ferramentas e energia mínima necessárias para explorar distâncias [26].	18
Tabela 2 – Energia e luminosidade de alguns aceleradores de partículas [26].	18
Tabela 3 – Instruções e respectivos circuitos criados automaticamente.	39
Tabela 4 – Palavras-chave e Operadores Lógico-Aritméticos da Linguagem C permitidos.	42
Tabela 5 – Comparação entre as implementações do método SSF.	67
Tabela 6 – Custo lógico e frequência (<i>hardware</i> para ponto-flutuante indicado por *)	73
Tabela 7 – Quantidade de <i>clocks</i> (<i>hardware</i> para ponto-flutuante indicado por *)	74
Tabela 8 – Tempo de execução (<i>hardware</i> para ponto-flutuante indicado por *)	74

LISTA DE ABREVIATURAS E SIGLAS

ALICE	<i>A Large Ion Collider Experiment</i>
ATLAS	<i>A Toroidal LHC ApparatuS</i>
BC	<i>Bunch Crossing</i>
CERN	<i>Conseil Européen pour la Recherche Nucléaire</i>
CMS	<i>Compact Muon Solenoid</i>
FIR	<i>Finite Impulse Response</i>
FPGA	<i>Field Programmable Gate Array</i>
GD	<i>Gradient Descent</i>
IDE	<i>Integrated Development Environment</i>
LAr	<i>Liquid Argon</i>
LHC	<i>Large Hadron Collider</i>
LHCb	<i>A Large Ion Collider beauty</i>
LHCf	<i>The Large Hadron Collider forward</i>
LP	<i>Linear Programming</i>
LUT	<i>Look-Up Table</i>
PMT	<i>PhotoMultiplier Tube</i>
PSC	<i>Processador Soft-Core</i>
RAM	<i>Random Access Memory</i>
RMS	<i>Root Mean Square</i>
RNA	<i>Redes Neurais Artificiais</i>
SAPHO	<i>Scalable-Architecture Processor for Hardware Optimization</i>
SR	<i>Sparse Representation</i>
SSF	<i>Separable Surrogate Functionals</i>
TileCal	<i>Tile Calorimeter</i>
TOTEM	<i>TOTAL Elastic and diffractive cross section Measurement</i>
ULA	<i>Unidade Lógico-Aritmética</i>

LISTA DE SÍMBOLOS

c	Velocidade da luz no vácuo
Δx	Distância
\mathcal{L}_u	Luminosidade
\leq	Menor ou igual que
P_ℓ	Problema- ℓ
$ \mathbf{x} $	Módulo de \mathbf{x}
$\ \mathbf{x}\ _\ell^\ell$	Norma- ℓ de \mathbf{x}
$\sum_i \mathbf{x}_i$	Somatório em i de \mathbf{x}_i
ϵ_0	Erro quadrático
λ	Multiplicador de Lagrange
\mathbf{x}_{opt}	Valor ótimo de \mathbf{x}
μ	Tamanho do passo em direção ao mínimo
$S_\lambda(\theta)$	Função de Shrinkage
$\ \mathbf{x}\ _2^2$	Norma quadrática euclidiana de \mathbf{x}
$\mathcal{L}(\mathbf{x})$	Lagrangiano de \mathbf{x}
∂	Derivada parcial
M^T	Transposta da matriz \mathbf{M}
M^{-1}	Inversa da matriz \mathbf{M}
M^+	Pseudo-inversa da matriz \mathbf{M}
@	Função que implementa a instrução PSET
/ $>$	Função que implementa a instrução NORM
Δt	Tempo de atraso

SUMÁRIO

1	INTRODUÇÃO	14
1.1	MOTIVAÇÃO E OBJETIVOS	16
1.2	ESTRUTURA DO TRABALHO	16
2	LHC - O GRANDE COLISOR DE HÁDRONS	17
2.1	O EXPERIMENTO ATLAS	21
2.1.1	O Calorímetro Hadrônico e o Sistema de <i>Trigger</i>	23
2.2	DESAFIO: O EFEITO <i>PILE-UP</i>	26
3	REVISÃO BIBLIOGRÁFICA	28
3.1	ESTIMAÇÃO DE ENERGIA USANDO REPRESENTAÇÃO ESPARSA	29
3.1.1	Inicialização Aprimorada do Algoritmo por meio da Matriz Pseudo-Inversa	31
3.2	REDES NEURAIIS ARTIFICIAIS	32
3.2.1	Estimação Online e Não-Linear de Energia usando Redes Neurais	34
4	SAPHO: UM PROCESSADOR AUTO-ESCALÁVEL	36
4.1	<i>HARDWARE</i>	37
4.2	<i>SOFTWARE</i>	39
4.2.1	Compilador C	42
4.2.2	Compilador Assembler	43
4.3	FORMATO DE PONTO-FLUTUANTE	43
5	IMPLEMENTAÇÃO	44
5.1	OTIMIZAÇÕES NA ESTRUTURA DO SAPHO	44
5.2	REPRESENTAÇÃO ESPARSA EM PONTO-FLUTUANTE	51
5.2.1	Inicialização Otimizada do Algoritmo	54
5.3	REPRESENTAÇÃO ESPARSA EM PONTO-FIXO	55
5.4	ESTRUTURA MULTICORE	59
5.5	REDES NEURAIIS ARTIFICIAIS USANDO <i>LOOK-UP TABLE</i>	61
5.5.1	Aproximação da Função Sigmoide usando Séries de Taylor	63
6	RESULTADOS	67
7	ANÁLISES DE DESEMPENHO DO SAPHO	72
8	CONCLUSÕES	76

REFERÊNCIAS	77
APÊNDICE A – Produção Bibliográfica	84
ANEXO A – Descrição dos Algoritmos do Método SSF	88
ANEXO B – Tutorial: Criando um Projeto com o SAPHO	94

1 INTRODUÇÃO

O processo da exploração de questões fundamentais sobre a composição do átomo e das partículas básicas que constituem a matéria necessita de estudos em Física de Altas Energias [1], por meio de experimentos envolvendo um refinado sistema de instrumentação: os colisionadores de partículas [2]. Tais experimentos consistem em acelerar e colidir feixes de partículas a velocidades próximas à da luz, gerando ambientes que se assemelham ao estado do universo nos instantes iniciais após o *Big Bang* [3]. O objetivo é que os feixes de partículas aceleradas colidam nas regiões de interesse onde estão presentes os detectores, de forma que é possível medir as propriedades das partículas subprodutos destas colisões.

As colisões entre partículas nos aceleradores geram uma grande quantidade de energia, permitindo a observação de elementos que antes eram desconhecidos para a ciência. Na Figura 1 é possível ver o diagrama das partículas que constituem o Modelo Padrão¹, destacando-se o Bóson de Higgs, que havia sido previsto teoricamente por esse modelo e finalmente teve a existência confirmada [4], sendo atualmente um dos maiores exemplos de descobertas feitas por meio de experimentos realizados com colisões de partículas.

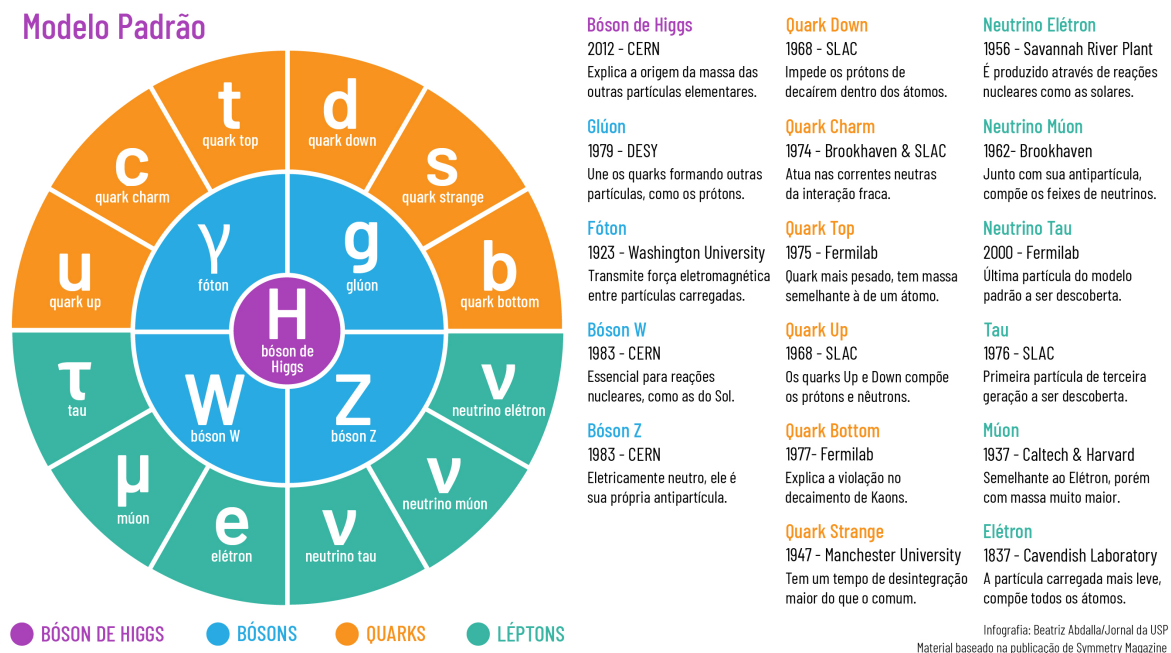


Figura 1 – Partículas elementares que compõem o Modelo Padrão [5].

Cerca de 5% da matéria conhecida do universo é explicada pelo Modelo Padrão, já os outros 95% são compostos pela matéria escura, que ainda não foi detectada diretamente pelos instrumentos e experimentos desenvolvidos até então [6].

¹ O Modelo Padrão é uma teoria da Física de Partículas que descreve as forças fundamentais fortes, fracas, eletromagnéticas e as partículas fundamentais que compõem a matéria [7].

Na prática, os aceleradores de partículas são equipamentos construídos para acelerar e aumentar a energia de feixes de partículas, através da geração de campos elétricos e campos magnéticos que são suficientemente fortes para orientar e focalizar os feixes [8].

Para garantir a confiabilidade dos dados medidos, é necessário um sofisticado sistema de processamento, de forma a permitir a correta interpretação de determinados eventos físicos. Tal sistema pode ser subdividido entre dois tipos de processamento: o *online* e o *offline* [9]. O processamento *online* é responsável por realizar a seleção das informações na medida em que as colisões ocorrem, respeitando a latência do sistema, enquanto o *offline* analisa os dados já armazenados após as colisões [10].

No contexto desta demanda por complexos sistemas de instrumentação, que contribuem em explorações como as componentes fundamentais da matéria e suas dinâmicas de interação, a Organização Europeia para a Pesquisa Nuclear (*Conseil Européen pour la Recherche Nucléaire*), também conhecida como CERN, fundada no ano de 1954 [11], em Genebra, Suíça, vem se destacando mundialmente nos experimentos envolvendo a Física de Altas Energias. Tais experimentos são extremamente importantes na compreensão da evolução do universo desde o *Big Bang* até os dias atuais.

O Experimento ATLAS é um dos responsáveis pela aquisição dos dados resultantes das colisões que ocorrem no LHC (*Large Hadron Collider*), que é o principal acelerador de partículas do CERN. O detector ATLAS é formado por sub-detectores dispostos em camadas, sendo cada uma destas responsável por medir propriedades específicas das partículas geradas pelas colisões.

Uma importante característica a ser mensurada é a energia, que é obtida através da estimação da amplitude dos sinais que são gerados na eletrônica de leitura dos calorímetros [12], quando as partículas interagem com seu material absorvedor.

No Calorímetro Hadrônico do ATLAS, devido a ocorrerem colisões no LHC a cada 25 ns e a largura do pulso ser de 150 ns, ocorre um efeito de empilhamento enquanto o sinal se desenvolve. Com o upgrade do LHC, a probabilidade de colisões consecutivas na mesma região do calorímetro irá aumentar, agravando este efeito de empilhamento.

Buscando resolver tal problema, foram propostos métodos que se baseiam na deconvolução destes sinais para recuperar a sua amplitude [10, 13]. Os algoritmos têm sido implementados com filtros digitais simples do tipo FIR (*Finite Impulse Response*) [14]. Ainda neste contexto, abordagens por meio de métodos baseados em teoria de Representação Esparsa de dados [15] e métodos baseados em Redes Neurais Artificiais [16] têm se destacado quanto à eficiência na reconstrução dos sinais e serão explorados no presente trabalho. Tais métodos vêm sendo estudados por pesquisadores do CERN Brazil Cluster e testados para implementação *offline*, devido ao alto custo computacional, mas poderiam ajudar também na seleção de eventos, caso fossem implementados *online*.

1.1 MOTIVAÇÃO E OBJETIVOS

Devido ao bom desempenho na reconstrução de energia apresentado pelo método iterativo de deconvolução baseado em Representação Esparsa de dados e pelo algoritmo baseado em Redes Neurais Artificiais, o objetivo do presente trabalho é buscar formas de implementar estes métodos em *hardware*, visando a reconstrução *online* de energia em Calorímetros de Altas Energias, com foco no Calorímetro Hadrônico do ATLAS.

Para a implementação de tais métodos em FPGA (*Field Programmable Gate Array*) [18], foi utilizado o SAPHO, um processador *Soft-Core* desenvolvido na Universidade Federal de Juiz de Fora, que é capaz de realizar as operações dos algoritmos propostos através de circuitos aritméticos em ponto-fixo e em ponto-flutuante.

No decorrer deste trabalho, o SAPHO foi customizado e adaptado, onde foram adicionadas novas funcionalidades e seu desempenho foi analisado. Além disso, arquiteturas *multicore* (vários núcleos) foram desenvolvidas para o processamento respeitar os requisitos temporais de operação, que são necessários devido à alta taxa de eventos no calorímetro.

1.2 ESTRUTURA DO TRABALHO

Esta dissertação está organizada da seguinte forma: no Capítulo 1 foram apresentados a contextualização, motivação e os objetivos do trabalho.

O ambiente deste trabalho é introduzido no Capítulo 2, com destaque para o Experimento ATLAS e seus sistemas de calorimetria e de filtragem *online*.

É apresentada, no Capítulo 3, uma introdução sobre o método atualmente em uso na reconstrução dos sinais, suas limitações para lidar com o problema de empilhamento e o embasamento matemático das metodologias a serem implementadas.

O SAPHO, que é a ferramenta utilizada para o desenvolvimento dos processadores embarcados, é detalhado no Capítulo 4. As otimizações na estrutura do SAPHO e as implementações são descritas no Capítulo 5, onde são propostas arquiteturas *multicore* para o processamento, visando estimar a energia respeitando a taxa de eventos no calorímetro.

O ambiente de simulação utilizado é introduzido no Capítulo 6, onde os resultados obtidos são discutidos e as comparações entre as implementações são realizadas.

As análises de desempenho do SAPHO são detalhadas no Capítulo 7.

No Capítulo 8, são apresentadas as conclusões gerais deste trabalho, as suas principais contribuições e propostas de trabalhos futuros para a continuidade da pesquisa.

2 LHC - O GRANDE COLISOR DE HÁDRONS

Para realizar estudos de objetos cada vez menores, os limites da tradicional Mecânica Newtoniana [19] acabam sendo ultrapassados e, a partir desse ponto, as leis da Mecânica Quântica [20] passam a descrever melhor o comportamento destes objetos. Assim, os aceleradores de partículas podem ser comparados com grandes microscópios e, a partícula, regida pelas leis da Mecânica Quântica, passa a ser vista como uma entidade dual. Tal dualidade é conhecida como partícula-onda [21], e muitos aspectos do comportamento desta entidade só podem ser descritos em termos de probabilidades, onde esta entidade dual é modelada como sendo inerentemente espalhada, não podendo mais ser descrita como um ponto com posição e velocidade determinadas, como seria na Mecânica Newtoniana.

Esta entidade dual pode ser visualizada como uma nuvem de probabilidades, tendo suas dimensões comparáveis ao comprimento de onda, de forma que a partícula fisicamente visível pode ser encontrada em algum ponto desta nuvem de probabilidades. Quando o comprimento de onda é reduzido, o volume da nuvem também é reduzido. Porém, para isso ocorrer, faz-se necessário aumentar o momento da partícula, ou seja, sua energia precisa ser aumentada. Portanto, ao aumentar a energia do experimento, é possível estudar objetos com dimensões cada vez menores. Nesse contexto, os aceleradores de partículas são ferramentas que permitem a observação de estruturas muito pequenas, através da produção de partículas com alto momento transversal e, conseqüentemente, comprimento de onda curto, uma vez que, segundo o princípio da incerteza de Heisenberg [22], o comprimento de onda associado é inversamente proporcional ao momento da partícula (p) [23].

Além disso, segundo os estudos da relatividade de Albert Einstein [24], com a famosa equação $E = mc^2$, descobre-se que a energia pode ser convertida em matéria e que a recíproca também é válida. Assim, os aceleradores podem criar partículas ao fazer com que dois feixes de partículas altamente energéticos se choquem, possibilitando então o estudo das partículas fundamentais [25]. Na prática, grandes detectores com diversas camadas são posicionados ao redor do ponto de colisão nos experimentos mais modernos, onde cada camada de detecção tem uma função específica na determinação da trajetória e na identificação de cada uma das muitas partículas que podem ser produzidas em uma única colisão. Assim, quanto maior a energia das partículas, ou seja, quanto maior a aceleração aplicada a estas, maior a eficiência na produção de partículas elementares.

Na Tabela 1 é possível observar algumas ordens de grandeza da energia mínima necessária para a exploração de distâncias Δx [26]. Como já era esperado, devido a teoria da dualidade entre partícula e onda, é possível notar que, para explorar as menores distâncias, são necessários valores de energia mais altos e, além disso, as ferramentas utilizadas para a obtenção das mais altas energias são os aceleradores de partículas.

Tabela 1 – Ferramentas e energia mínima necessárias para explorar distâncias [26].

Ferramentas Utilizadas	Distância (centímetro)	Energia (elétron-volt*)
Microscópios	10^{-5}	2 eV
Raios-X	10^{-8}	2 keV
Raios- γ	10^{-11}	2 MeV
Aceleradores de Partículas	10^{-14}	2 GeV
Aceleradores de Partículas	10^{-16}	200 GeV
Aceleradores de Partículas	10^{-17}	2 TeV

* elétron-volt é a unidade de energia definida como o trabalho realizado ao se mover um elétron através de uma diferença de potencial de 1 volt. 1 eV equivale a $1,6 \times 10^{-19}$ joules.

Um acelerador de partículas é basicamente composto por uma fonte de partículas carregadas, um campo acelerador e um campo que força a partícula a se mover em uma órbita bem definida. O acelerador de alvo fixo lança um conjunto de partículas contra um alvo fixo de forma a promover uma colisão entre ambos. O espalhamento das partículas resultantes desta colisão fornece informações sobre a estrutura do feixe e do alvo. Já nos colisores, os feixes são acelerados e direcionados para colidirem com igual momento e sinais opostos, utilizando melhor a energia envolvida na colisão. Estes são menos versáteis e produzem um número menor de interações por unidade de tempo quando comparados com os de alvo fixo, porém, eles têm a vantagem de obterem energias maiores [26].

Uma importante característica dos aceleradores de partículas é a luminosidade \mathcal{L}_u , que é definida como o número que, multiplicado pela seção transversal total de um dado processo σ , fornece o número total de colisões por unidade de tempo N , ou seja, $N = \mathcal{L}_u \sigma$.

Atingir uma alta luminosidade é um dos principais objetivos dos aceleradores de partículas, uma vez que a maior parte dos eventos produzidos durante as colisões não são de interesse para os estudos da física de altas energias.

Na Tabela 2 são indicadas características de energia e luminosidade de alguns dos principais aceleradores de partículas do CERN.

Tabela 2 – Energia e luminosidade de alguns aceleradores de partículas [26].

Acelerador	\mathcal{L}_u ($cm^{-2}s^{-1}$)	Energia (elétron-volt)
Super Proton Synchrotron (1981)	6×10^{30}	630-900 GeV
Large Electron-Positron (1989)	10^{31-32}	90-200 GeV
Large Hadron Collider (2010-2012)	$0,76 \times 10^{34}$	7-8 TeV
Large Hadron Collider (2015)	10^{34}	14 TeV

O Grande Colisor de Hádrons ou LHC, do inglês *Large Hadron Collider*, é atualmente o principal acelerador de partículas do CERN, sendo considerado o maior e mais energético acelerador de partículas já construído [27]. Ele é composto por um anel de ímãs supercondutores de 27 km de circunferência, com várias estruturas aceleradoras para aumentar a energia das partículas ao longo do caminho, situando-se a 100 metros de profundidade entre as fronteiras da França e da Suíça, como ilustrado na Figura 2.

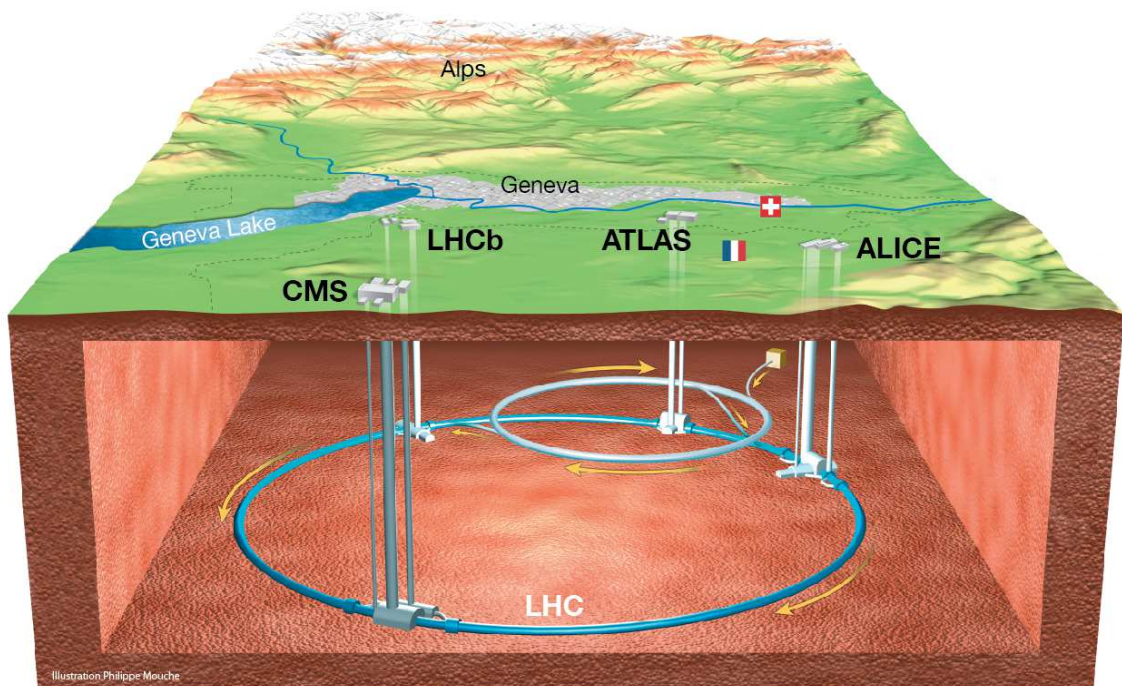


Figura 2 – Visão geral do LHC [28].

O complexo do LHC [29], que pode ser visto na Figura 3, possui detectores em pontos estratégicos (onde ocorrem as colisões) ao longo de sua circunferência, sendo estes, os instrumentos responsáveis pela aquisição dos dados resultantes dos eventos, ao possibilitarem a reconstrução das colisões ocorridas, de forma a permitir a detecção das partículas subprodutos destas colisões [30].

Estes experimentos são baseados em um modelo de camadas, onde cada uma delas é responsável por estimar as propriedades específicas dos diferentes tipos de partículas. Dentre elas, há as partículas que interagem de forma eletromagnética, que incluem os elétrons, fótons e pósitrons, e há também as partículas hadrônicas, dentre elas, os prótons e nêutrons, que são partículas que interagem mais evidentemente através de força forte. Com relação aos experimentos do LHC, as principais características destacadas são:

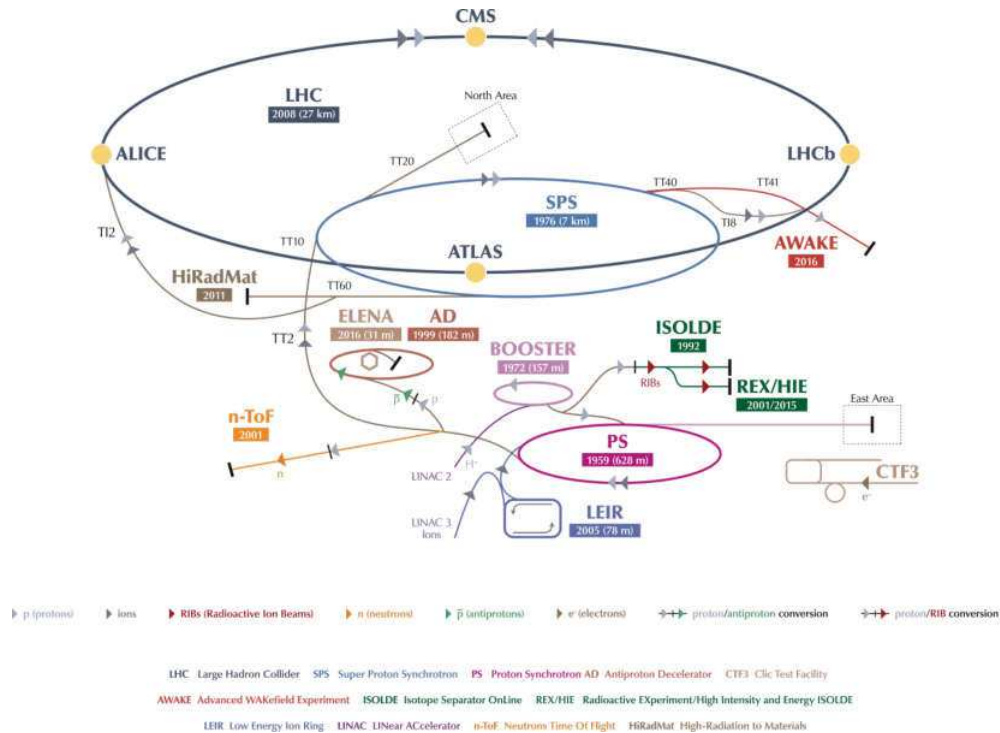


Figura 3 – Complexo do LHC com todos detectores e sub-detectores [29].

- **CMS** – *Compact Muon Solenoid* [31]
 É um detector de propósito geral, que foi projetado para estudar o Bóson de Higgs, partículas supersimétricas e a física de íons pesados.
- **LHCb** – *A Large Ion Collider beauty* [32]
 Detector dedicado a pesquisas sobre a simetria aparente entre a matéria e a anti-matéria presentes no universo.
- **ALICE** – *A Large Ion Collider Experiment* [33]
 É o único detector especializado em colisões de íons de chumbo cujo principal objetivo é o estudo da formação e propriedades do plasma de *quark-gluon*, uma matéria que se acredita apenas ter existido por poucos instantes após o *Big Bang*.
- **ATLAS** – *A Toroidal LHC Apparatus* [34]
 Experimento de propósito geral, que foi otimizado para detectar o maior número possível de eventos físicos que ocorrerão no LHC.
- **TOTEM** e **LHCf** – *TOTAL Elastic and diffractive cross section Measurement* [35] e *The Large Hadron Collider forward* [36]
 Experimentos de pequeno porte dedicados à física projetiva (*forward*) de prótons e íons pesados.

2.1 O EXPERIMENTO ATLAS

O ATLAS é um dos principais experimentos do LHC, possuindo propósito geral para a detecção das colisões do tipo próton-próton. Sua colaboração envolve cerca de 38 países e 174 institutos de pesquisa, contando com mais de 5.000 cientistas ao redor do mundo, sendo projetado com foco no estudo da maior quantidade possível de fenômenos físicos passíveis de serem gerados em colisões no LHC [37], desde a busca pelo Bóson de Higgs até dimensões extras e partículas que possam constituir a matéria escura.

O detector ATLAS, ilustrado na Figura 4, pesa cerca de 7.000 toneladas, e suas dimensões são de aproximadamente 25 metros de altura por 44 metros de largura. Ele tem formato cilíndrico e cobre um ângulo sólido próximo a 4π ao redor da região de colisão das partículas. Ainda nesta figura, é possível notar que, além dos magnetos responsáveis pela geração de intensos campos magnéticos que auxiliam na medida de momento das partículas carregadas, o ATLAS é composto também por três sub-detectores básicos: o detector de trajetórias, os calorímetros eletromagnético e hadrônico e, por fim, o detector de múons, respectivamente, ordenados do mais interno para o mais externo.

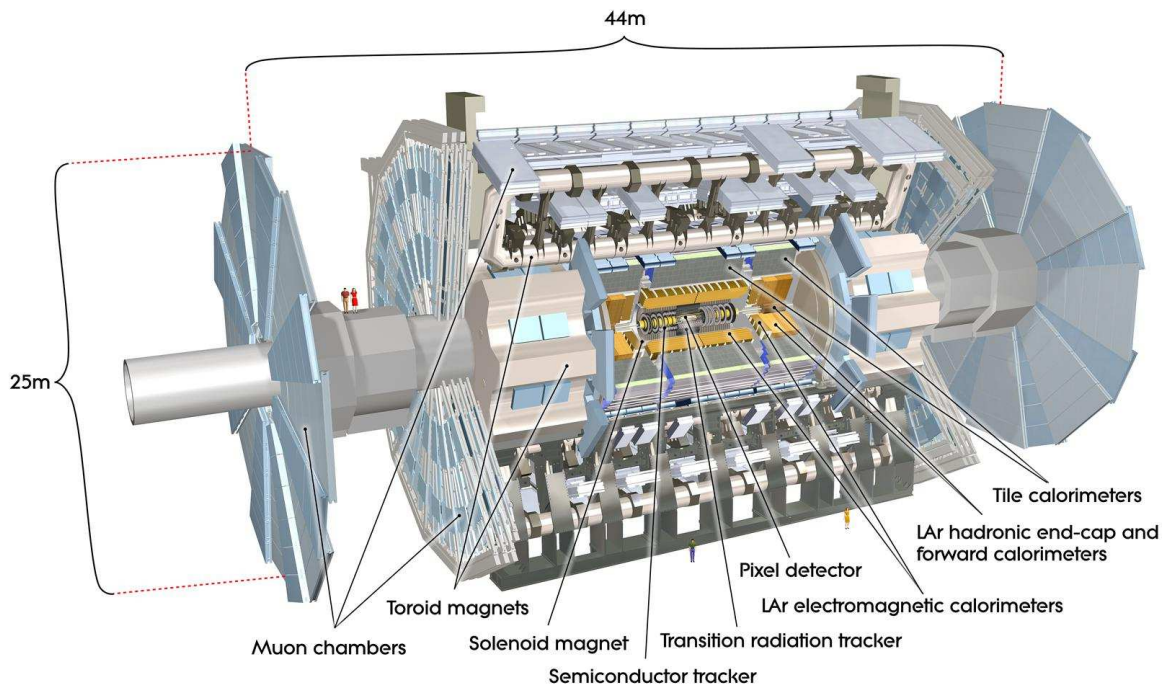


Figura 4 – O detector ATLAS e seus sub-sistemas [38].

O Detector de Trajetórias (*Inner Detector*), que está na camada mais interna, é subdividido em três sub-detectores (*Pixel Detector*, *Semiconductor Tracker* e *Transition Radiation Tracker*), com a função de determinar os traços das partículas carregadas, auxiliando na determinação do seu momento e posição [39].

O Espectrômetro de Múons (*Muon Spectrometer*) identifica e mede o momento dos múons, localizando-se na camada mais externa do detector, devido ao fato de os múons serem as únicas partículas detectáveis que alcançam distâncias tão grandes além do ponto de colisão [40].

No sistema de calorimetria do ATLAS (Figura 5), que está localizado na camada intermediária do detector, o Calorímetro de Argônio Líquido (LAr - *Liquid Argon*), também chamado de Calorímetro Eletromagnético, foi projetado para medir a energia das partículas que interagem de forma eletromagnética (elétrons e fótons) com seu material [41]. O Calorímetro Hadrônico (TileCal - *Tile Calorimeter*), que também é conhecido como Calorímetro de Telhas, foi projetado para mensurar a energia das partículas mais propícias a interagirem de forma hadrônica (principalmente hádrons neutros) com seu material [42].

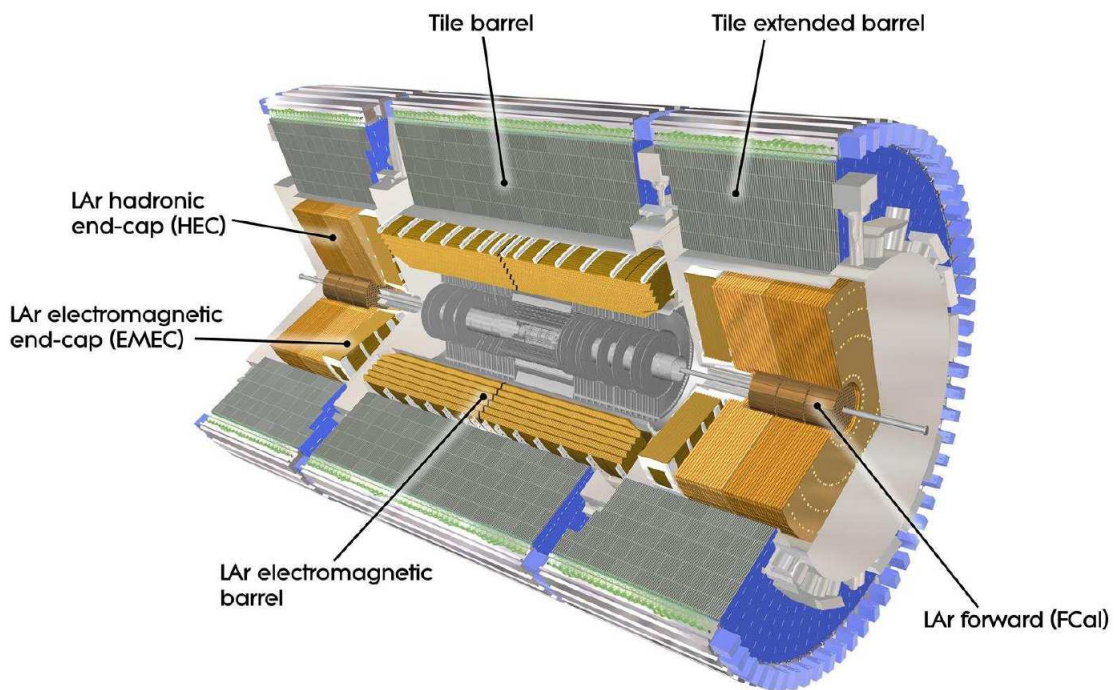


Figura 5 – Sistema de calorimetria do ATLAS [43].

Na prática, os calorímetros têm a função de absorver, amostrar e medir a energia das partículas que os incidem. Estas partículas, ao entrarem em contato com o material dos calorímetros, geram um chuveiro de partículas, onde parte de sua energia é depositada, coletada e medida, o que é possível devido aos calorímetros serem compostos por um material denso, formando barreiras que permitem que as partículas sejam absorvidas por completo [39]. Os múons, de alta energia, não são absorvidos pelo experimento, depositando apenas uma pequena parte de sua energia nos calorímetros. Quando o processo de chuveiro é iniciado, as partículas sofrem decaimentos, produzindo partículas de menor energia, e este processo se dá até a absorção total da energia da partícula pelo calorímetro [44].

Na Figura 6 é ilustrado um esquema com diversos tipos de partículas interagindo com os sub-detectores do ATLAS, onde é possível identificar e caracterizar as partículas e acordo com as características mensuradas quando tais partículas atravessam as camadas de detecção do experimento.

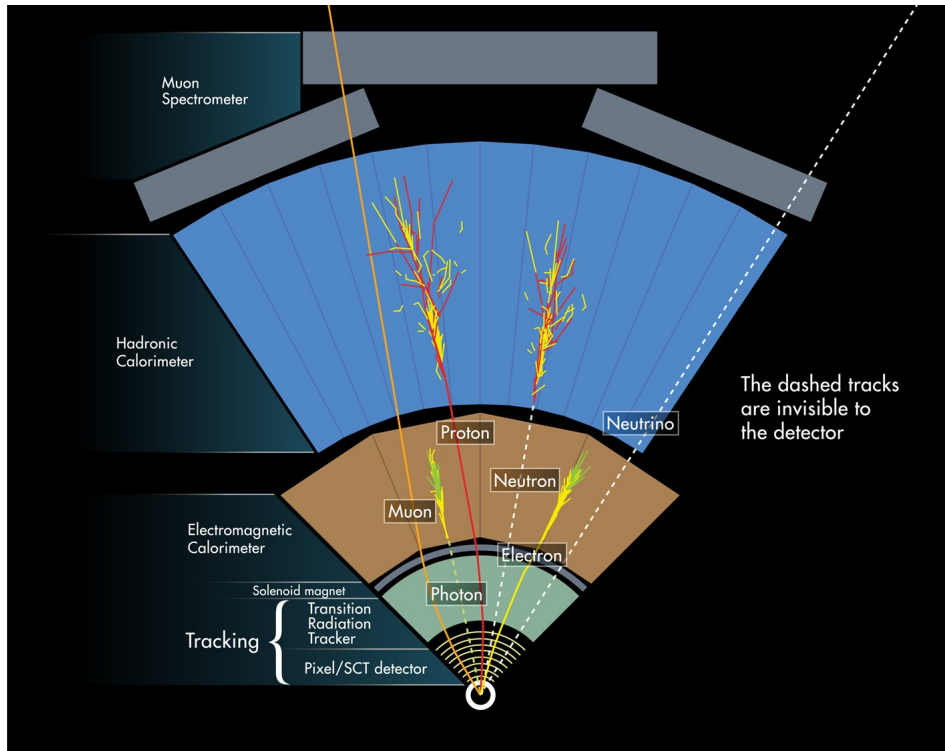


Figura 6 – Partículas interagindo com os sub-detectores do ATLAS [45].

2.1.1 O Calorímetro Hadrônico e o Sistema de *Trigger*

O Calorímetro Hadrônico do ATLAS possui o aço como material absorvedor e telhas cintilantes como material amostrador de energia. Estas telhas são excitadas quando partículas carregadas as atravessam, ocorrendo a produção de fótons. Com isto, há a conversão de luz em sinal elétrico, que se dá por células fotomultiplicadoras também conhecidas como PMT's (*PhotoMultiplier Tube*). A luz gerada nos cintiladores é levada às células multiplicadoras por fibras óticas, localizadas nas duas extremidades das telhas, para que haja redundância na leitura e, conseqüentemente, maior confiabilidade [46].

Com a finalidade de se obter um sinal de pulso padrão, o sinal convertido pelo PMT é repassado a um circuito conformador de pulsos, e o pulso final apresenta um formato determinístico e uma amplitude proporcional à energia que a partícula depositou. As células do TileCal são formadas por conjuntos de telhas cintilantes, onde cada célula é lida por duas PMT's, visando redundância, e cada PMT corresponde a um canal de leitura, havendo, no total, cerca de 10.000 canais de leitura [44].

Na Figura 7 está ilustrado um módulo do TileCal com vista tri-dimensional, onde é possível observar as telhas dispostas perpendicularmente à direção do feixe de partículas.

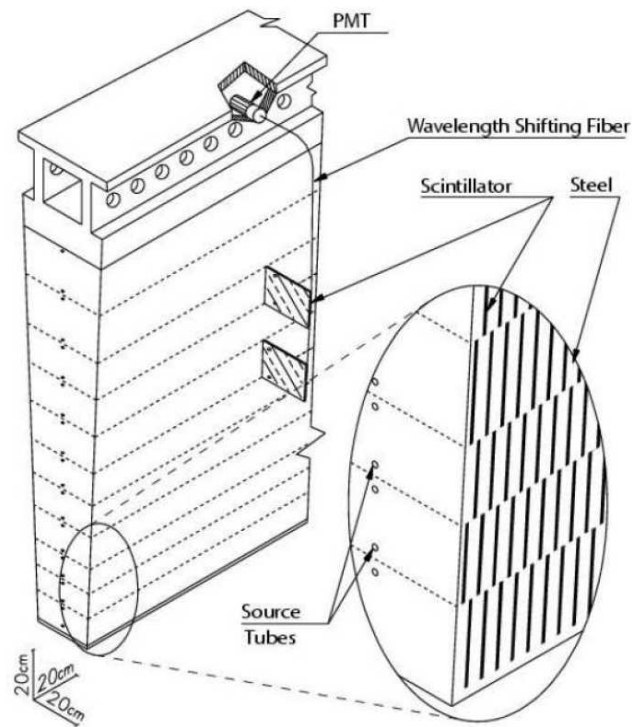


Figura 7 – Esquema de um módulo do TileCal [34].

O pulso característico do TileCal está ilustrado na Figura 8. Ele é composto por 7 amostras espaçadas de 25 ns (durando cerca de 150 ns), digitalizado a uma taxa de 40 MHz, sincronizada com a taxa de colisão [47].

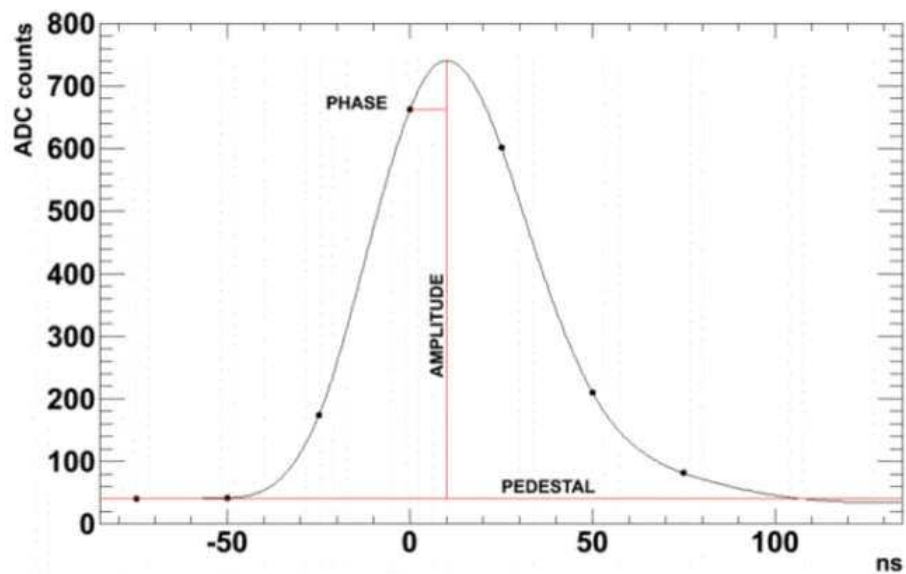


Figura 8 – Formato de um pulso característico do TileCal [47].

O principal objetivo do TileCal é contribuir na reconstrução do padrão de deposição de energia dos chuveiros produzidos pelas interações próton-próton e auxiliar nos cálculos de momento transverso. Sua eletrônica inclui circuitos de *front-end* e digitalizadores de sinais, que são projetados de acordo com as características de alta velocidade e baixo ruído das suas células fotomultiplicadoras. Por fim, os sinais digitalizados correspondentes aos eventos no calorímetro são transferidos para *buffers* através de fibras ópticas [44].

As interações nos sub-detectores do ATLAS geram um enorme fluxo de dados: são cerca de 70 TB/s (terabytes por segundo) de informação, vindas da colisão entre dois feixes de prótons com 2.808 grupos de 10^{11} partículas cada um, viajando em direções opostas, a uma velocidade de 99,9998% da velocidade da luz, colidindo a uma taxa constante de 40 MHz, de forma que a cada 25 ns ocorre um evento de colisão [48].

A alta taxa de eventos, em conjunto com as centenas de milhares de canais de leitura dos subdetectores do ATLAS, fazem com que seja impossível realizar o armazenamento e processamento de todos estes dados. Assim, um sistema de filtragem (*trigger*) na aquisição dos dados é necessário, a fim de selecionar quais eventos podem ser armazenados e descartar os dados irrelevantes para os eventos físicos de interesse.

Essa filtragem é realizada pelo sistema de seleção de eventos e aquisição de dados do ATLAS. Este sistema de *trigger* é dividido em níveis conectados em cascata, que operam *online* e, em cada um deles, o critério de seleção é refinado [49].

Tal sistema é importante pois, a partir de uma taxa de *Bunch Crossing* (BC) inicial de 40 MHz (momento em que as partículas colidem) no sistema de *trigger* do ATLAS, a taxa de eventos selecionados deve ser reduzida para até 200 Hz no último nível, visando o armazenamento para posterior análise [50].

O primeiro nível de *trigger* [51], também chamado de L1Calo, é subdividido em *trigger* de calorimetria e *trigger* de múons, realizando a filtragem das informações coletadas dos sistemas de calorimetria e das câmaras de múons, respectivamente. Para executar suas rotinas de filtragem, ele identifica as assinaturas básicas da física de interesse, baseando sua decisão na multiplicidade de objetos encontrados, que podem ser objetos locais (múons, elétrons e jatos) ou globais (energia faltante e energia total).

Devido à alta velocidade requerida de processamento, a filtragem do L1Calo é implementada em *hardware*, e a mesma reduz a taxa de eventos de entrada de 40 MHz para até 100 kHz. Este nível de *trigger* é o foco do presente trabalho.

2.2 DESAFIO: O EFEITO *PILE-UP*

Atualizações graduais estão previstas para ocorrer no LHC nos próximos anos, de forma a elevar a densidade de feixes de prótons, fenômeno que no âmbito de Física de Altas Energias é conhecido como o aumento de luminosidade do acelerador [52, 53]. Com isto, a taxa de interações entre as partículas e a quantidade de partículas elementares detectadas cresce, aumentando assim a probabilidade da ocorrência de fenômenos raros.

Como o tempo de resposta da eletrônica de leitura nos calorímetros é maior do que o período entre as colisões, o aumento na taxa de eventos intervirá principalmente na sobreposição entre sinais provenientes de eventos subsequentes, ocasionando a deformação do sinal recebido e dificultando a equalização do canal. Tal efeito é conhecido como empilhamento de sinais ou *pile-up* [10, 54], o qual é ilustrado na Figura 9.

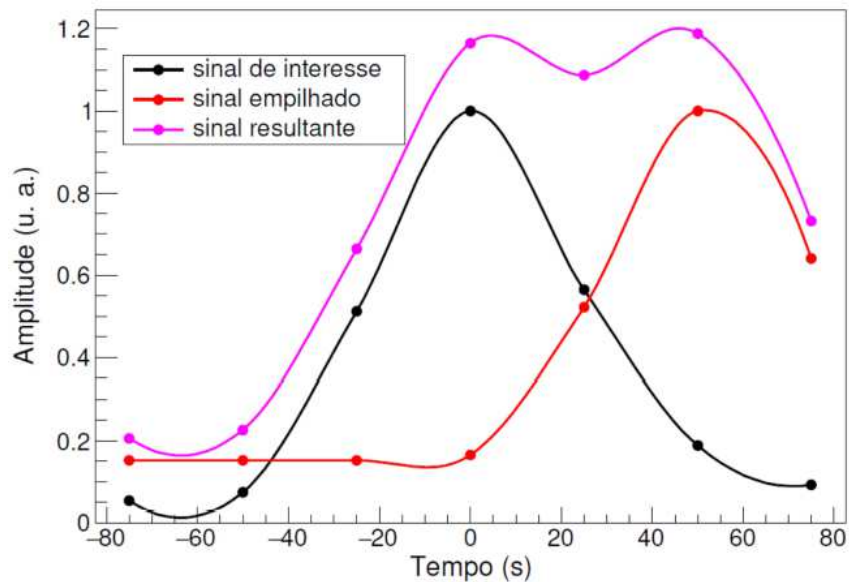


Figura 9 – Efeito *pile-up* no TileCal [10].

A Figura 9 exemplifica o *pile-up* onde, primeiro, foi gerado o sinal de interesse (cor preta), resultante de uma primeira colisão, que sensibilizou uma determinada célula do TileCal. Depois de 50 ns, a mesma célula foi novamente sensibilizada devido a uma nova colisão, gerando um segundo sinal (cor vermelha).

Como se tratam de eventos muito próximos e inferiores ao período de 150 ns (o necessário para a identificação de um pulso conformado), forma-se um sinal resultante (cor magenta), que é o efeito do empilhamento entre os outros dois sinais [10].

Os dados utilizados nas simulações e testes realizados neste trabalho são gerados por um *Toy Monte Carlo* [55], que já foi empregado em [56, 57, 58, 59]. Com isso, é possível simular sinais equivalentes aos gerados nos calorímetros do ATLAS, com a vantagem de possuir maior controle nos parâmetros relacionados ao processo de empilhamento de sinais.

Dentre estes parâmetros, um importante a ser ressaltado é o valor de ocupação, que representa a porcentagem da relação entre o número de *bunches* em que houve deposição de energia, em uma determinada célula do calorímetro, e o número de *bunches* total em uma janela de aquisição. Por exemplo, uma ocupação de 50% indica que, em média, um *bunch* a cada dois sofreu deposição de energia em um determinado canal do TileCal.

Na Figura 10 é possível ver um gráfico cujos sinais foram gerados através de simulações com o *Toy Monte Carlo*, realizadas no *software* MATLAB [60]. Em verde é representado o sinal visto pela eletrônica de leitura do calorímetro, após passar pelo circuito conformador que convolui o sinal alvo com o pulso característico do TileCal, e o sinal acaba sendo deformado devido ao efeito *pile-up*. Ainda nesta figura, o valor alvo de deposição de energia em cada *Bunch Crossing* é representado pela amplitude do sinal azul.

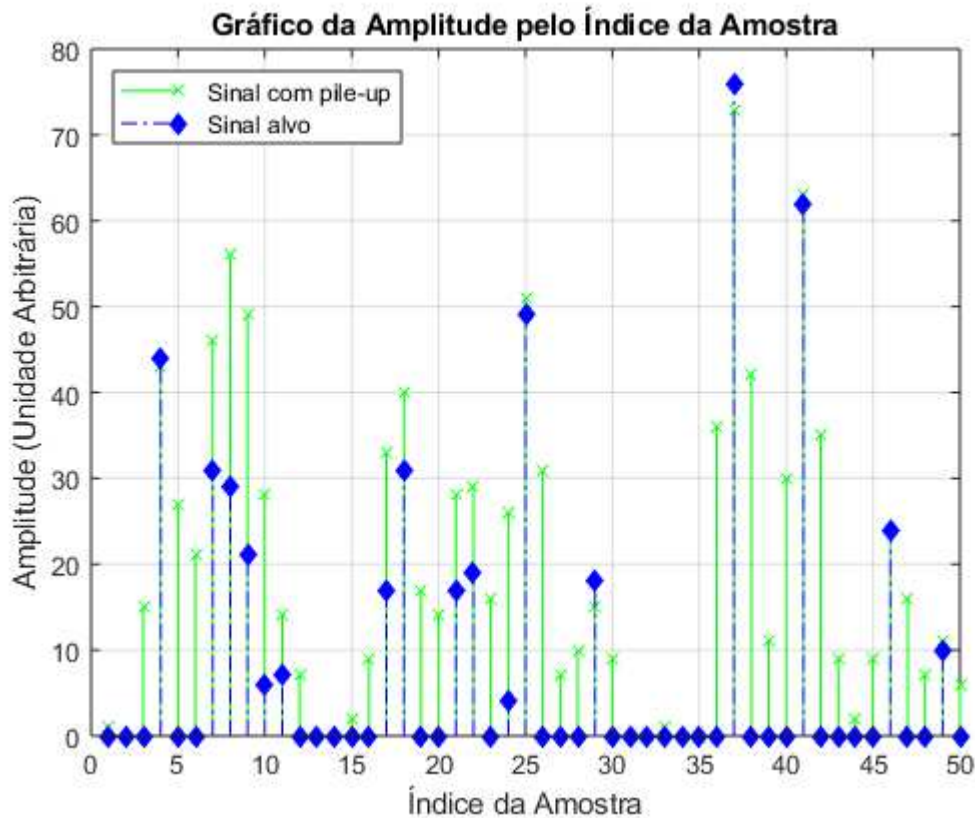


Figura 10 – Sinais do TileCal simulados usando *Toy Monte Carlo*.

O objetivo é recuperar o sinal alvo com o menor erro possível, a partir do sinal obtido na eletrônica de leitura do calorímetro, para propagar a informação pelo sistema de *trigger*. Neste contexto, o desafio do presente trabalho é realizar implementações em FPGA de processadores dedicados, visando a reconstrução *online* dos sinais no primeiro nível de *trigger* do TileCal, utilizando algoritmos sensíveis ao efeito *pile-up*.

3 REVISÃO BIBLIOGRÁFICA

O método atualmente utilizado para a reconstrução dos sinais no primeiro nível de *trigger* do TileCal é o Filtro Casado [47]. Esse tipo de filtro possui coeficientes proporcionais aos pesos de cada amostra do pulso característico do sinal, para sinais determinísticos, imersos em ruído branco gaussiano [61]. Tal técnica é baseada em correlacionar o pulso característico com o sinal do próprio pulso de interesse, uma vez que o efeito produzido pelo Filtro Casado é a maximização da relação sinal ruído [51].

Dessa forma, o desempenho do Filtro Casado depende do prévio conhecimento da forma do pulso em meio ao ruído. Tal método não é indicado para a operação em ambientes de alta luminosidade, uma vez que o efeito *pile-up* modifica a forma do pulso característico do sinal. Nesse contexto, novas técnicas de estimação de energia e detecção de sinais precisam ser investigadas.

Propostas recentes para a detecção de sinais se baseiam na modelagem da cadeia eletrônica do TileCal como um canal de comunicação, de forma que a estimação de energia é obtida pela deconvolução (ou equalização) desse canal [10].

No trabalho [13], é mostrado que técnicas de deconvolução baseadas em filtros FIR (*Finite Impulse Response*) [14] possuem simples implementação, além de produzirem uma melhor estimação em relação ao Filtro Casado para condições de alta taxa de eventos, apresentando bons resultados e baixo custo computacional. Esta metodologia usa a minimização do erro médio quadrático (RMS, do inglês *Root Mean Square*) entre os dados e o modelo para a determinação dos coeficientes do filtro [62]. Porém, mesmo com melhor desempenho nessas condições, a desvantagem de tal abordagem é que a deconvolução cria artefatos em amostras adjacentes, gerando falsos positivos no sistema de *trigger*.

Buscando evitar o problema da geração de falsos positivos, uma outra abordagem através de métodos baseados em um modelo matricial de sobreposição de sinais determina a amplitude dos diversos sinais empilhados dentro de uma janela de aquisição, por meio de algoritmos iterativos, seguindo um critério pela busca da esparsidade dos dados reconstruídos [63]. Tal abordagem apresenta desempenho superior ao método baseado em filtros FIR no que diz respeito à acurácia da reconstrução da informação [58], porém o custo computacional é alto, representando um desafio na proposição de algoritmos a serem implementados em ambiente embarcado.

Nesse contexto, teorias de SR (*Sparse Representation*), ou Representação Esparsa de dados, buscam soluções mais eficientes [64], em termos de implementação. O método SSF (*Separable Surrogate Functionals*) [17] implementa uma forma iterativa na busca pela SR, utilizando apenas operações de soma e produto, podendo, com isso, ser implementado de forma eficiente em FPGAs modernas.

O uso de técnicas lineares para estimação de energia mostrou-se eficiente quando comparado ao método atualmente em uso no sistema de *trigger*, porém, com aumento do *pile-up*, a amplitude dos sinais pode ser afetada por sinais secundários defasados, modificando a natureza do ruído de medição e o tornando não-gaussiano. Logo, os métodos lineares de estimação da amplitude do pulso não desempenham essa tarefa de forma ótima nesse ambiente, o que motiva a implementação de métodos não-lineares.

Os estudos para a implementação de um procedimento de estimação de amplitude por meio de Redes Neurais Artificiais *feedforward perceptron* multicamadas (FF-MLP) foram apresentados em [65]. Tal abordagem foi comparada com os métodos baseados em filtragem inversa utilizando filtros FIR. O estimador neural apresentou erro menor na reconstrução de energia em ambientes com alta ocorrência de *pile-up* e atende os requisitos de implementação em *hardware* dedicado.

Tanto a técnica baseada em Representação Esparsa de dados quanto as Redes Neurais Artificiais (RNA) se mostraram promissoras. A primeira, necessitando apenas conhecer a forma do pulso característico do sinal, baseado em uma modelagem que faz a deconvolução ao mesmo tempo em que diminui a presença de falsos positivos explorando a esparsidade do problema, já a RNA trata o ruído não linear gerado pela sobreposição dos pulsos e desvio de fase, fazendo um processo de deconvolução mais acurado, porém necessita de um ambiente de simulação mais preciso. Cada técnica tem suas vantagens e desvantagens que serão descritas neste trabalho, onde será realizada a implementação de ambas em FPGA e serão também comparados os recursos computacionais utilizados.

3.1 ESTIMAÇÃO DE ENERGIA USANDO REPRESENTAÇÃO ESPARSA

O trabalho [57] propõe um modelo onde, dado um pulso de referência normalizado¹ do calorímetro, representado pelo vetor \mathbf{h} , e uma sequência \mathbf{x} , que representa os valores de energia a serem reconstruídos, a convolução entre estes dois sinais é o sinal de leitura \mathbf{r} amostrado na eletrônica de *front-end* do calorímetro, cujo *clock* é síncrono com a taxa de colisões do acelerador (40 MHz).

Para a matriz de convolução \mathbf{H} , cujas colunas contêm versões deslocadas do sinal de referência normalizado \mathbf{h} , uma formulação matricial para o processo de convolução é:

$$\mathbf{r} = \mathbf{H}\mathbf{x} \quad (3.1)$$

O processo de deconvolução consiste em reconstruir a sequência \mathbf{x} quando \mathbf{r} e \mathbf{H} são conhecidos, o que é o caso no problema em questão. Como o tamanho do vetor \mathbf{r} é maior do que o do vetor \mathbf{x} , existem infinitas soluções para este sistema de equações.

¹ Pulso de referência com amplitude unitária.

O trabalho [56] mostra que, para a deconvolução de sinais impulsivos, a representação mais esparsa de \mathbf{x} é a melhor escolha.

Em [64] é demonstrado que a solução SR da Equação 3.1 é obtida resolvendo-se, para $0 \leq \ell \leq 1$, o problema:

$$(P_\ell) : \min_{\mathbf{x}} \|\mathbf{x}\|_\ell^\ell \quad \text{sujeito a} \quad \mathbf{r} = \mathbf{H}\mathbf{x} \quad (3.2)$$

Onde a norma- ℓ do vetor \mathbf{x} é dada por:

$$\|\mathbf{x}\|_\ell^\ell = \sum_i |x_i|^\ell \quad (3.3)$$

Fazendo $\ell = 1$, o Problema P_1 resulta em um problema típico de Programação Linear ou LP, do inglês *Linear Programming* [66]. Nesse contexto, no trabalho [56] foi proposto o uso de LP em SR, tendo como foco a reconstrução de energia, onde o desempenho de tal método se mostrou superior a outros métodos janelados de deconvolução. Porém, destaca-se que o foco daquele trabalho foi a reconstrução *offline* de energia, uma vez que LP apresenta um custo computacional muito elevado.

Nos trabalhos [63] e [67], métodos modernos de SR, usando P_1 , foram analisados e uma implementação em FPGA de uma versão adaptada do método conhecido como SSF mostrou-se bem sucedida. Esse método é baseado no Problema P_1 , empregando-se uma relaxação na restrição, de modo a permitir um erro quadrático pequeno ϵ_0 para englobar problemas com adição de ruído.

Dessa forma, temos:

$$(P_{1,\epsilon_0}) : \min_{\mathbf{x}} \|\mathbf{x}\|_1^1 \quad \text{sujeito a} \quad |\mathbf{r} - \mathbf{H}\mathbf{x}|^2 < \epsilon_0 \quad (3.4)$$

O Problema (P_{1,ϵ_0}) pode ser transformado em um problema de otimização sem restrição, usando um multiplicador de Lagrange λ adequado, onde ϵ_0 é absorvido por λ :

$$(P_{1,\lambda}) : \min_{\mathbf{x}} \|\mathbf{x}\|_1^1 + \lambda |\mathbf{r} - \mathbf{H}\mathbf{x}|^2 \quad (3.5)$$

Na literatura existem diversos algoritmos que propõem uma solução para $(P_{1,\lambda})$, por meio de métodos iterativos de Coordenadas Descendentes (CD) [64]: dado um vetor inicial \mathbf{x}_0 , o valor ideal \mathbf{x}_{opt} pode ser inferido recursivamente em pequenos passos. No entanto, o uso de métodos de CD padrão, como o método Newton-Raphson [68], torna-se proibitivo para $(P_{1,\lambda})$, devido a descontinuidade da norma l_1 .

Em [17] é proposta a inserção de termos que não alteram a posição das coordenadas \mathbf{x}_{opt} em $(P_{1,\lambda})$, mas que são capazes de dividir o problema multivariado em problemas unidimensionais separados, que podem ser resolvidos por partes. A equação resultante é denominada uma função substituta e o respectivo método é o SSF.

Após manipulações algébricas e definindo o tamanho do passo em direção ao mínimo como μ , o procedimento iterativo pode ser compactado como:

$$\mathbf{x}_{i+1} = S_\lambda \left(\mathbf{x}_i + \mu [\mathbf{H}^T (\mathbf{r} - \mathbf{H}\mathbf{x}_i)] \right) \quad (3.6)$$

É proposta, em [63], uma modificação na função de *Shrinkage*², $S_\lambda(\theta)$, para permitir apenas a reconstrução positiva da energia, que é o caso na calorimetria (Figura 11).

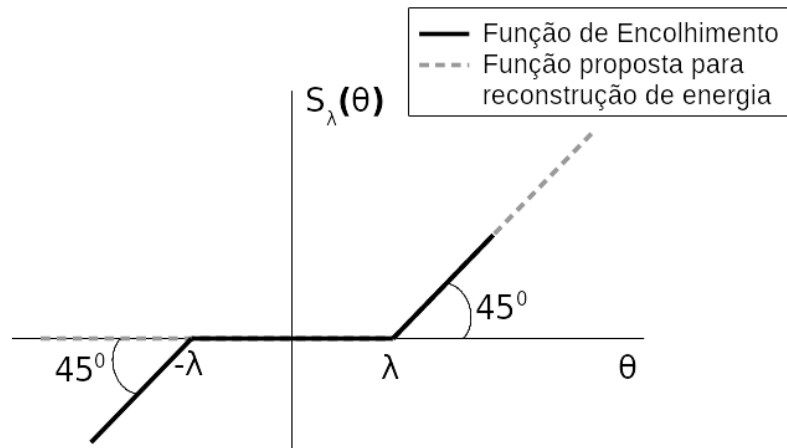


Figura 11 – Função proposta para reconstrução de energia [63].

Com isso, a função resultante pode ser implementada simplesmente por meio de uma subtração seguida por uma operação de limiar. O argumento na Equação 3.6 é identificado como uma iteração do método Gradiente Descendente (GD) linear e compreende apenas operações de soma e multiplicação. Os parâmetros λ e μ da Equação 3.6 tiveram suas calibrações realizadas com dados de simulação do TileCal, sendo fixados em 0 e 0.25, respectivamente [63]. A vantagem de fixar tais parâmetros é que circuitos adicionais de multiplicação e de comparação no *hardware* desenvolvido não serão necessários.

3.1.1 Inicialização Aprimorada do Algoritmo por meio da Matriz Pseudo-Inversa

Para que o processo iterativo do método definido pela Equação 3.6 possa ser realizado, o vetor \mathbf{x} é inicializado com uma janela contendo os dados vindos do conversor analógico-digital do primeiro nível de *trigger* do TileCal. Porém, foi realizado um estudo no decorrer do presente trabalho, publicado em [69], que avalia uma forma alternativa para a inicialização do vetor \mathbf{x} , através de um pré-processamento a ser realizado, na respectiva janela de dados, antes da etapa iterativa do algoritmo, visando reduzir o número de iterações necessárias para a convergência do método.

² A função de *Shrinkage* é a função de encolhimento unidimensional, que deve ser aplicada a cada componente no argumento separadamente.

Tal procedimento se baseia em, respeitando a restrição de manter o modelo já proposto, deve-se obter uma solução aproximada de \mathbf{x} através da minimização da sua norma quadrática euclidiana [64], uma vez que a matriz de convolução \mathbf{H} da Equação 3.1 não possui inversa. Assim, esse problema é definido por:

$$(P_2) : \min_{\mathbf{x}} \|\mathbf{x}\|_2^2 \quad \text{sujeito a} \quad \mathbf{r} = \mathbf{H}\mathbf{x} \quad (3.7)$$

Utilizando multiplicadores de Lagrange λ para o conjunto de restrições do Problema (P_2) , o Lagrangiano [70] pode então ser definido como:

$$\mathcal{L}(\mathbf{x}) = \|\mathbf{x}\|_2^2 + \lambda^T(\mathbf{H}\mathbf{x} - \mathbf{r}) \quad (3.8)$$

Realizando a derivada parcial de $\mathcal{L}(\mathbf{x})$ em relação a \mathbf{x} , obtém-se:

$$\frac{\partial \mathcal{L}(\mathbf{x})}{\partial \mathbf{x}} = 2\mathbf{x} + \mathbf{H}^T \lambda \quad (3.9)$$

Cuja solução, obtida para \mathbf{x} , após realizadas manipulações algébricas, é:

$$\begin{aligned} \hat{\mathbf{x}}_{opt} &= -\frac{1}{2}\mathbf{H}^T \lambda \rightarrow \mathbf{H}\hat{\mathbf{x}}_{opt} = -\frac{1}{2}\mathbf{H}\mathbf{H}^T \lambda = \mathbf{r} \rightarrow \lambda = -2(\mathbf{H}\mathbf{H}^T)^{-1}\mathbf{r} \\ \hat{\mathbf{x}}_{opt} &= -\frac{1}{2}\mathbf{H}^T \lambda \rightarrow \hat{\mathbf{x}}_{opt} = \mathbf{H}^T(\mathbf{H}\mathbf{H}^T)^{-1}\mathbf{r} \rightarrow \hat{\mathbf{x}}_{opt} = \mathbf{H}^+ \mathbf{r} \end{aligned} \quad (3.10)$$

Tal demonstração está detalhada em [64]. Nesta solução, a matriz \mathbf{H}^+ é definida como matriz pseudo-inversa de \mathbf{H} e a proposta é realizar a inicialização do vetor \mathbf{x} como $\mathbf{H}^+ \mathbf{r}$, uma vez que este pré-processamento garante que o vetor \mathbf{x} seja inicializado mais próximo da solução e, assim, a convergência do método SSF ocorre mais rapidamente [69].

3.2 REDES NEURAIS ARTIFICIAIS

As Redes Neurais Artificiais são inspiradas no cérebro humano, que é um complexo sistema de processamento de informações, capaz de auto-organizar seus neurônios para executar funções de reconhecimento de padrões e controle motor [71].

O constituinte estrutural básico do cérebro é o neurônio, que é formado basicamente por dendritos, axônios e corpo celular. A comunicação sináptica [72] ocorre quando os neurônios transmitem e recebem informações entre si através dos neurotransmissores, que são substâncias químicas liberadas na junção entre os axônios e dendritos.

Nestas sinapses, a membrana do neurônio receptor se polariza e gera sinais que podem ou não ser transmitidos aos outros neurônios, a depender se o estímulo gerado foi ou não suficiente para gerar um disparo do neurônio. Trazendo esta ideia para o contexto da computação moderna, o processo de excitação e transmissão de informações pode ser representado pelo modelo de neurônio [73], ilustrado na Figura 12.

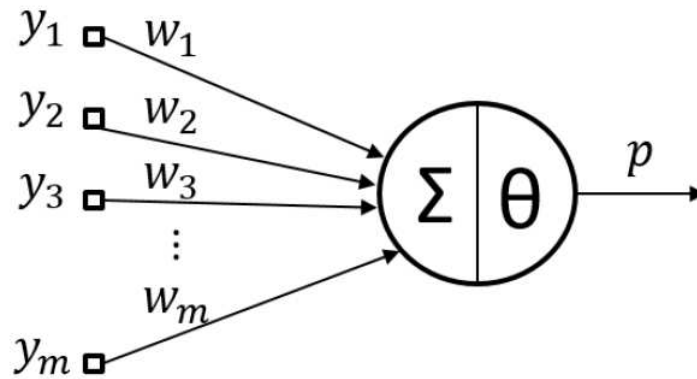


Figura 12 – Modelo de um neurônio [73].

Neste modelo, \mathbf{y} são as entradas que representam os estímulos recebidos pelos dendritos, \mathbf{w} emula o comportamento das sinapses e \mathbf{p} representa um axônio.

Este processo é descrito pela Equação 3.11, onde o efeito da sinapse \mathbf{i} no neurônio pós-sináptico é computado por $\mathbf{y}_i \mathbf{w}_i$. Após somar o efeito de todas as sinapses em cada neurônio, a saída será ativada ou não de acordo com a comparação com o limiar θ .

$$p = \sum_{i=1}^n y_i w_i \geq \theta \quad (3.11)$$

Ainda neste contexto, há também as funções de ativação, que introduzem não-linearidades nas RNAs, permitindo que elas capturem relações complexas nos dados que as transformações lineares por si só não podem representar. Estas funções determinam a saída do neurônio com base em sua entrada ponderada e um *bias* pode ser utilizado. Elas funcionam como o limiar que controla se o neurônio deve "disparar", ou seja, se ele vai transmitir sua saída ou não. Dessa forma, funções de ativação com características distintas podem ser usadas, impactando na dinâmica de aprendizado e na convergência do modelo.

As Redes Neurais Artificiais podem ser representadas por diversos neurônios com suas respectivas entradas, conexões sinápticas e saídas. Dentre as arquiteturas de RNA [74] mais comuns, temos i) *perceptron*, composta por uma camada de entrada e uma de saída, sendo empregada em tarefas de classificação e regressão; ii) *perceptron* multicamadas, que possui pelo menos uma camada intermediária entre o nó de entrada e a camada de saída, chamada de camada escondida; iii) recorrentes, quando há realimentação da saída para a entrada, sendo indicadas para tratar sequências de dados, como processamento de linguagem natural e séries temporais; iv) convolucionais, quando usam operações de convolução para capturar padrões locais e hierárquicos, sendo implementadas em visão computacional e podendo processar dados com estrutura de grade, como imagens.

Uma rede neural *perceptron* multicamadas do tipo *feedforward*³ foi desenvolvida no trabalho de [65] e será discutida a seguir. No trabalho referenciado, estão detalhadas as análises de eficiência, a quantização, a escolha da figura de mérito, o erro máximo permitido, o treinamento, o ajuste dos pesos sinápticos e a limitação da função de ativação, que foi discretizada em seu domínio para possibilitar a sua implementação direta em LUT (*Look-Up Table*), a fim de ser armazenada em memória interna da FPGA. Vale ressaltar que o foco do presente trabalho não é o desenvolvimento do algoritmo ou as análises de eficiência, mas sim a sua implementação em um processador embarcado em FPGA.

3.2.1 Estimação Online e Não-Linear de Energia usando Redes Neurais

O processamento digital de sinais no tempo discreto pode ser implementado de maneira *offline*, parcialmente *offline* ou *online* (em tempo real). Na Figura 13, é possível observar o diagrama de um filtro FIR aplicado na estimação *online* e linear de energia. A janela de observação é composta por uma sequência de m registradores de deslocamento, que geram um atraso no sinal do canal de leitura do calorímetro. Os coeficientes C_m do filtro, que são constantes, são multiplicados por cada um dos valores y_m da janela de dados de entrada. A amostra y_m é referente ao valor de energia a ser recuperado pelo sistema de estimação, que é representado por x_m .

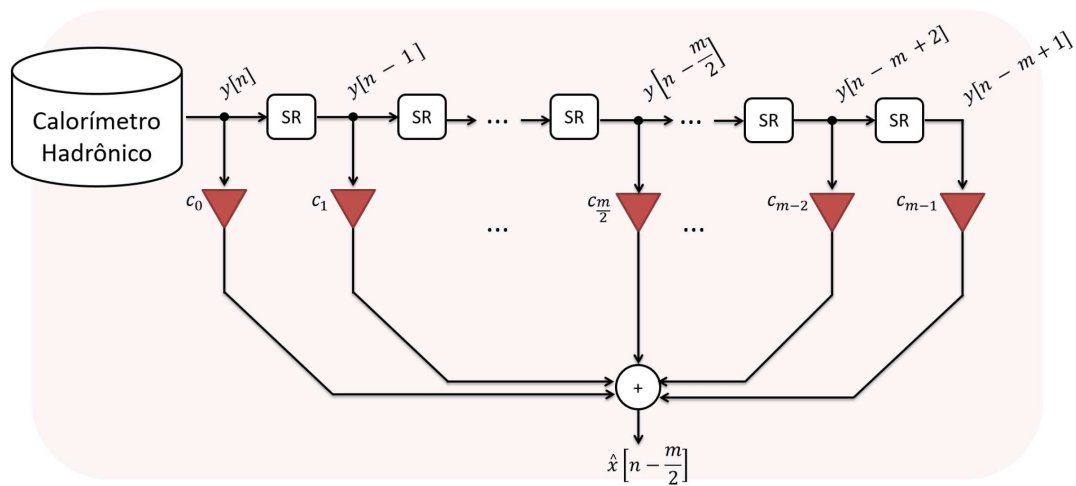


Figura 13 – Filtro FIR em aplicação de fluxo contínuo [16].

Após todos os resultados do filtro serem combinados linearmente, é obtido o valor estimado da energia referente à amostra central da janela ($y[n - \frac{m}{2}]$), que sofre distorções inerentes ao calorímetro, onde o resultado final é representado por $\hat{x}[n - \frac{m}{2}]$.

Este tipo de filtro possui uma latência de $\frac{m}{2}$ amostras em relação aos dados de saída do calorímetro, devido ao atraso temporal causado pelos registradores de deslocamento.

³ Os dados nas redes *feedforward* fluem em uma única direção, da camada de entrada para a camada de saída, sem realimentação.

Para realizar a estimação da amplitude do sinal de interesse utilizando Redes Neurais, em aplicação *online*, onde a estimativa é feita a cada nova amostra adquirida, podemos substituir as operações lineares de soma e produto do filtro FIR por uma combinação não linear através de um estimador neural, como ilustrado na Figura 14.

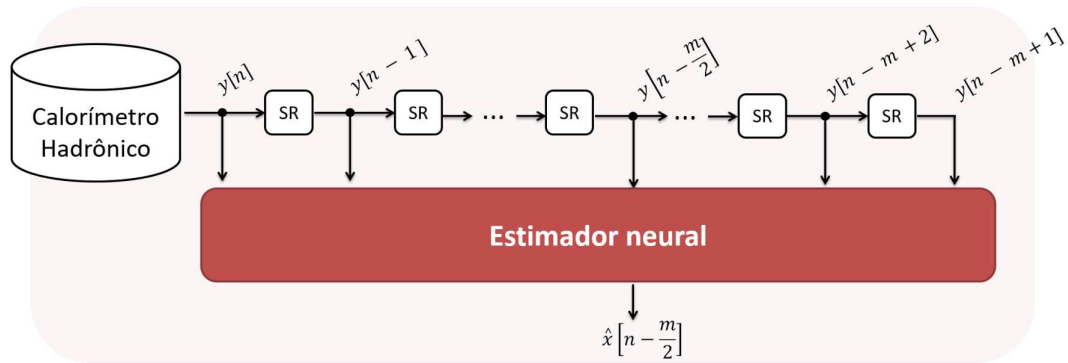


Figura 14 – Rede *feedforward* em aplicação de fluxo contínuo [16].

Para o sinal centrado em uma janela de observação de $\mathbf{m} + 1$ amostras, caso a rede possua um número de nós par, a saída estimada $\hat{\mathbf{x}}$ é referente a colisão central à janela $\frac{\mathbf{m}}{2}$, caso o número de nós seja ímpar, a saída estimada é referente a $\frac{\mathbf{m}+1}{2}$.

Na Rede Neural a ser implementada no presente trabalho, o número de neurônios e nós de entrada foram determinados a partir de simulações exaustivas [16], resultando em 4 neurônios para a camada escondida e dez nós de entrada ($\mathbf{m} = 9$). A Função Sigmoide foi utilizada para os neurônios na camada escondida e, para a saída da rede, foi utilizado um combinador linear das saídas dos 4 neurônios da camada oculta somados a um *bias*.

Será proposto também neste trabalho uma modificação na RNA, onde serão utilizadas Séries de Taylor para aproximação da Função Sigmoide, visando reduzir o custo computacional da implementação. Os detalhes serão discutidos na Seção 5.

Por fim, como esta dissertação trata de dois sistemas muito diferentes, foi optado por desenvolver um núcleo de processamento usando o SAPHO⁴ (*Scalable-Architecture Processor for Hardware Optimization*), um processador parametrizável e de código aberto desenvolvido no Núcleo de Instrumentação e Processamento de Sinais da Universidade Federal de Juiz de Fora (NIPS/UFJF), que realiza operações por meio de circuitos aritméticos em ponto-fixa e ponto-flutuante [75]. Tal ferramenta será detalhada a seguir.

⁴ Código fonte disponível em: <https://github.com/nipscernufjf/SAPHO>

4 SAPHO: UM PROCESSADOR AUTO-ESCALÁVEL

Para permitir uma implementação viável em *hardware* de algoritmos complexos, grande parte dos sistemas embarcados atuais utilizam Processadores *Soft-Core* (PSCs) [76].

Dentre os diversos PSCs disponíveis, tanto nos comerciais quanto nos de código aberto [77], uma característica comum a eles é a sua arquitetura fixa, de forma que, independente da complexidade do programa embarcado, a mesma quantidade de recursos em *hardware* é alocada. Além disso, o tamanho da palavra de dados é fixada (geralmente em 32 bits), sendo, na maioria das vezes, superdimensionada. Um exemplo de PSC comercial com ferramentas de desenvolvimento amigáveis é o NIOS II [78], da Intel, que é parametrizável e contém inúmeras formas de configuração, para atender diversas faixas de interesse, tendo baixo custo de *hardware* a alto desempenho.

Neste contexto, foi desenvolvido o SAPHO (*Scalable-Architecture Processor for Hardware Optimization*), um PSC baseado em uma arquitetura Harvard [79] e com conjunto de instruções reduzido (*Reduced Instruction Set Computer* - RISC) [80].

O SAPHO não possui uma microarquitetura fixa, ele é parametrizável de acordo com o código implementado. Sua Memória de Dados (MD) e Memória de Programa (MP) possuem o número de endereços auto-escalável, e a sua Unidade Lógico-Aritmética (ULA) pode ser configurada para operar com aritmética de ponto-fixo ou ponto-flutuante.

O *hardware* do processador foi desenvolvido em linguagem *Verilog* [82], podendo ser sintetizado em qualquer FPGA.

Além do processador, foram desenvolvidas também as ferramentas necessárias para programá-lo: um subconjunto da Linguagem C [81] denominado C^+ , um compilador **C**, um compilador **Assembler**¹ e uma interface gráfica de desenvolvimento. Algumas vantagens que podem ser destacadas a respeito deste processador são:

- Alocação de recursos de *hardware* auto-escalável em tempo de projeto, de acordo com o programa a ser executado.
- Arquitetura com três estágios de *pipeline* e ULA combinacional, o que permite a execução de uma instrução por ciclo de *clock*, sem quebra de *pipeline*, mesmo em rotinas de salto condicionais.
- Possibilidade de configurar a ULA para operar em ponto-fixo ou em ponto-flutuante, com tamanho de palavra configurável.
- A programação pode ser feita utilizando-se um subconjunto da linguagem C ou diretamente em linguagem *Assembly* [83].

¹ Normalmente, Assembler se refere a um montador para linguagem de máquina. Porém, no contexto do SAPHO, refere-se a um compilador da linguagem *Assembly* para *Verilog*.

4.1 HARDWARE

Com o intuito de descrever os detalhes de *hardware*, os blocos principais do SAPHO estão ilustrados na Figura 15, em que as linhas sólidas representam o fluxo dos dados, as linhas verdes representam o fluxo dos endereços e as linhas pontilhadas representam os sinais de controle. Os blocos destacados em azul são instanciados automaticamente, quando necessários, a depender do programa embarcado. Os demais blocos são sempre instanciados, uma vez que são necessários para o funcionamento do processador, independente de sua configuração. Dessa forma, os recursos de *hardware* são otimizados, customizando a implementação das operações específicas necessárias para executar o programa.

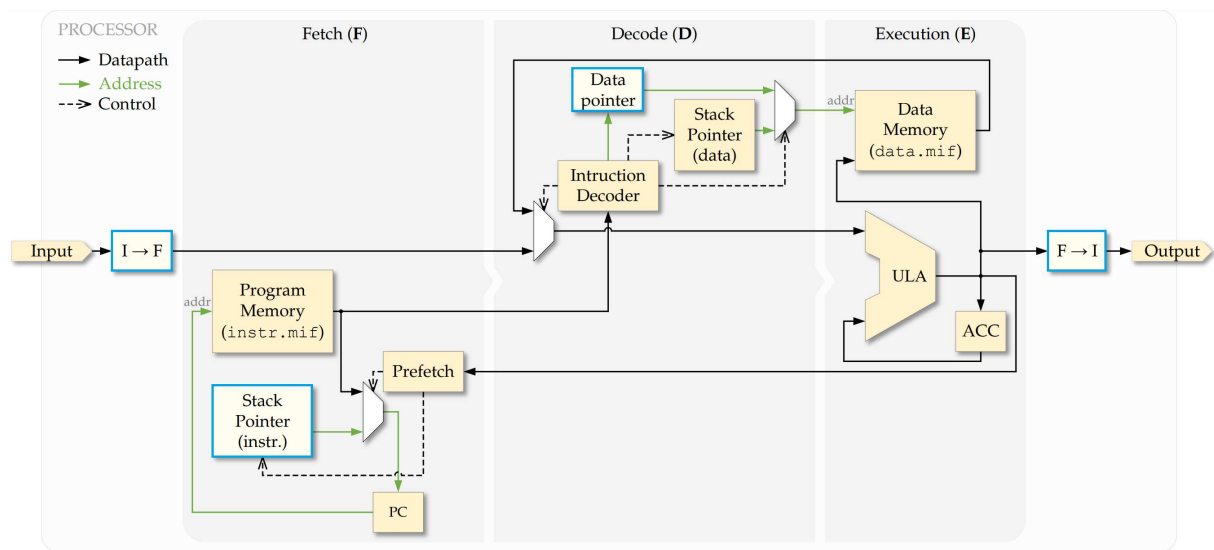


Figura 15 – Processador SAPHO e seus blocos principais.

Como as memórias são síncronas, o processador necessita de três estágios de *pipeline*: i) *fetch* (**F**), que busca a instrução; ii) *decode* (**D**), que busca do parâmetro na memória de dados e iii) *execution* (**E**), que executa a instrução.

Outro aspecto importante a ser ressaltado é que esta arquitetura foi desenvolvida para ser capaz de executar o programa sem quebra de *pipeline* ($F \rightarrow D \rightarrow E$). Pelo fato de a ULA ser combinacional, é possível acumular uma operação lógico-aritmética no registrador de dados (ACC) a cada ciclo de *clock*.

O SAPHO contém pilhas de dados e de instrução compartilhadas nas memórias de dados e de programa, respectivamente. Um ponteiro para estas pilhas (*Stack Pointer data* para pilha de dados e *Stack Pointer instr* para retorno de sub-rotinas) faz o controle da escrita e leitura. Caso a ULA seja configurada como ponto-flutuante, circuitos de transformação entre representação de ponto-fixe e ponto-flutuante são automaticamente instanciados nos barramentos de entrada e saída, para que a comunicação do processador com dispositivos de I/O seja em ponto-fixe, em Complemento a 2 [84].

Ainda na Figura 15, podemos destacar os seguintes blocos:

1. ***Int to Float ($I \rightarrow F$) e Float to Int ($F \rightarrow I$)***: Se a ULA for configurada para operar em ponto-flutuante, circuitos de transformação entre representação de ponto-fixe e ponto-flutuante serão instanciados automaticamente nos barramentos de entrada e de saída. O SAPHO usa uma representação customizada de ponto-flutuante que foi desenvolvida para otimizar implementações de *hardware* em FPGA. Mais detalhes sobre esta representação e suas vantagens estão descritos na Seção 4.3.
2. ***Memórias***: Elas possuem tamanhos dependentes do código desenvolvido e também são aceitos valores que não são potência de 2. O **Assembler**, ao compilar o código, é capaz de determinar o exato número de endereços necessário para armazenar todas as variáveis e instruções contidas no código e, dessa forma, parametrizar a instância das memórias. O conteúdo dessas memórias também é gerado pelo **Assembler** e salvo em arquivos (.mif), que são instanciados na FPGA, juntamente com o *hardware* do processador.
3. ***Prefetch***: Faz a busca da próxima instrução na memória enquanto o processador está executando a instrução atual, sendo responsável por separar o *opcode* do operando e por controlar o decodificador de instruções e o contador de programa.
4. ***Contador de Programa - PC***: Este bloco é responsável por apontar para a instrução a ser lida na MP. Durante a execução normal do programa, ele é incrementado a cada instrução que é executada. Quando uma instrução de salto é detectada, ele é carregado com um valor específico, relativo à próxima instrução. O tamanho em bits do PC, é configurado pelo **Assembler**, de acordo com o tamanho da memória de programa.
5. ***Stack Pointer - SP***: Existem dois blocos, um para dados e outro para instruções. Eles apontam para a posição do topo da pilha, que fica alocada nos endereços mais altos das memórias de dados e programa, respectivamente. A pilha de dados é acessada com as instruções **PUSH** e **POP**. Já a pilha de instruções é preenchida com o endereço de retorno para uma chamada de função feita com a instrução **CALL**. A pilha de instruções (e seu respectivo SP) é gerada somente se for identificado o uso de sub-rotinas (instruções **CALL** e **RETURN**).
6. ***Register File***: É um bloco é gerado sempre que o programa do usuário faz uso de *arrays*. Ele indexa os elementos do *array*, fazendo um *offset* na posição de memória do primeiro elemento do *array* para acessar a posição do elemento desejado.
7. ***Unidade Lógico-Aritmética***: Uma característica importante da ULA deste PSC é a grande flexibilidade de parametrização automática. Isto permite que o processador seja adaptado à função para a qual o mesmo foi designado.

Além de parâmetros que são escolhidos pelo usuário, em tempo de projeto, como aritmética de ponto-flutuante ou ponto-fixe e o tamanho da palavra de dados, a ULA é automaticamente parametrizada pelo **Assembler**, dependendo das instruções que foram geradas pelo compilador **C**.

A Tabela 3 mostra os circuitos que são criados automaticamente pelo **Assembler**. É possível observar, nesta tabela, quais circuitos são criados dentro da ULA e, baseado na configuração do processador, quais circuitos serão criados para a representação em ponto-fixe e em ponto-flutuante. As instruções destacadas em negrito são otimizações propostas no presente trabalho e serão melhor detalhadas na Seção 5.1.

Tabela 3 – Instruções e respectivos circuitos criados automaticamente.

Instrução	Circuito	ULA	P. Fixo	P. Flut.
DIV	Divisão	X	X	X
OR	Ou bit a bit	X	X	
LOR	Ou lógico	X	X	X
GRE	Maior que	X	X	X
MOD	Resto da divisão	X	X	
MLT	Multiplicação	X	X	X
LES	Menor que	X	X	X
EQU	Igual a	X	X	X
AND	And bit a bit	X	X	
LAN	And lógico	X	X	X
INV	Inversor bit a bit	X	X	
LIN	Inversor lógico	X	X	X
SHR	Shift para direita	X	X	
SHL	Shift para esquerda	X	X	
SRS	Shift com sinal	X	X	
CALL	Pilha de instrução		X	X
SRF	Endereçamento indireto		X	X
PSET	Set se for positivo		X	X
NORM	Normalização	X	X	
ABS	Valor absoluto	X	X	X
SIGN	Sinalização	X		X

4.2 SOFTWARE

O SAPHO gera o código *Verilog* do processador de forma automática, a partir do código escrito pelo programador em C^+ , em uma IDE (*Integrated Development Environment*) que foi desenvolvida na linguagem $C\#$. Essa IDE possui ferramentas para auxiliar na parametrização e desenvolvimento de processadores, tendo o intuito de executar os compiladores **C** e **Assembler** de forma transparente para o usuário.

Um tutorial para o desenvolvimento de processadores utilizando a versão mais atual do SAPHO está detalhado no Anexo B.

Na Figura 16 está a tela principal da interface gráfica do SAPHO. Nesta tela, é mostrada a hierarquia do projeto, onde é possível ver o número de processadores contidos no mesmo, bem como acessar o arquivos **C** (.c) de cada um.

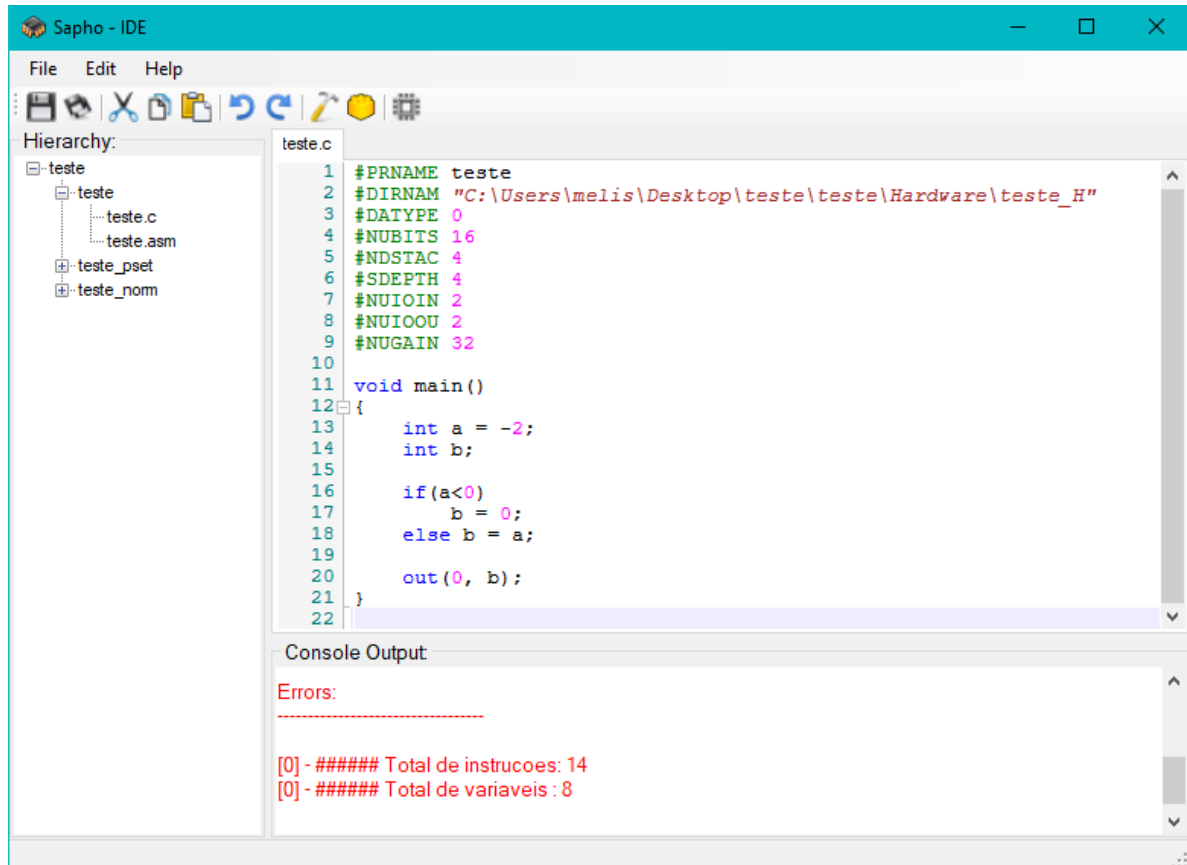


Figura 16 – Código em C^+ na IDE do SAPHO.

O projeto baseado em múltiplos processadores facilita o desenvolvimento de sistemas *multicore*, uma vez que é possível conectar facilmente os processadores através dos barramentos de I/O. Na barra superior são encontrados os atalhos para alguns comandos básicos, como o de compilar e o de adicionar um processador. Na janela console, são mostradas as mensagens resultantes dos processos de compilação e, se tudo estiver correto, o número de instruções e de variáveis são mostrados no final.

Neste código, em linguagem C^+ , a variável **a** foi declarada com valor -2 e é verificado se ela é negativa. Se **a** for negativa, a variável **b** recebe zero, caso contrário, **b** recebe o valor de **a**. Por fim, o valor de **b** é encaminhado à porta de saída de índice 0.

Após esse código ser compilado, é gerado um código em linguagem de máquina *Assembly*, a ser compilado pelo **Assembler** para gerar a parametrização do *hardware*. Tal exemplo pode ser visto na Figura 17.

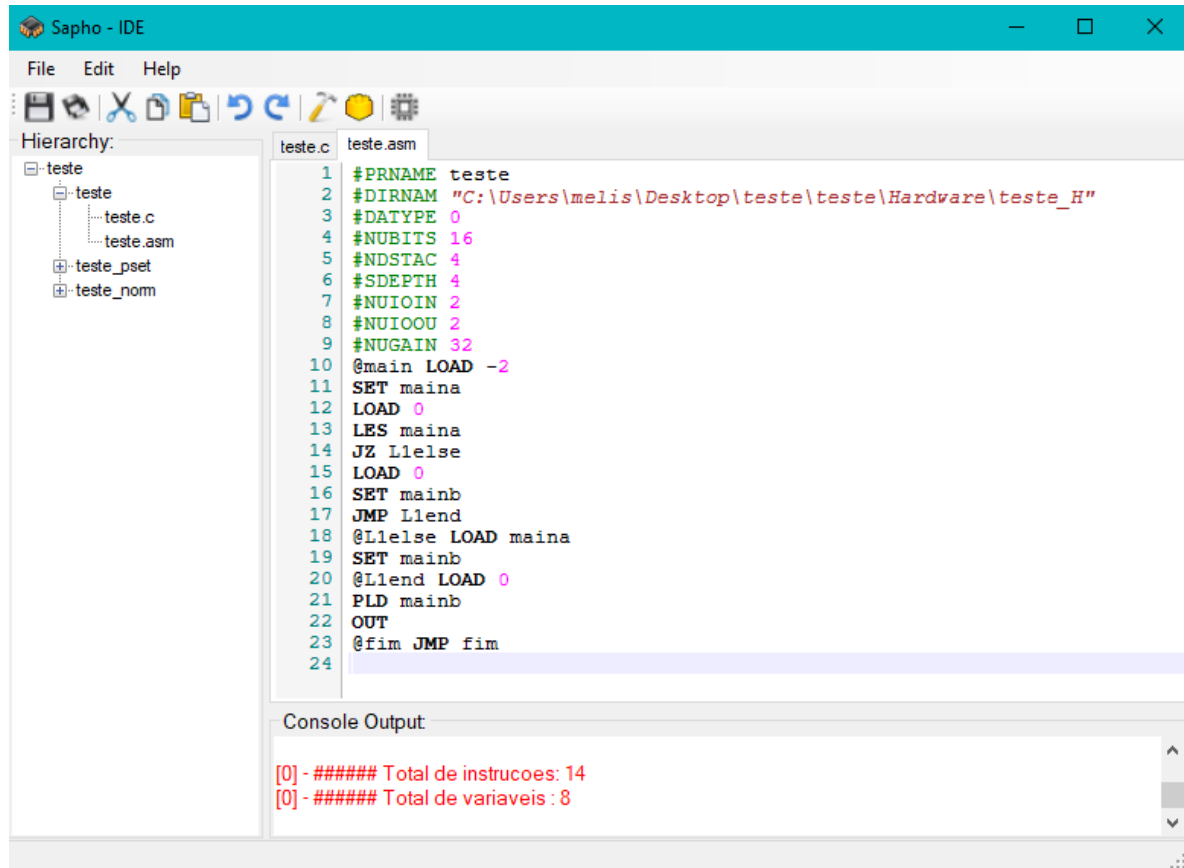


Figura 17 – Código em *Assembly* na IDE do SAPHO.

Ao adicionar um novo processador ao projeto, a janela denominada "*Configuration Wizard*", mostrada na Figura 18, é aberta para proporcionar a parametrização do mesmo.

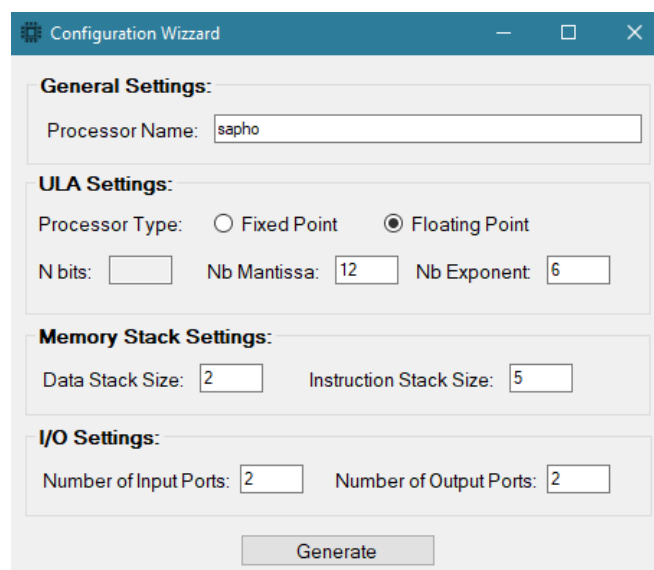


Figura 18 – Tela de configuração do processador na IDE do SAPHO.

Nesta janela, é possível escolher os seguintes parâmetros:

- Tipo: ponto-fixado ou ponto-flutuante.
- Número de bits (N bits) para representação, caso seja escolhido ponto-fixado, e número de bits para mantissa (Nb Mantissa) e expoente (Nb Exponent), caso seja escolhido ponto-flutuante [85].
- Tamanho da Pilha de Dados (*Data Stack Size*).
- Tamanho da Pilha de Instruções (*Instruction Stack Size*).
- Número de portas de entrada e saída.

Os parâmetros escolhidos são adicionados como diretivas de compilação ao cabeçalho do arquivo *.c*, como já foi mostrado na Figura 16, e impactarão nos recursos de *hardware* utilizados pelo processador. Ao compilar o projeto, são gerados os arquivos *data.mif* e *inst.mif*, que serão sintetizados como conteúdo das memórias de dados e de programa, respectivamente. Além destes arquivos, é gerado um arquivo em *Verilog*, que é utilizado para instanciar os processadores do projeto no *hardware*, contendo toda a parametrização que foi feita na IDE do SAPHO.

4.2.1 Compilador C

O Compilador C foi desenvolvido utilizando as ferramentas GNU *flex* e *bison* [86], para detectar palavras-chaves e padrões de texto, respectivamente. Para uma sintaxe amigável, optou-se por utilizar a linguagem C^+ , um subconjunto da linguagem C.

A Tabela 4 exibe as funções e operadores lógico-aritméticos que o compilador reconhece. As funções destacadas em negrito são otimizações realizadas no decorrer deste trabalho e serão melhor detalhadas na Seção 5.1.

Tabela 4 – Palavras-chave e Operadores Lógico-Aritméticos da Linguagem C permitidos.

Palavras-chave	<i>in()</i>	<i>out()</i>	<i>void</i>	<i>int</i>	<i>float</i>	<i>return</i>	<i>while</i>	<i>if</i>	<i>else</i>	<i>abs()</i>	<i>sign()</i>									
Operadores	-	+	*	/	<	>	!	%	&		»	«	>=	<=	==	!=	&&		@	/>

O programa deve ser escrito em um arquivo único e aceita sub-rotinas. Caso as mesmas sejam detectadas, a pilha de instruções será instanciada no *hardware* para promover o retorno automático de funções, o que permite a implementação de algoritmos recursivos. O uso de ponteiros é permitido somente com indexação de *arrays* unidimensionais de tamanho fixo, de modo a ser possível o cálculo prévio do tamanho da memória de dados. A alocação dinâmica de memória não é permitida, uma vez que o objetivo deste processador é otimizar o uso de recursos em *hardware*.

4.2.2 Compilador Assembler

O compilador **Assembler** possui 47 instruções e é o responsável por gerar os arquivos em *Verilog* com a descrição do *hardware* do processador. O mesmo é desenvolvido com o auxílio do programa *flex* da GNU para o reconhecimento do *opcode* e dos operandos de cada instrução. Este compilador reconhece a quantidade final de instruções do programa e o número de variáveis necessárias, de modo a criar a memória de programa e de dados com o tamanho necessário.

4.3 FORMATO DE PONTO-FLUTUANTE

A representação de números em ponto-flutuante, em circuitos digitais, geralmente é feita utilizando-se como referência o padrão IEEE 754 [87], que utiliza 32 bits no formato *single* e 64 bits no formato *double*. Esse padrão define também algumas representações de números especiais e a forma como devem ser realizadas as operações entre números.

Os circuitos digitais que realizam operações aritméticas de acordo com o padrão IEEE 754 consomem muitos recursos de *hardware* e, geralmente, operam em circuitos sequenciais, ou seja, demoram alguns ciclos de *clock* para gerarem suas respostas. De forma a trabalhar com números em ponto-flutuante de uma maneira mais eficiente, do ponto de vista dos recursos de *hardware* e fluxo de dados, o SAPHO utiliza uma representação de ponto-flutuante simplificada, proposta em [88].

O formato de ponto-flutuante do SAPHO tem estrutura semelhante ao do padrão IEEE 754, em que o número é dividido em sinal (S), mantissa (M) e expoente (E). A mantissa é representada em módulo, como no padrão IEEE (somente valores positivos). Já no caso do expoente, é usada uma representação direta em Complemento a 2, para facilitar a descrição do *hardware*.

Uma das vantagens desta representação simplificada é que a mesma pode ser utilizada para qualquer combinação de números de mantissa e expoente. Considera-se sempre a mantissa normalizada entre 1 e 2. Desta forma, a sua parte inteira é sempre um bit igual a 1 (um) que neste caso, é necessária a sua representação.

Assim, a representação de ponto-flutuante do SAPHO é descrita como:

$$(-1)^S \times M \times 2^E \quad (4.1)$$

Um dos fatores que torna tal representação mais leve, em relação ao padrão IEEE, é a não utilização de recursos extras para representação de números especiais e tratamento de exceções. Porém, na maioria dos casos práticos, tais exceções não são atingidas quando o programa é cuidadosamente desenvolvido. Por isso, optou-se por priorizar a otimização de recursos em *hardware* em detrimento do tratamento de exceções.

5 IMPLEMENTAÇÃO

A plataforma de desenvolvimento utilizada nos testes e implementações do presente trabalho foi a FPGA EP4CE115 da família Cyclone IV E¹. Dentre os principais recursos, destacam-se os 3,9 Mbits de RAM (*Random Access Memory*), 266 blocos DSP e 114.480 elementos lógicos disponíveis. As ferramentas utilizadas para simulações operacionais do processador foram o Modelsim [89] e o Quartus².

A seguir serão detalhadas as modificações e otimizações realizadas na estrutura do SAPHO e o desenvolvimento dos processadores dedicados utilizando tal ferramenta.

Em sequência, será apresentada cada implementação realizada dos algoritmos para chegar na versão otimizada do método SSF. Será detalhada a arquitetura *multicore* proposta para um processamento que seja factível, respeitando a taxa de colisões do LHC. Por fim, será implementado também o algoritmo baseado em Redes Neurais Artificiais.

5.1 OTIMIZAÇÕES NA ESTRUTURA DO SAPHO

As modificações realizadas na estrutura do processador e seus compiladores, visando otimizar as implementações do presente trabalho [90, 91, 92], são apresentadas nessa seção. Novos blocos foram inseridos no *hardware* do SAPHO e, para acessar estes novos recursos, os compiladores foram atualizados, de forma a gerar as novas instruções em *Assembly*.

- Instrução **PSET**

O custo computacional necessário para o método baseado em Representação Esparsa de dados é alto, devido ao grande número de operações matriciais. A operação que foi descrita nas Figuras 16 e 17 é de grande importância na implementação deste método, uma vez que os valores obtidos na reconstrução de energia devem ser positivos [63] e, por isso, os termos menores que zero precisam ser anulados.

Nesse contexto, foi realizada uma modificação na estrutura do processador, através da inclusão de uma nova instrução em um novo bloco de processamento chamado **PSET**, de forma a permitir que tal operação seja feita diretamente em um novo circuito adicionado ao *hardware* em *Verilog*, em um único ciclo de *clock*.

Na Figura 19 é possível ver o código em C^+ e em *Assembly* antes da otimização. É feita uma comparação utilizando as funções *if* e *else*, indicado ao lado esquerdo da figura, e ao lado direito é possível ver as instruções em *Assembly* geradas para a primeira linha que em C^+ está destacada.

¹ Esta FPGA foi escolhida pois há *kits* disponíveis para testes em laboratório na UFJF.

² Disponível em: <https://fpgasoftware.intel.com/>

```

if (aux_0 < 0) x_0 = 0; else x_0 = aux_0; | LOAD 0
if (aux_1 < 0) x_1 = 0; else x_1 = aux_1; | LES aux_0
if (aux_2 < 0) x_2 = 0; else x_2 = aux_2; | JZ L3else
if (aux_3 < 0) x_3 = 0; else x_3 = aux_3; | LOAD 0
if (aux_4 < 0) x_4 = 0; else x_4 = aux_4; | SET x_0
if (aux_5 < 0) x_5 = 0; else x_5 = aux_5; | JMP L3end
if (aux_6 < 0) x_6 = 0; else x_6 = aux_6; | @L3else LOAD aux_0
if (aux_7 < 0) x_7 = 0; else x_7 = aux_7; | SET x_0
if (aux_8 < 0) x_8 = 0; else x_8 = aux_8; | @L3end
if (aux_9 < 0) x_9 = 0; else x_9 = aux_9; |
if (aux_10 < 0) x_10 = 0; else x_10 = aux_10; |

```

Figura 19 – Código em C^+ e respectivo *Assembly* antes de implementar o PSET.

Ainda na Figura 19, a primeira instrução **LOAD** está carregando o valor 0 no acumulador na saída da ULA, a partir da memória de dados. A instrução **LES** está checando se a variável *aux_0* é menor que o valor carregado anteriormente. A próxima instrução **JZ** indica que, caso o resultado da comparação anterior (**LES**) seja falso, é gerado um salto no código para o *else* do *loop* (**@L3else**), o qual é seguido pela instrução de carregar o valor de *aux_0* na variável *x_0*. E caso o resultado daquela comparação (**LES**) seja verdadeiro, o processador segue para a próxima instrução que é um **LOAD 0** seguido por um **SET x_0**, que carrega o valor 0 na variável *x_0*. Por fim, a instrução **JMP** está fazendo um salto para o fim do *loop* **@L3end**.

Na Figura 20 é possível observar a sintaxe necessária para utilizar a nova função, que foi definida pelo símbolo @, simplificando a sintaxe já conhecida das funções *if* e *else* comentadas anteriormente. É possível ver também que as instruções antes necessárias para realizar a operação agora se resumem apenas na nova instrução **PSET**, simplificando então o código em *Assembly* gerado pelo compilador **C**.

```

x_0@aux_0 ; | LOAD aux_0
x_1@aux_1 ; | PSET x_0
x_2@aux_2 ; | LOAD aux_1
x_3@aux_3 ; | PSET x_1
x_4@aux_4 ; | LOAD aux_2
x_5@aux_5 ; | PSET x_2
x_6@aux_6 ; | LOAD aux_3
x_7@aux_7 ; | PSET x_3
x_8@aux_8 ; |
x_9@aux_9 ; |

```

Figura 20 – Código em C^+ e respectivo *Assembly* com o PSET.

A instrução **PSET** analisa o valor ao lado direito do @ e, se ele for positivo, a variável da esquerda recebe este valor, caso contrário, ela recebe 0. Para o correto funcionamento desta nova instrução, foi introduzido um novo bloco de processamento na estrutura do processador que realiza esta comparação em um ciclo de *clock* e, para tal, o decodificador de instruções foi modificado para reconhecer a instrução e acionar o bloco correspondente, que está localizado na saída da ULA, como ilustrado no diagrama simplificado da estrutura do SAPHO na Figura 21.

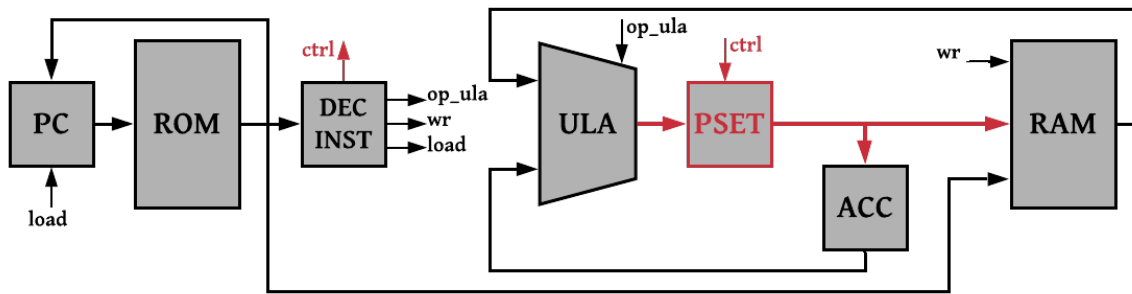


Figura 21 – Diagrama simplificado do SAPHO com o PSET.

A descrição do *hardware* em *Verilog* do bloco **PSET** em ponto-fixe e em ponto-flutuante pode ser vista nas Figuras 22 e 23, respectivamente.

```

generate
  if(PSET)
    always@(*)
    begin
      if (acc[NUBITS-1] == 1) pset_data <= 0;
      else pset_data <= acc;
    end
  else
    always@(*) pset_data = {NUBITS{1'bx}};
  endgenerate

```

Figura 22 – Código *Verilog* do PSET implementado em ponto-fixe.

```

always @(*) begin
  if ((acc[NBMANT+NBEXPO] == 1) && (ctrl == 1))
    x <= 0;
  else
    x <= acc;
end

```

Figura 23 – Código *Verilog* do PSET implementado em ponto-flutuante.

- Instrução **NORM**

A operação de normalização é de grande importância para implementações em ponto-fixe, sendo realizada através de uma divisão da variável que se deseja normalizar por um valor de ganho. Para um ganho na base 2, a operação se torna um *shift*.

Uma vez observado que tal operação se repete nos *loops* do SSF [90] e que, além disso, o valor do ganho é constante, a nova instrução **NORM** foi definida na ULA. Esta instrução normaliza o número à direita do operador, de forma que os circuitos de divisão necessários em *hardware* são drasticamente simplificados, uma vez que a ULA não tem mais a necessidade de verificar o valor do dividendo nas operações de divisão pelo ganho.

Para isso, foi incluído o novo parâmetro **NUGAIN**, onde é definido o valor constante de ganho (na base 2) a ser utilizado para as operações de normalização.

Na Figura 24 é possível ver a sintaxe em C^+ utilizada para a divisão por um ganho, que é realizada pela instrução **DIV**, em *Assembly*. É apresentada, na Figura 25, a sintaxe para usar a otimização do circuito de divisão por um ganho, definida pelo operador $/>$, que gera, em *Assembly*, a instrução **NORM**. Por fim, a descrição do *hardware* em *Verilog* do bloco **NORM** implementado na ULA pode ser vista na Figura 26.

```
x_0 = x_0/128; | LOAD 128
x_1 = x_1/128; | DIV mainx_0
x_2 = x_2/128; | SET mainx_0
```

Figura 24 – Código em C^+ e respectivo *Assembly* antes de implementar o **NORM**.

```
x_0 = /> x_0; | LOAD x_0
x_1 = /> x_1; | NORM
x_2 = /> x_2; | SET x_0
```

Figura 25 – Código em C^+ e respectivo *Assembly* com o **NORM**.

```
generate
  if (NRM == 1)
    assign nrm = in2 / NUGAIN;
  else
    assign nrm = {NUBITS{1'bx}};
endgenerate
```

Figura 26 – Código *Verilog* do **NORM** implementado na ULA.

- Instrução **ABS**

A obtenção do módulo ou valor absoluto de um número é de grande importância para o método de reconstrução de energia baseado em Redes Neurais [91, 92], uma vez que é necessário acessar as funções de ativação em LUT. Tal cálculo só podia ser implementado no SAPHO através de uma estrutura condicional usando as funções *if* e *else*, como pode ser visto na Figura 27. Tal abordagem não é muito eficaz por necessitar de vários ciclos de *clock*, devido a sua grande quantidade de instruções.

Visando otimizar esta operação, a ULA do processador foi modificada para suportar a nova instrução **ABS**, definida no compilador **Assembler**, que é capaz de realizar o cálculo do valor absoluto em *hardware* em apenas um ciclo de *clock*. Foi necessário também adicionar a função **abs()** na biblioteca padrão do Compilador **C**.

```

if(soma1 < 0){
    indice1 = -soma1;
}
else{
    indice1 = soma1;
}

```

```

LOAD 0
LES mainsoma1
JZ Llelse
LOAD mainsoma1
MLT -1
SET mainindice1
JMP Llend
@Llelse LOAD mainsoma1
SET mainindice1
@Llend

```

Figura 27 – Código em C_-^+ e respectivo *Assembly* antes de implementar o ABS.

```

indice1 = abs(soma1);

```

```

LOAD mainsoma1
ABS
SET mainindice1

```

Figura 28 – Código em C_-^+ e respectivo *Assembly* com o ABS.

O código em *Verilog* que foi implementado na ULA em ponto-fixado para gerar o circuito da instrução **ABS** está indicado na Figura 29.

```

generate
    if (ABS == 1)
        begin
            always @(*)
                if(in2[NUBITS-1])
                    abs = -in2 ;
                else
                    abs = in2;
        end
    else
        always@(*)
            abs = {NUBITS{1'bx}};
endgenerate

```

Figura 29 – Código *Verilog* do ABS implementado em ponto-fixado.

- Instrução **SIGN**

Para a proposta da implementação em ponto-flutuante da Rede Neural [92], a saída da rede é a combinação linear entre os valores provenientes das LUTs de 3 neurônios com o valor relativo à Série de Taylor do quarto neurônio. Entretanto, os valores provenientes destes neurônios precisam ser sinalizados com a soma ponderada de cada um com o valor respectivo da LUT, que é sempre positivo. Um exemplo deste tipo de operação é ilustrado no código C_-^+ da Figura 30, onde se pode observar que são necessárias 11 instruções em *Assembly* para que o resultado use o sinal de uma variável e o valor da outra.

```

float value = -10.0;
float signal = +3.8;
// queremos que a variável resultado tenha o valor da
// variável value e o sinal da variável signal
float resultado;

if((value*signal) < 0)
    resultado = -value;
else
    resultado = value;

```

```

LOAD mainsignal
MLT mainvalue
PLD 0
SLES
JZ L1else
LOAD mainvalue
NEG
SET mainresultado
JMP L1end
@L1else LOAD mainvalue
SET mainresultado
@L1end

```

Figura 30 – Código em C^+ e respectivo *Assembly* antes de implementar o SIGN.

Para otimizar esta operação, foi desenvolvida a função `sign()`, cujo novo *hardware* foi adicionado na ULA em ponto-flutuante e necessita de apenas 2 ciclos de *clock* para realizar a sinalização da variável `resultado`. Para tal, o Compilador **C** teve sua biblioteca padrão atualizada e o Compilador **Assembler** foi modificado para suportar a nova instrução **SIGN**. A função `sign()` precisa de dois argumentos: a variável que se deseja utilizar o sinal e a variável que se deseja utilizar o valor. Na Figura 31 está indicada a sintaxe da nova função implementada em C^+ e o código *Assembly* gerado.

```

resultado = sign(signal, value);

```

```

LOAD mainsignal
PLD mainvalue
SIGN
SET mainresultado

```

Figura 31 – Código em C^+ e respectivo *Assembly* com o SIGN.

Com esta otimização, o código em C^+ passa a ser resumido em apenas uma linha enquanto o respectivo *Assembly* gerado possui 4 instruções e não usa a instrução **MLT**, que é mais custosa para o processador e foi necessária na implementação sem a otimização.

Na Figura 32 está a descrição do *hardware* em *Verilog*, que foi atualizado na ULA em ponto-flutuante do processador para implementar as instruções **ABS** e **SIGN**.

- Inicialização Automática de *Array*

Por fim, o SAPHO foi também modificado para permitir a implementação direta de LUT armazenada na memória, uma funcionalidade importante para o método baseado em RNA [91, 92]. Para tal, os compiladores foram atualizados para receberem uma *string* contendo o nome do arquivo (em formato `.txt`), onde estão os dados do *array*. Os dados presentes neste arquivo são automaticamente colocados no topo do arquivo `data.mif` e, com isso, nenhuma instrução em *Assembly* é necessária para preencher o *array*.

Assim, como arquivo `data.mif` é usado para inicializar a memória de dados, o processador não precisa mais dispor de vários ciclos de *clock* para preencher o *array*.

```

generate
  if ((NEG == 1) && (ABS == 0) && (SIGN == 1)) begin
    assign sm = (op == 4'd5) ? !s2 : (op == 4'd13) ? s1 : s2;
    assign opm = (op == 4'd5 || op == 4'd13) ? 4'd0 : op;
  end

  else if ((NEG == 1) && (ABS == 1) && (SIGN == 1)) begin
    assign sm = (op == 4'd5) ? !s2 : (op == 4'd12) ? 1'b0 : (op == 4'd13) ? s1 : s2;
    assign opm = (op == 4'd5 || op == 4'd12 || op == 4'd13) ? 4'd0 : op;
  end

  else if ((NEG == 0) && (ABS == 1) && (SIGN == 1)) begin
    assign sm = (op == 4'd12) ? 1'b0 : (op == 4'd13) ? s1 : s2;
    assign opm = (op == 4'd12 || op == 4'd13) ? 4'd0 : op;
  end

  else if ((NEG == 1) && (ABS == 1) && (SIGN == 0)) begin
    assign sm = (op == 4'd5) ? !s2 : (op == 4'd12) ? 1'b0 : s2;
    assign opm = (op == 4'd5 || op == 4'd12) ? 4'd0 : op;
  end

  else if ((NEG == 1) && (ABS == 0) && (SIGN == 0)) begin
    assign sm = (op == 4'd5) ? !s2 : s2;
    assign opm = (op == 4'd5) ? 4'd0 : op;
  end

  else if ((NEG == 0) && (ABS == 1) && (SIGN == 0)) begin
    assign sm = (op == 4'd12) ? 1'b0 : s2;
    assign opm = (op == 4'd12) ? 4'd0 : op;
  end

  else if ((NEG == 0) && (ABS == 0) && (SIGN == 1)) begin
    assign sm = (op == 4'd13) ? s1 : s2;
    assign opm = (op == 4'd13) ? 4'd0 : op;
  end

  else begin
    assign sm = s2;
    assign opm = op;
  end
endgenerate

```

Figura 32 – Código *Verilog* do ABS e SIGN implementados em ponto-flutuante.

A sintaxe para a declaração de *arrays* no SAPHO antes e depois da otimização é destacada, respectivamente, nas Figuras 33 e 34.

```

int meuArray[5];
meuArray[0] = 2;
meuArray[1] = 3;
meuArray[2] = 4;
meuArray[3] = 6;
meuArray[4] = 1;
LOAD 0
PLD 2
SRF
SET mainmeuArray
LOAD 1
PLD 3
SRF
SET mainmeuArray

```

Figura 33 – Inicialização de *array* no SAPHO antes da otimização.

```

int meuArray[5] "tabela.txt";

```

Figura 34 – Inicialização de *array* no SAPHO com a otimização.

5.2 REPRESENTAÇÃO ESPARSA EM PONTO-FLUTUANTE

Dado o embasamento matemático do método SSF, descrito na Seção 3, a implementação do algoritmo para se obter o sinal representando a estimação de energia pode ser resumida na resolução da Equação 5.1, realizando-se 150 iterações (estudo feito em [63]).

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mu \mathbf{H}^T (\mathbf{r} - \mathbf{H} \mathbf{x}_i) \quad (5.1)$$

Vale ressaltar que não é possível escrever diretamente os cálculos matriciais em linguagem \mathbf{C}^+ no SAPHO, portanto, as operações matriciais precisam ser escritas termo a termo, demandando uma grande quantidade de linhas de código. Visando simplificar esse trabalho manual, foram desenvolvidas rotinas na linguagem *Python* [93], criadas especialmente para escrever as operações necessárias para realizar os produtos matriciais desse método em linguagem \mathbf{C}^+ no SAPHO. Uma outra característica importante a ser ressaltada é que a matriz de convolução \mathbf{H} possui diversos coeficientes nulos, como pode ser visto na Figura 35, o que facilita a implementação dos produtos matriciais em \mathbf{C}^+ .

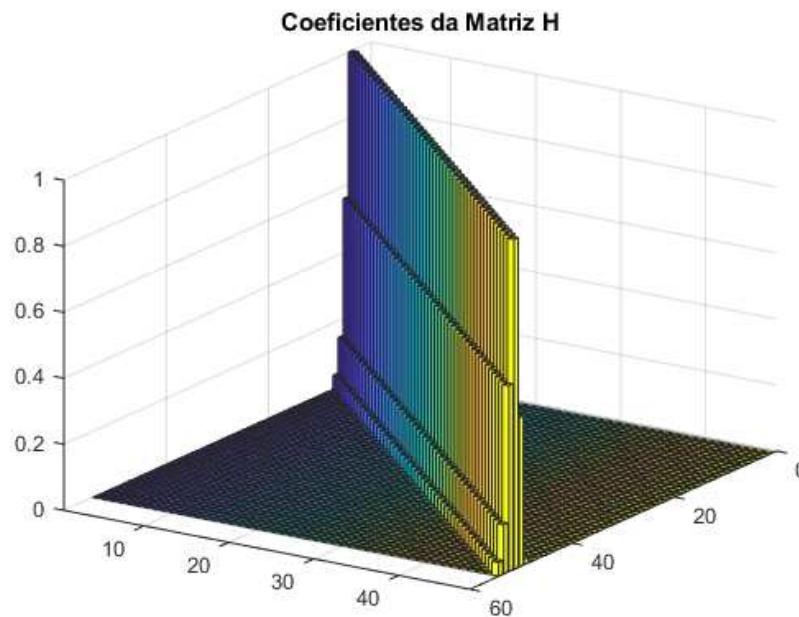


Figura 35 – Coeficientes da matriz de convolução.

Nas implementações que serão apresentadas a seguir, os parâmetros conhecidos são a matriz de convolução \mathbf{H} , a constante μ , que foi fixada por simulações em 0.25 [63], e também os dados de entrada \mathbf{r} com características de empilhamento. Além disso, é possível também simular o sinal esperado após a reconstrução de energia, para fins de comparação e verificação do correto funcionamento do algoritmo, uma vez que o *Toy Monte Carlo* de [55] foi utilizado para gerar os sinais dos testes.

A primeira implementação (versão 1.0) em ponto-flutuante do método SSF foi realizada da forma mais genérica possível, onde cada termo da matriz \mathbf{H} na Equação 5.1 é declarado como uma variável. O número total de *clocks* necessários para a realização das 150 iterações deste algoritmo foi 14.057.029 e código implementado em \mathbf{C}^{\pm} pode ser visto de forma resumida no Algoritmo 1, Anexo A.

Como os dados de entrada $r[n]$ chegam a cada 25 ns e a janela de entrada para esse método possui 55 amostras consecutivas do conversor ADC, todo o processamento para uma janela precisa ocorrer num período de $55 \times (25 \text{ ns}) = 1375 \text{ ns}$, que é o tempo necessário para que a próxima janela de entrada de dados seja formada. Assim, todo o processamento ocorrerá de forma síncrona com a frequência do LHC, como é esperado. Desta forma, para que a versão 1.0 do processador implementado respeite a taxa de colisões do LHC, a frequência necessária ao processador é $14.057.029 / (1.375 \text{ ns}) = 10,22 \text{ THz}$.

Tal frequência não é factível para implementação em eletrônica digital [94], o que motivou na busca por implementações mais eficientes para tal processamento [95, 96].

Para a versão 1.1 da implementação do algoritmo, foi realizada a operação distributiva na Equação 5.1 de forma a reduzir os produtos matriciais pela matriz \mathbf{H}^T , que na implementação anterior estava em evidência no *loop* de 150 iterações. Dessa forma, ao se realizar a manipulação algébrica, a nova equação é:

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mu \mathbf{H}^T \mathbf{r} - \mu \mathbf{H}^T \mathbf{H} \mathbf{x}_i \quad (5.2)$$

Assim, a parcela $\mu \mathbf{H}^T \mathbf{r}$ só precisa ser calculada uma vez, reduzindo o número de operações dentro do *loop*, ao comparar com a primeira versão da implementação. A frequência de operação para esta implementação é 10,05 THz, ou seja, diminuiu em 170 GHz ao ser comparada com a implementação anterior. O Algoritmo 2, no Anexo A, resume o código implementado em \mathbf{C}^{\pm} para essa versão da implementação.

Visando reduzir ainda mais o número de instruções, foram realizadas mais manipulações algébricas na equação do método SSF. Como o produto $\mathbf{H}^T \mathbf{H}$ é constante, este resultado passa a ser definido como a matriz simétrica \mathbf{A} (ilustrada na Figura 36).

O produto $\mathbf{H}^T \mathbf{r}$ não é constante, uma vez que \mathbf{r} sempre está se atualizando com os dados da respectiva janela de entrada, mas essa operação só precisa ser realizada uma vez, antes da primeira iteração. Portanto, tal resultado passa a ser definido como matriz \mathbf{Hs} . Além disso, os termos nulos de \mathbf{H} e \mathbf{A} passam agora a ser descartados. A equação pode então ser descrita como:

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mu (\mathbf{Hs} - \mathbf{Ax}_i) \quad (5.3)$$

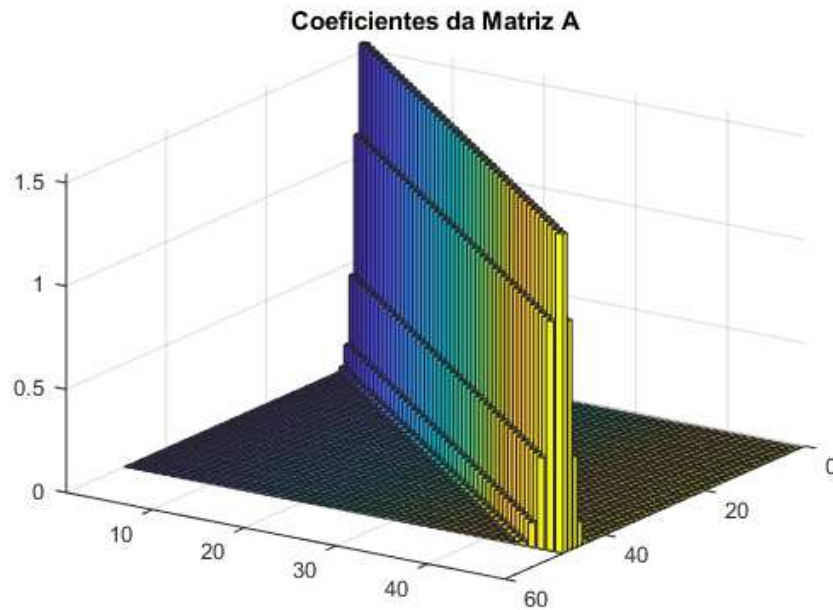


Figura 36 – Coeficientes da matriz de autocorrelação.

A terceira implementação do método (versão 2.0) foi realizada com os termos não nulos de \mathbf{H} e \mathbf{A} sendo declarados de forma literal como variáveis. A quarta implementação (versão 2.1) foi realizada com esses valores escritos diretamente no código, como constantes, visando diminuir ainda mais o número de instruções da implementação. O Algoritmo 3, no Anexo A, resume como as operações da Equação 5.3 foram realizadas em \mathbf{C}_+^+ . A frequência operacional necessária para esta implementação é 764 GHz, ou seja, caiu para menos de 8% do total da frequência operacional que era necessária na implementação anterior.

Para a versão 2.1, também foi utilizado o Algoritmo 3, porém os termos não nulos de \mathbf{H} e \mathbf{A} não são mais declarados de forma literal (variáveis), passando a ser escritos diretamente no código como constantes. Após realizada a análise do número total de instruções, incluindo as que se repetem no *loop*, dividido pelo tamanho da janela de entrada de dados, foi possível constatar que a frequência de operação necessária para essa implementação é de 447 GHz.

Para a implementação da versão 2.2, descrita no Algoritmo 4 do Anexo A, a Equação 5.3 foi realizada de forma ainda mais direta. O sinal de subtração da expressão foi colocado em evidência e não há mais vetores. Nesse algoritmo, o sinal \mathbf{x} não é mais representado como um vetor, mas sim declarado termo a termo como 48 variáveis. O sinal de entrada \mathbf{r} (que no algoritmo é representado por \mathbf{d}) também é declarado termo a termo, como 55 variáveis. Dessa forma, evita-se o acesso indireto a memória pelo índice dos *arrays* em \mathbf{C} , que precisa de mais ciclos de *clock* para executar.

Para esta implementação, a frequência necessária de operação é 258 GHz. É possível notar que a frequência foi drasticamente reduzida, ao comparar com os 10,22 THz obtidos na primeira implementação do método SSF, porém tal valor ainda não é factível para implementação com as tecnologias atuais. Um dos principais motivos para esses valores de frequência de operação é o grande número de iterações necessárias ao algoritmo.

5.2.1 Inicialização Otimizada do Algoritmo

Visando otimizar a implementação do método, é introduzida a matriz pseudo-inversa de \mathbf{H} (matriz \mathbf{B} na Figura 37), que permite a inicialização do vetor iterativo \mathbf{x} com valores mais próximos da solução do problema [69], como visto na Seção 3.1.

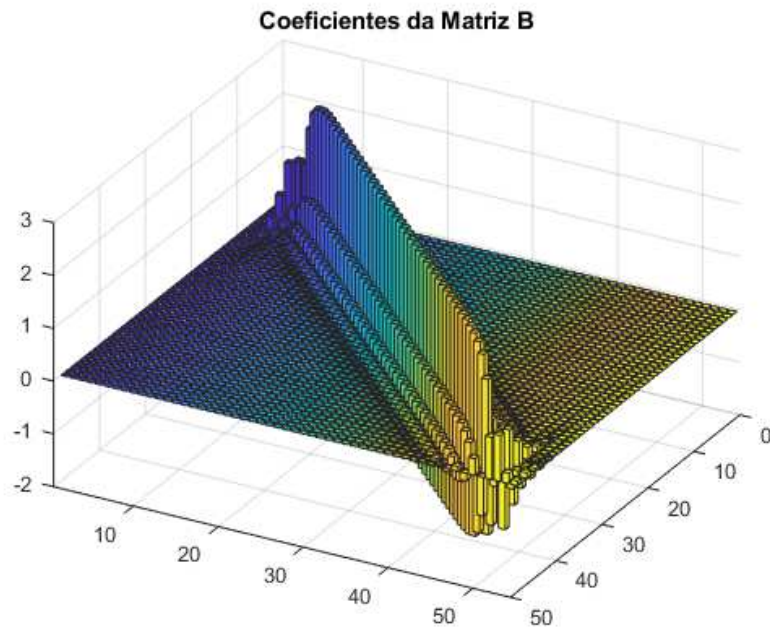


Figura 37 – Coeficientes da matriz pseudo-inversa.

Após realizadas as simulações para verificar o erro RMS pelo número de iterações, é possível analisar a convergência do método SSF nos gráficos da Figura 38. Este gráfico mostra que o método SSF padrão converge para o resultado (com erro RMS abaixo de 2,5) para cerca de 150 iterações. Já o método SSF com o pré-processamento na inicialização de \mathbf{x} , por meio da matriz pseudo-inversa, converge para o resultado com cerca de 18 iterações. Ainda nestes gráficos, é possível observar que quanto maior a ocupância (medida de empilhamento), mais iterações são necessárias e maior o erro final.

Além disso, estudos realizados em [69] mostram que é possível zerar coeficientes próximos de zero das matrizes \mathbf{H} , \mathbf{A} e \mathbf{B} , respeitando um patamar de corte, reduzindo assim o número de operações a serem realizadas pelo processador.

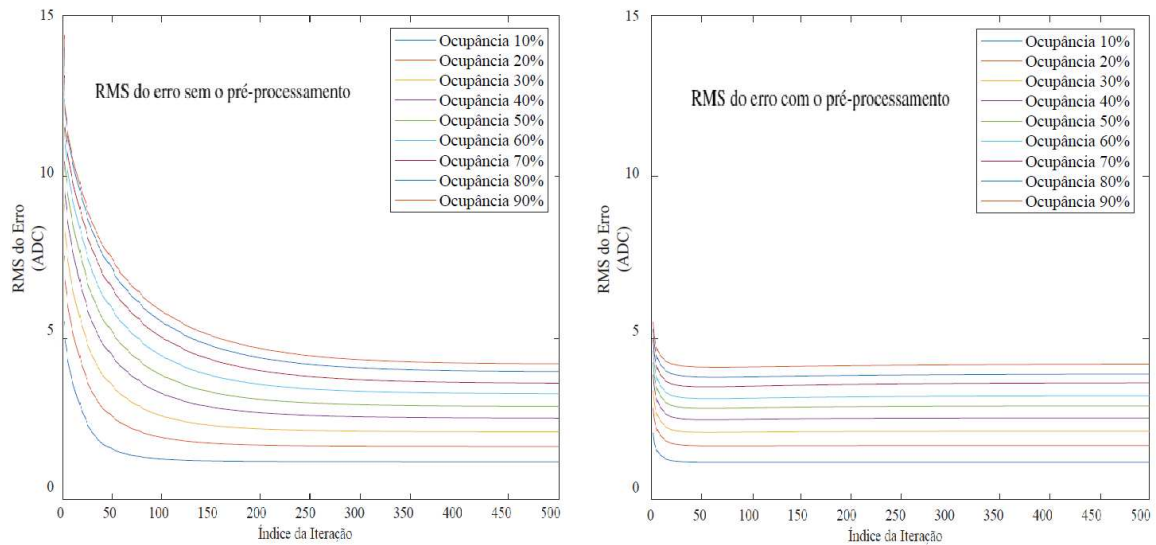


Figura 38 – Erro RMS da estimação de energia pelo método SSF [69].

No trabalho [69] foi mostrado que as curvas da convergência para o método SSF com coeficientes anulados nas matrizes e da convergência do mesmo usando as matrizes originais praticamente se sobrepõem.

A implementação do método SSF com a matriz pseudo-inversa está descrita no Algoritmo 5 (Anexo A), onde os termos próximos de zero das matrizes \mathbf{H} , \mathbf{A} e \mathbf{B} foram descartados. A versão 3.0 desta implementação foi realizada com o processo de inicialização de \mathbf{x} antes de realizar o *loop* de 18 iterações. A versão 3.1 possui o mesmo algoritmo da versão 3.0, porém foi acrescentada a nova função @ (**PSET**, substituindo as estruturas *if* e *else*), que foi aprimorada no SAPHO especialmente para reduzir o número de instruções nas partes do algoritmo em que os termos negativos são anulados. Assim, a diferença entre essas implementações está no número de instruções.

Para a implementação da versão 3.0, a frequência necessária de operação é 23 GHz, o que é menos de 10% da frequência na melhor implementação sem o uso da matriz pseudo-inversa (versão 2.2). Para a implementação da versão 3.1, a frequência de operação caiu para 21 GHz, graças ao uso da nova instrução **PSET**.

5.3 REPRESENTAÇÃO ESPARSA EM PONTO-FIXO

Após ter sido realizada toda análise e simplificação do método em ponto-flutuante, foi desenvolvido um processador em ponto-fixo para operar o algoritmo, visando diminuir ainda mais o custo lógico da implementação. Para tal, foi necessário realizar uma quantização dos elementos das matrizes de convolução e pseudo-inversa, de forma que os valores menores que 1 nas operações não fossem zerados quando truncados e também para que uma precisão mínima fosse mantida.

No gráfico da Figura 39 está a relação entre os valores para o ganho e os respectivos valores RMS do erro, onde é possível notar que um ganho de 2^5 já é suficiente para que o erro de quantização seja negligenciado.

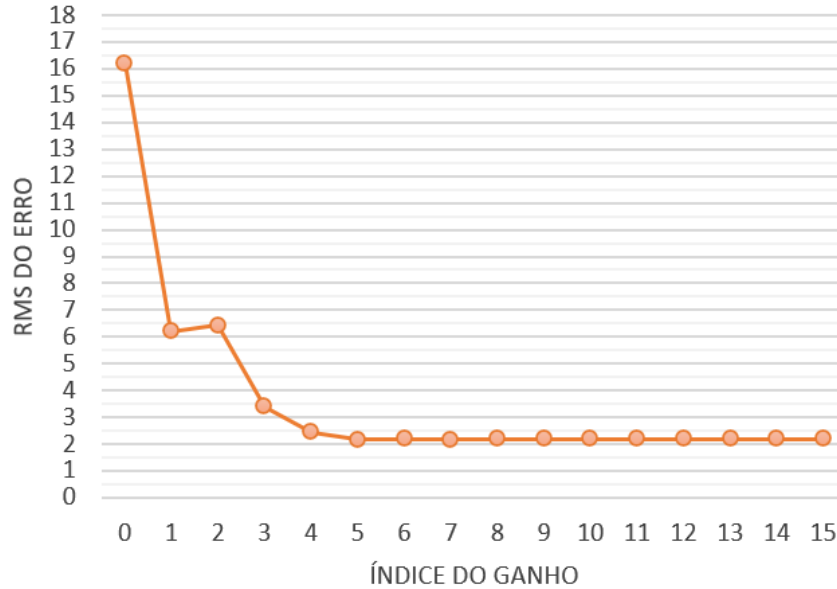


Figura 39 – Relação entre o valor RMS do erro e o ganho.

Com o intuito de garantir uma margem de segurança, o fator de quantização escolhido foi 2^7 (ou seja, um ganho de 128), garantindo que os coeficientes das matrizes **H**, **A** e **B** não sejam anulados nas operações. Dessa forma, os números são representados com precisão numérica de 2^{-7} .

Além disso, foram realizados testes variando o número de bits do processador, de forma que não houvesse comprometimento no desempenho, como problemas de *overflow* devido as operações de multiplicação e soma. O número de bits foi definido como 32.

Essa implementação seguiu a linha de raciocínio da melhor versão do processador operando em ponto flutuante com a função **PSET** (Algoritmo 5, Anexo A), porém todas as variáveis e constantes utilizadas devem possuir valores inteiros. Para isso, foi necessário acrescentar produtos e divisões pelo ganho escolhido ao desenvolver o código em C^+ . A frequência de operação necessária é de 23 GHz e esse aumento já era esperado, devido ao aumento no numero de operações para a aplicação do ganho necessário ao algoritmo.

O valor definido para o ganho é constante, logo, esse termo é repetido em todas as operações de divisão. Visando reduzir o custo lógico dessas operações, foi criada no SAPHO a instrução **NORM**, já apresentada em 5.1. Essa instrução é definida na ULA do processador e tem a função de normalizar o número desejado pelo valor constante do ganho (neste caso, definido como 2^7). Com isso, os circuitos de divisão são simplificados pois a ULA não precisa mais verificar ambas as entradas nessa operação.

Como a diferença entre as duas implementações do método em ponto-fixado está apenas na substituição do circuito de divisão pela operação de normalização, a frequência do processamento se manteve em cerca de 23 GHz.

O que foi alterado de forma significativa ao comparar essas duas implementações foi o número de elementos lógicos, que foi reduzido de 1.621 para 382.

Abaixo estão descritas, de forma resumida, as principais características dos algoritmos de cada versão que foi implementada do método SSF. Nos resultados (Seção 6) será mostrado como cada uma das modificações e aprimoramentos no algoritmo foi importante para chegar em uma implementação factível.

1. Implementação em Ponto-Flutuante - Versão 1.0:

- Realizada de forma genérica.
- Todos os termos de \mathbf{H} declarados como variáveis.
- O sinal \mathbf{x} é declarado como um vetor.
- Todas operações são realizadas dentro de um *loop* de 150 iterações.

2. Implementação em Ponto-Flutuante - Versão 1.1:

- Realizada a operação distributiva na equação.
- Todos os termos de \mathbf{H} declarados como variáveis.
- O sinal \mathbf{x} é declarado como um vetor.
- O produto $\mu\mathbf{H}^T$ é calculado só uma vez e está fora do *loop*.

3. Implementação em Ponto-Flutuante - Versão 2.0:

- Os termos nulos de \mathbf{H} e de \mathbf{A} passam a ser descartados.
- Os termos não nulos de \mathbf{H} e de \mathbf{A} são declarados como variáveis.
- O sinal \mathbf{x} é declarado como um vetor.
- A matriz $\mathbf{H}\mathbf{s} = \mathbf{H}^T\mathbf{r}$ é calculada só uma vez antes de o *loop* começar.

4. Implementação em Ponto-Flutuante - Versão 2.1:

- Os termos nulos de \mathbf{H} e de \mathbf{A} são descartados.
- Os termos não nulos de \mathbf{H} e de \mathbf{A} passam a ser escritos diretamente no código, como constantes.
- O sinal \mathbf{x} é declarado como um vetor.
- A matriz $\mathbf{H}\mathbf{s} = \mathbf{H}^T\mathbf{r}$ é calculada uma vez antes de o *loop* começar.

5. Implementação em Ponto-Flutuante - Versão 2.2:

- O sinal \mathbf{x} passa a ser representado por 48 variáveis.
- Todos os termos constantes são escritos diretamente no código.
- Sinal de subtração colocado em evidência (constantes positivas).
- A matriz $\mathbf{Hs} = \mathbf{H}^T \mathbf{r}$ é calculada uma vez antes de o *loop* começar.

6. Implementação em Ponto-Flutuante - Versão 3.0:

- Implementação da inicialização otimizada de \mathbf{x} .
- O *loop* passa a ter 18 iterações.
- Todos os termos constantes são escritos diretamente no código.
- Aplicação do patamar de corte para anular termos próximos de zero.
- A matriz $\mathbf{Hs} = \mathbf{H}^T \mathbf{r}$ e a inicialização de \mathbf{x} são calculados antes do *loop*.

7. Implementação em Ponto-Flutuante - Versão 3.1:

- Implementação da inicialização otimizada de \mathbf{x} .
- Constantes escritas diretamente no código e patamar de corte aplicado.
- A matriz $\mathbf{Hs} = \mathbf{H}^T \mathbf{r}$ e a inicialização de \mathbf{x} são calculados antes do *loop*.
- Uso da nova instrução **PSET**.

8. Implementação em Ponto-Fixo - Versão 1:

- Implementação da inicialização otimizada de \mathbf{x} .
- Constantes escritas diretamente no código e patamar de corte aplicado.
- A matriz $\mathbf{Hs} = \mathbf{H}^T \mathbf{r}$ e a inicialização de \mathbf{x} são calculados antes do *loop*.
- Uso da nova instrução **PSET**.
- Foi aplicado um ganho de 2^7 .

9. Implementação em Ponto-Fixo - Versão 2:

- Implementação da inicialização otimizada de \mathbf{x} .
- Constantes escritas diretamente no código e patamar de corte aplicado.
- A matriz $\mathbf{Hs} = \mathbf{H}^T \mathbf{r}$ e a inicialização de \mathbf{x} são calculados antes do *loop*.
- Uso das novas instruções **PSET** e **NORM**.
- Foi aplicado um ganho de 2^7 .

5.4 ESTRUTURA MULTICORE

Como a melhor versão da implementação do método SSF em ponto-fixado possui uma frequência de processamento que ainda não é factível para que o processador possa operar de forma individual (23 GHz), foi projetada uma estrutura com múltiplas instâncias [90], onde cada processador pode operar com frequências de *clock* factíveis, visando respeitar a taxa de eventos do sistema de aquisição de dados do ATLAS.

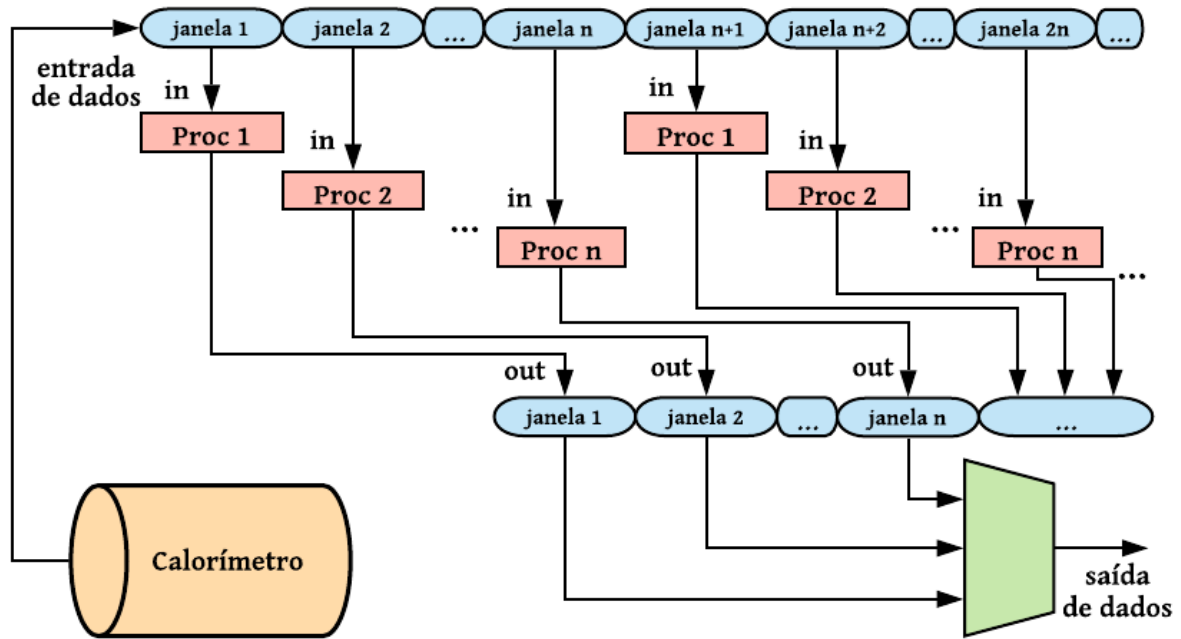


Figura 40 – Diagrama da estrutura *multicore* implementada.

Na Figura 40 está ilustrado um diagrama da estrutura *multicore* para uma quantidade n de processadores, em paralelo, com funcionamento em *pipeline* [97], para um canal do calorímetro, sendo que 10.000 desses esquemas devem ser implementados independentemente. Nesta estrutura, cada janela de dados é direcionada e executada por um processador, mantendo assim um fluxo contínuo e sequencial de processamento. Após cada processador receber sua janela de dados de entrada, as operações dos mesmos são realizadas de forma independente entre si. Na saída, um multiplexador recebe as janelas contendo o sinal reconstruído por cada processador, respectivamente, no instante correto.

A janela de dados de saída do primeiro processador e a janela de dados de entrada do último processador da estrutura ocorrem de forma simultânea, assim, o primeiro processador pode então carregar a sua janela de dados de entrada novamente o processo se repete sem que ocorra perda de dados. A quantidade de processadores depende da frequência de operações dos mesmos e foram testadas diversas configurações.

Com isso, foi possível sincronizar a estrutura *multicore* aqui apresentada com a taxa de eventos do LHC, de forma que os dados das janelas de entrada e de saída ocorram a cada 25 ns. Além disso, essa estrutura foi implementada para diferentes quantidades de processadores, em paralelo, operando em diferentes frequências (mais detalhes na Seção 6).

Para tal, foi necessário analisar a quantidade de *clocks* entre cada dado na respectiva janela de entrada. Na Figura 41 é possível observar, ao lado esquerdo, a janela de entrada de dados sendo carregada com a função `in()`, em \mathbf{C}^+ , que pega o dado da porta de entrada desejada e, ao lado direito, está destacado o respectivo código em *Assembly* gerado, que contém 4 instruções para cada entrada.

18	<code>int d_0 = in(0);</code>	13	<code>LOAD 0</code>
19	<code>int d_1 = in(0);</code>	14	<code>PUSH</code>
20	<code>int d_2 = in(0);</code>	15	<code>IN</code>
21	<code>int d_3 = in(0);</code>	16	<code>SET maind_0</code>
22	<code>int d_4 = in(0);</code>	17	<code>LOAD 0</code>
23	<code>int d_5 = in(0);</code>	18	<code>PUSH</code>
24	<code>int d_6 = in(0);</code>	19	<code>IN</code>
25	<code>int d_7 = in(0);</code>	20	<code>SET maind_1</code>
26	<code>int d_8 = in(0);</code>	21	<code>LOAD 0</code>
27	<code>int d_9 = in(0);</code>	22	<code>PUSH</code>
28	<code>int d_10 = in(0);</code>	23	<code>IN</code>
29	<code>int d_11 = in(0);</code>	24	<code>SET maind_2</code>
30	<code>int d_12 = in(0);</code>	25	<code>LOAD 0</code>

Figura 41 – Código para o preenchimento da janela de entrada de dados.

Foi possível observar, após simular no Modelsim, que o espaçamento entre cada pulso de entrada de dados é de 4 ciclos de *clock*. Portanto, para que os sinais de *enable* dos dados de entrada ocorram sincronizados com a taxa de eventos de 40 MHz do LHC (a cada 25 ns), foi contabilizado que é necessário que o processador opere a uma frequência de $4 \times 2 \times (40 \text{ MHz}) = 320 \text{ MHz}$.

Além disso, sabe-se que o tamanho de uma janela de dados de entrada é $55 \times (25 \text{ ns}) = 1.375 \text{ ns}$ e, no Modelsim, foi observado que o processador levou 186.781 ns para realizar as operações, após ter preenchido sua janela de entrada, para então poder começar a preencher a janela de dados de saída.

Dessa forma, seriam necessários que mais de 136 processadores ($\frac{186.781}{1.375} \text{ ns} = 136, 12$) completassem suas janelas de dados de entrada, sequencialmente, para que o primeiro processador tivesse tempo de terminar seu processamento e pudesse, enfim, carregar a sua nova janela de entrada. Foi então necessário adicionar instruções neutras³ no código em *Assembly*, visando adicionar ciclos de *clock*, de forma que o tempo de processamento fosse suficiente para que 137 processadores carregassem suas janelas de entrada. Estas instruções neutras foram adicionadas ao final do código, antes de a janela de saída de dados começar a ser carregada.

³ A instrução neutra escolhida em *Assembly* foi a `MLT 1`, a qual não realiza nenhuma alteração nas operações, apenas faz um produto pelo elemento neutro da multiplicação 1.

Assim, foram adicionados ciclos de *clock* por meio de instruções neutras de forma que o tempo de processamento pudesse ocorrer no período de $137 \times (1.375 \text{ ns}) = 188.375 \text{ ns}$ e, com isso, para uma frequência de operação de 320 MHz, foram necessários 138 processadores, no total, operando em paralelo, como ilustrado no diagrama da Figura 40.

Tal procedimento foi possível devido a uma modificação na IDE e estrutura do SAPHO, que passou a permitir que o Compilador **C** e o **Assembler** fossem executados de forma independente.

Vale ressaltar que, a partir desta modificação no SAPHO, foi possível realizar a programação, edição e compilação do código *Assembly* de forma direta e independente do código **C⁺** na IDE, uma vez que, antes disso, o código *Assembly* era gerado automaticamente, de acordo com o respectivo código **C⁺** e não podia ser modificado.

Seguindo a mesma lógica, este procedimento foi realizado para outros valores de frequência, uma vez que, ao adicionar ciclos de *clock* entre os pulsos de entrada, é necessário aumentar a frequência operacional de forma a manter o período de 25 ns entre estes pulsos. Por exemplo, ao adicionar 1 ciclo de *clock*, a frequência operacional necessária passa a ser $5 \times 2 \times (40 \text{ MHz}) = 400 \text{ MHz}$ e, além disso, as operações passam a ser realizadas em um período menor, necessitando de menos processadores em paralelo, uma vez que a janela de dados de entrada se mantém em $55 \times (25 \text{ ns}) = 1.375 \text{ ns}$. Tais detalhes serão apresentados e comparados na Seção 6.

5.5 REDES NEURAIAS ARTIFICIAIS USANDO *LOOK-UP TABLE*

O diagrama em ponto-fixa da RNA proposta a ser implementada no SAPHO está ilustrado na Figura 42. Como há variação da escala no circuito, foi utilizada a notação Qi.f, onde *i* é o número de bits da parte inteira e *f* da parte fracionária [98]. Tal notação permite maior detalhe para representar a aritmética de ponto-fixa em questão, que leva em conta o número de bits e considera a posição do ponto na sequência binária. O complemento de 2 foi utilizado para representar os números decimais negativos.

A estratégia de quantização da função de ativação dos neurônios 1, 2 e 3 da camada escondida pode ser observada no gráfico da esquerda na Figura 43, onde foi admitido um erro máximo de 1% em comparação com a RNA simulada em ponto-flutuante [91].

No gráfico da direita está ilustrado o processo de quantização dos pesos da camada escondida. Esta mesma análise foi feita para as demais partes da rede e, com isso, foi possível otimizar a arquitetura do processador no SAPHO, onde foi definida a sequência binária com configuração de 31 bits.

O uso de LUT é comum em funções de ativação de redes multicamadas [99], sendo endereçada de modo que o resultado da soma ponderada das entradas dos neurônios representa um endereço que acessará o valor correspondente a saída da função de ativação.

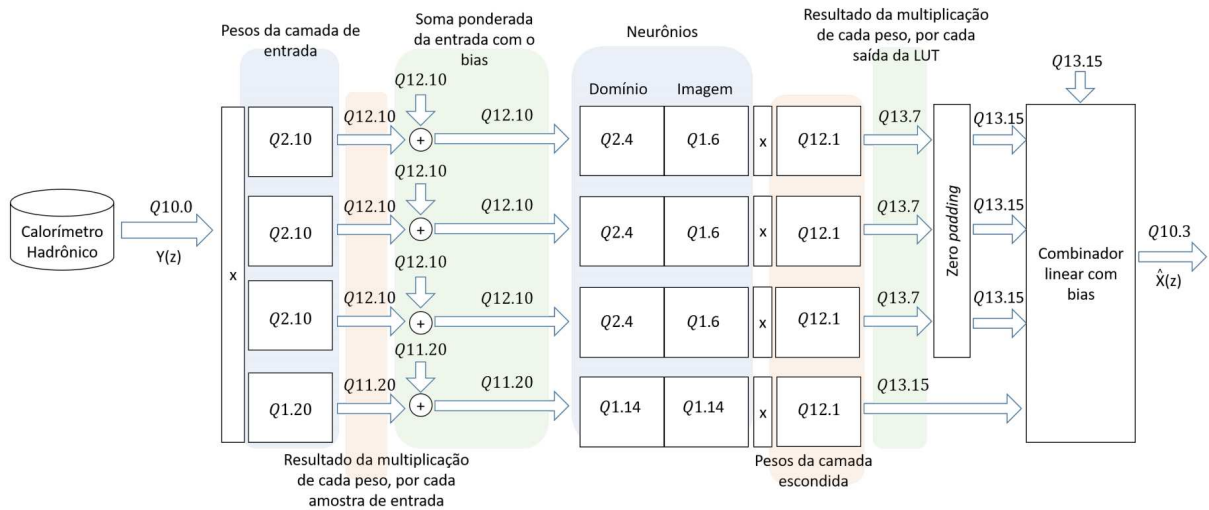


Figura 42 – Diagrama em ponto-fixa da implementação da RNA [65].

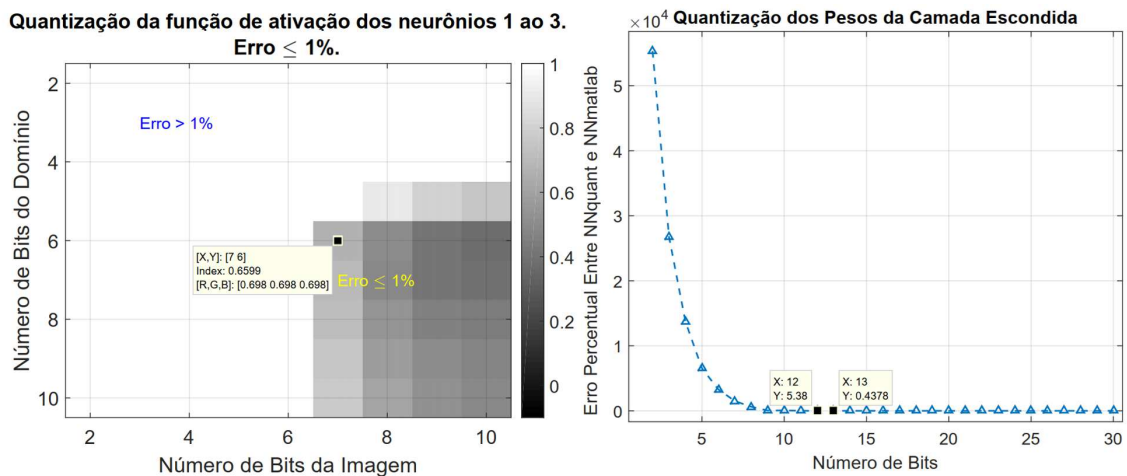


Figura 43 – Análises de quantização [65].

Dentre os quatro neurônios, três deles demandaram pesos menos significativos na camada de saída e utilizaram a mesma LUT. O quarto neurônio precisou de pesos de maior precisão que os demais na camada de saída e sua quantização demandou maior memória para a representação de sua LUT.

A Função Sigmoide foi quantizada e limitada apenas em sua porção positiva, uma vez que é uma função ímpar. Quando a sinapse é negativa, a saída da LUT é multiplicada por -1. Se a entrada possui módulo maior que 1048576 para o neurônio de maior precisão ou módulo maior que 3072 para os demais neurônios, a saída será a última posição da respectiva LUT. O diagrama da Figura 42 foi implementado sequencialmente em *software* no SAPHO, na linguagem C^+ . O algoritmo implementado pode ser visto na Figura 44.

O processador desenvolvido inicia lendo os 10 componentes do vetor de entrada, que estão armazenados externamente em registradores de deslocamento.

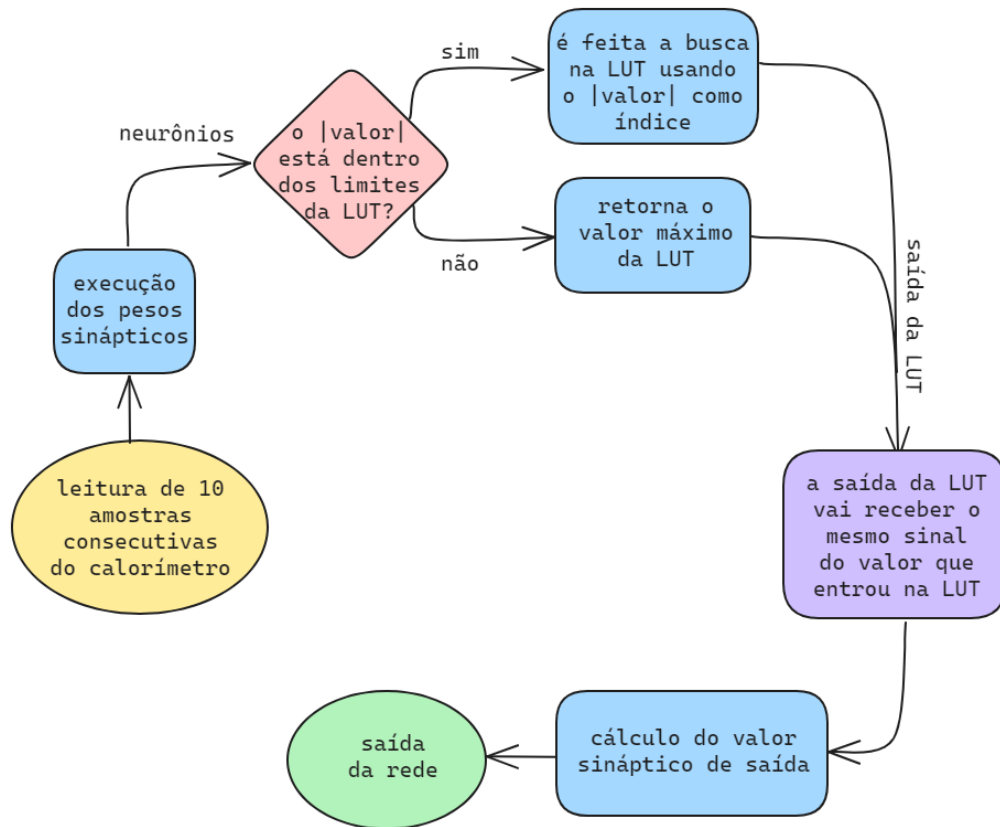


Figura 44 – Fluxograma da RNA em ponto-fixa.

Os coeficientes das sinapses, representados em ponto-fixa, são constantes em *software* e para as LUTs são utilizados *arrays* de constantes previamente calculadas.

Após a execução dos pesos sinápticos, o resultado é dividido pelo ganho de 2^6 , já prevendo a necessidade de se adequar ao domínio da LUT. Para realizar esta divisão de forma otimizada, diminuindo o consumo de recursos lógicos da FPGA, foi utilizada a nova instrução **NORM** com o parâmetro **NUGAIN=6**, definindo o índice do ganho.

O tempo necessário para o processamento de uma janela de dados impede que um único processador possa ser utilizado. Para contornar isso, foi desenvolvida uma arquitetura multicore de processadores em um *loop* infinito, onde uma nova janela começa a ser calculada no momento que a atual é finalizada.

O diagrama com a arquitetura multicore desta estrutura paralela de operação é o mesmo que foi detalhado anteriormente para o método SSF, na Figura 40.

5.5.1 Aproximação da Função Sigmoide usando Séries de Taylor

Devido ao alto custo computacional e consumo de memória da implementação da LUT do neurônio que demandou maior precisão, foi realizado o estudo da substituição desta LUT por uma Série de Taylor [100], visando aproximar a Função Sigmoide.

Para tal, foi desenvolvido um método iterativo, variando a ordem da série entre 3 e 6 e o centro entre 0 e 1,4. Estes limites foram definidos seguindo a aproximação da Função Sigmoide utilizada no trabalho [92]. Além disso, como a função em questão é ímpar, foi necessário fazer a aproximação apenas para o primeiro quadrante, pois os demais pontos são contemplados pela utilização dos respectivos valores absolutos.

Os resultados desta análise são apresentados na Figura 45 e a combinação escolhida que atende ao critério desejado foi ordem 4 e centro 0,1.

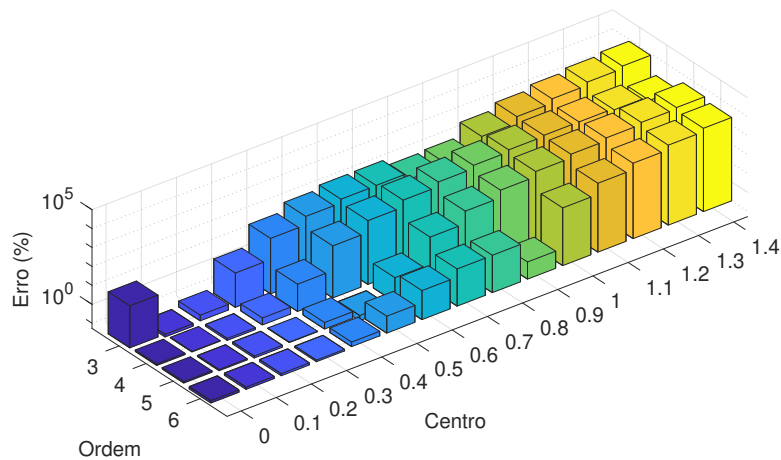


Figura 45 – Erro obtido variando a ordem e o centro da Série de Taylor.

Dessa forma, o polinômio que descreve essa aproximação é dado por:

$$P(x) = 0,0997 + 0,9901(x - 0,1) - 0,0987(x - 0,1)^2 - 0,3202(x - 0,1)^3 \quad (5.4)$$

Para a implementação em ponto-flutuante deste polinômio, visando minimizar o custo lógico, foi realizada uma análise de combinações entre número de bits de mantissa e expoente, de forma que esta soma não excedesse o limite de 31 bits imposto pelo SAPHO.

Tal análise pode ser vista na Figura 46, onde o erro produzido por cada combinação é levado em consideração e, admitindo um erro máximo de 1 %, o par 7 e 15 foi definido como número de bits de expoente e mantissa, respectivamente.

O processador desenvolvido para operação da RNA em ponto-fixa foi modificado para operar em ponto-flutuante e a LUT do neurônio que demanda maior precisão foi substituído pelo Polinômio $P(x)$ em 5.4. O algoritmo implementado no SAPHO para o processamento dedicado em ponto-flutuante está indicado na Figura 47.

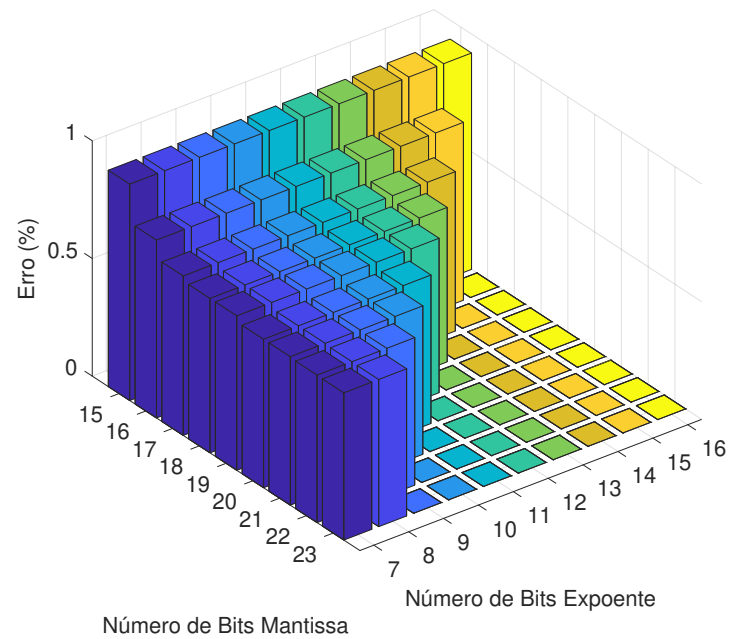


Figura 46 – Erro obtido variando número de bits do expoente e da mantissa.

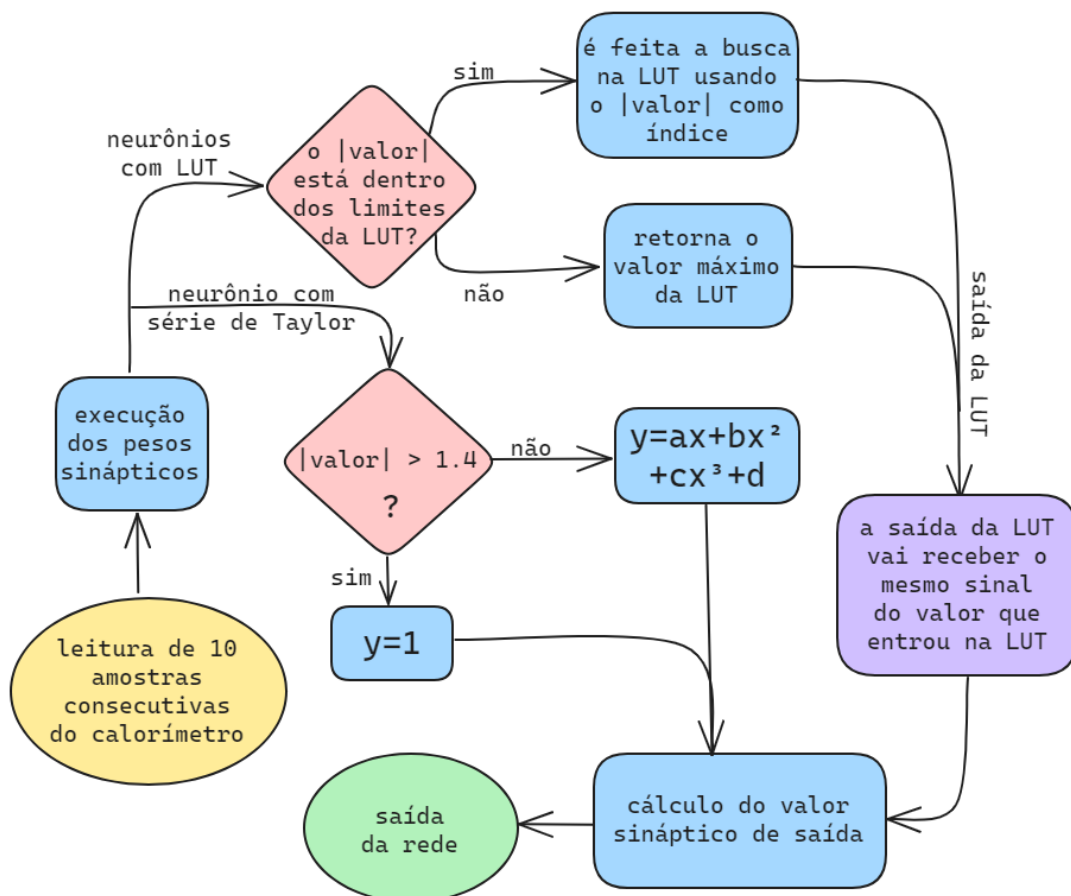


Figura 47 – Fluxograma da RNA em ponto-flutuante.

Após a execução dos pesos sinápticos neste diagrama, é verificado se os resultados dos três neurônios que utilizam LUT estão dentro dos limites. Se sim, a busca na LUT é realizada, se não, é retornado o valor máximo da LUT. Tal busca é feita utilizando o valor absoluto, então é necessário observar se o sinal numérico precisa ser corrigido e, para isso, foi utilizada a nova função **sign()** do SAPHO.

É verificado para o neurônio com a Série de Taylor se, em termos absolutos, o resultado é maior ou igual a 1,4. Se sim, é retornado 1, se não, o polinômio é calculado.

Por fim, para o cálculo do valor sináptico de saída, os valores provenientes de cada neurônio são combinados linearmente junto com o *bias*.

Assim como nas demais implementações, foi também necessário o desenvolvimento de uma estrutura *multicore*, onde para cada frequência operacional configurada, é utilizada uma quantidade k de processadores operando em paralelo, com funcionamento em *pipeline*. As comparações e resultados serão apresentados na seguinte seção.

6 RESULTADOS

Nesta seção serão descritas as simulações operacionais de cada versão das implementações do método SSF e das Redes Neurais Artificiais em FPGA e também das estruturas *multicore* desenvolvidas para diferentes quantidades de processadores operando em paralelo para diferentes frequências.

Para o método SSF, a relação entre as características da implementação de cada versão dos processadores, operando de forma individual, pode ser vista na Tabela 5. Essa tabela mostra que a otimização da inicialização do algoritmo com o uso da matriz pseudo-inversa, a aplicação do patamar de corte para os termos próximos de zero nas matrizes e o aprimoramento na estrutura do SAPHO para as novas funções (**PSET** e **NORM**) ajudaram a reduzir a frequência e custo lógico de forma significativa.

Tabela 5 – Comparação entre as implementações do método SSF.

Versão	Frequência (GHz)	Instruções	Variáveis	Elementos Lógicos	Bits de Memória
ponto-flut. v1.0	10.224	33.674	5.510	1.405	936.864
ponto-flut. v1.1	10.050	34.203	5.557	1.407	955.518
ponto-flut. v2.0	764	5.152	282	1.270	102.294
ponto-flut. v2.1	447	3.729	218	1.254	84.612
ponto-flut. v2.2	258	3.622	185	1.253	88.581
ponto-flut. v3.0	23	4.934	700	782	108.446
ponto-flut. v3.1	21	4.370	700	782	108.446
ponto-fixo v1	23	4.473	407	1.621	98.011
ponto-fixo v2	23	4.621	407	382	100.803

É possível notar também que a melhor implementação em ponto-fixo do método SSF, apesar de ter seguido a mesma linha de raciocínio da melhor implementação em ponto-flutuante, ficou com uma frequência de operação um pouco maior (devido ao aumento no número de operações para aplicação do ganho), porém o custo lógico foi reduzido.

Após realizada a análise das características dos processadores operando de forma individual, foi analisada também a estrutura *multicore*, onde o método SSF é comparado com a reconstrução de energia por meio de Redes Neurais Artificiais.

Para a frequência de entrada de dados de 40MHz e sendo N o número de ciclos de *clock* entre os pulsos de entrada de dados, o cálculo da frequência operacional da estrutura *multicore* é dado por $f_{\text{MHz}} = 2 * 40 * N$ e as análises aqui apresentadas foram realizadas para a frequência variando no intervalo entre 320 MHz e 2 GHz.

No gráfico das Figuras 48 e 49 é possível analisar como os parâmetros de número de processadores e tempo de atraso Δt (tempo necessário para que os dados do sinal reconstruído possam ser descarregados na saída em um fluxo contínuo e sequencial) variam de acordo com a frequência do *clock* de operação dos processadores para cada método.

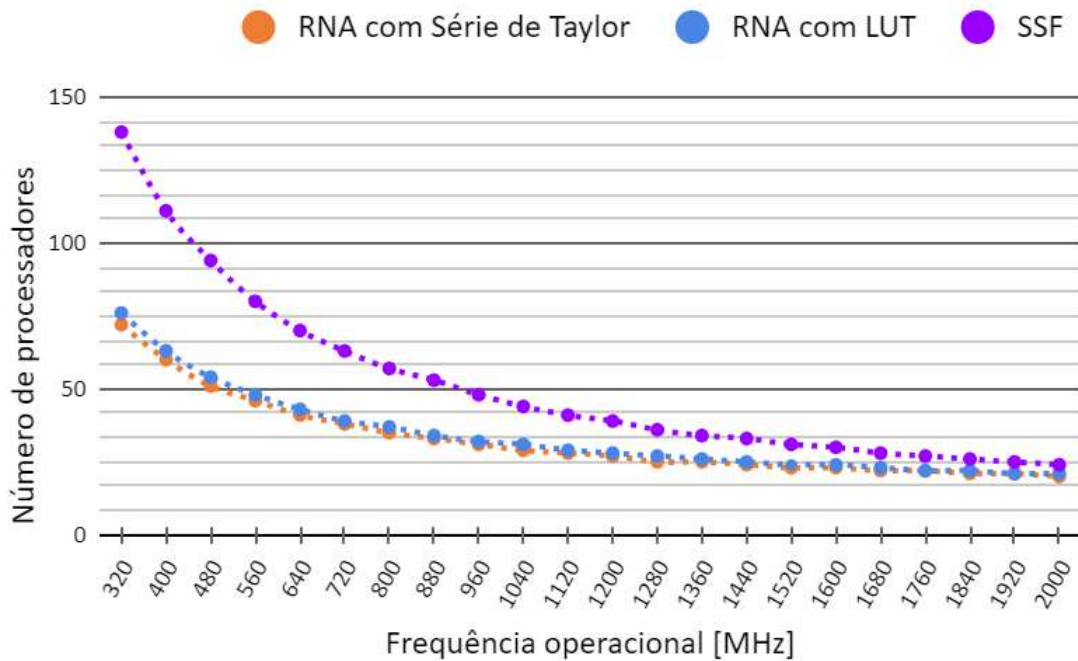


Figura 48 – Quantidade de processadores variando com a frequência.

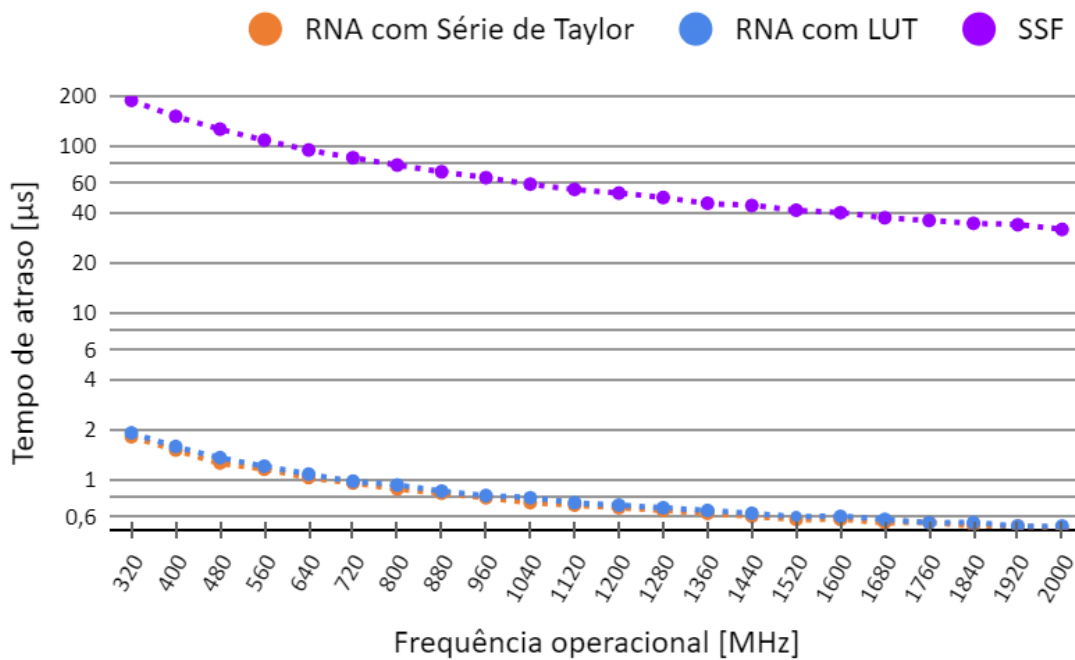


Figura 49 – Tempo de atraso variando com a frequência.

Na Figura 50 é possível observar como o consumo de recursos de *hardware* diminui com o aumento da frequência de operação. Vale ressaltar que o Quartus foi configurado para equilibrar a alocação dos recursos de forma automática.

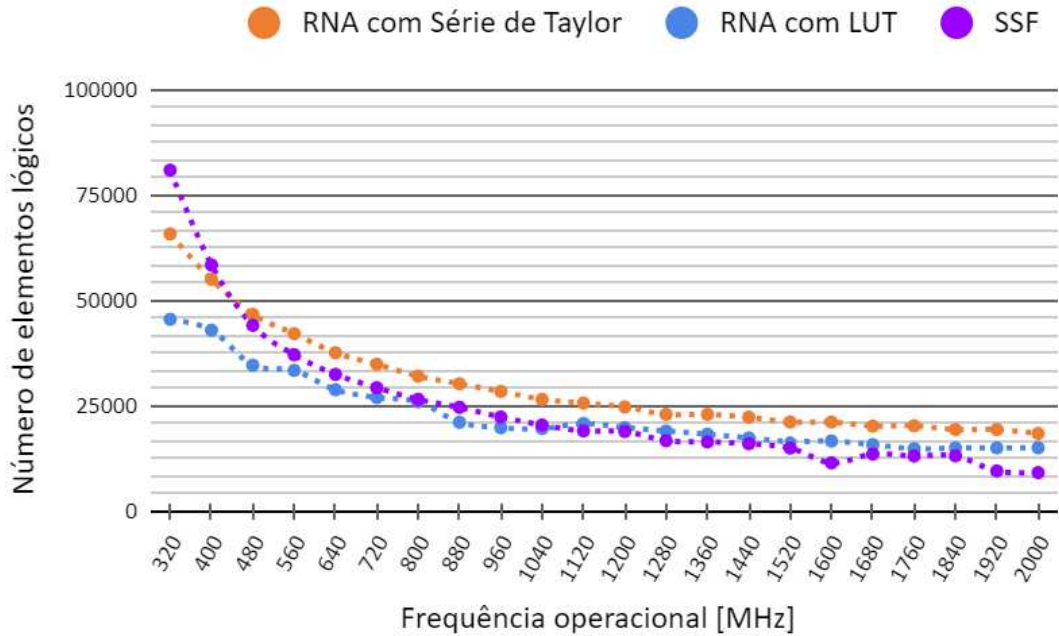


Figura 50 – Elementos lógicos variando com a frequência.

Foi realizada também uma análise dos bits de memória necessários para a implementação das estruturas *multicore* de acordo com a frequência de operação (Figura 51).

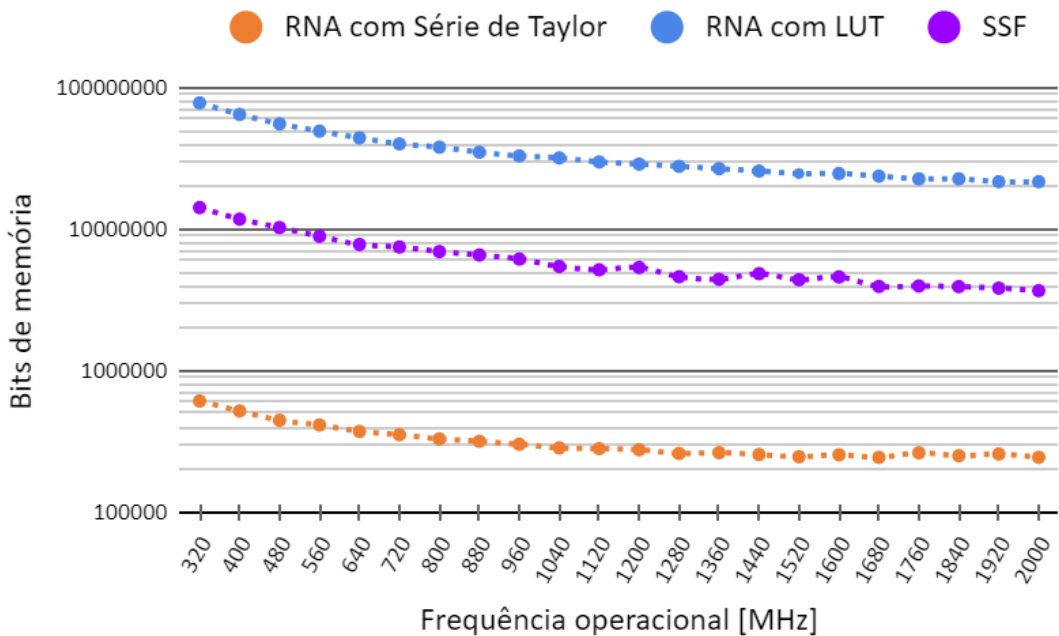


Figura 51 – Bits de memória variando com a frequência.

Mesmo para a estrutura *multicore* que possui o maior requisito de memória (cerca de 80 Mbits), a implementação é factível para operação em FPGAs modernas, que podem possuir mais de 200 Mbits de memória disponíveis [94].

Na Figura 52 é possível observar, na interface do Modelsim, o sinal a ser reconstruído por um processador (com frequência operacional de 320 MHz), cujos dados vêm dos canais de leitura do TileCal, onde cada pulso representa o instante em que um dado chega e, como pode-se notar, os pulsos de entrada ocorrem a cada 25 ns, de forma que está sincronizado com a taxa de eventos do LHC.

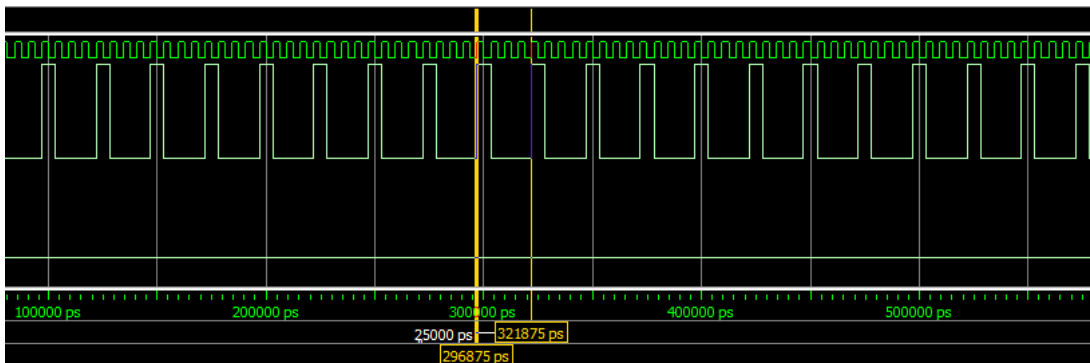


Figura 52 – Janela de entrada dos dados do método SSF no Modelsim.

Por fim, para ilustrar a estimaco de energia proposta pelo presente trabalho, no grfico da Figura 53 so comparados trs sinais: (i) o sinal com *pile-up*, representando os dados vindos dos canais de leitura do TileCal a uma taxa de 40 MHz; (ii) o sinal alvo, que representa o sinal do calormetro sem nenhum efeito de empilhamento, ou seja, seria o sinal ideal que se deseja obter; (iii) o sinal reconstrudo por um processador operando o mtodo SSF com a inicializaco otimizada do algoritmo e um *loop* de 18 iteraoes. Destaca-se o RMS abaixo de 2,5 que foi explicitado na Figura 39 para a escolha do ganho referente ao processamento em ponto-fixado.

Vale ressaltar que este trabalho no  focado nos estudos de eficincia dos mtodos apresentados, mas sim nos estudos e validao da implementaco de processadores customizados em FPGA para a operao de tais mtodos.

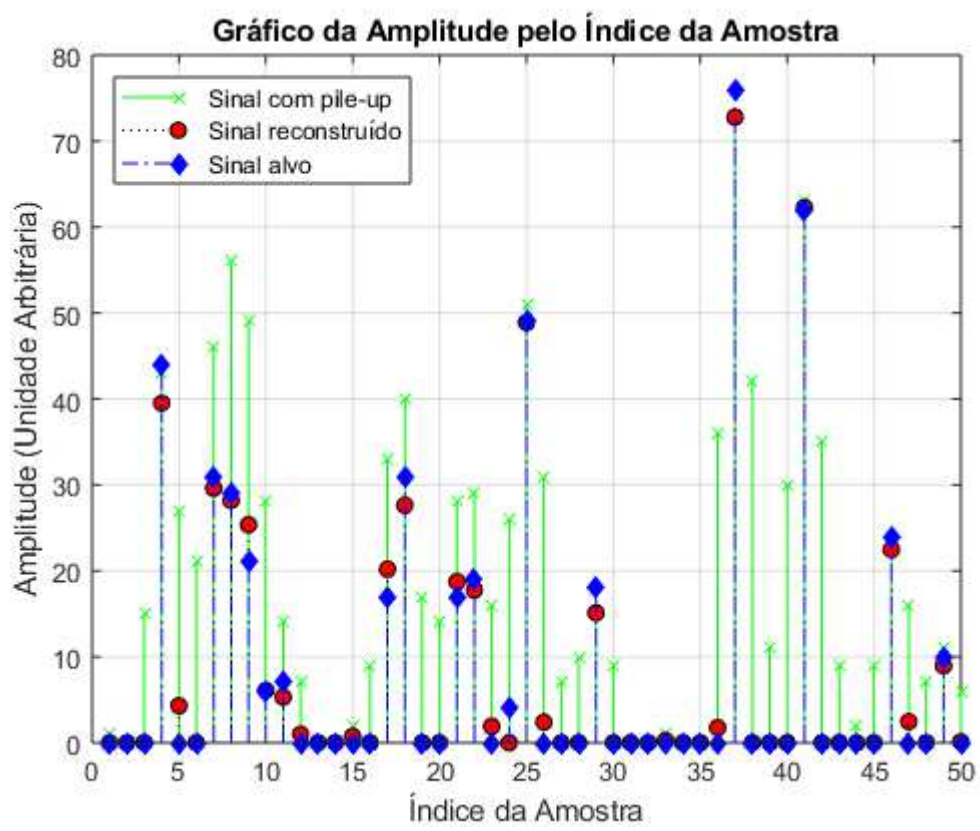


Figura 53 – Comparação entre o sinal reconstruído, o alvo e o sinal ruidoso.

7 ANÁLISES DE DESEMPENHO DO SAPHO

O desempenho do SAPHO foi comparado com o do NIOS II, que foi escolhido por ser configurável e por abranger características encontradas nos diversos PSCs disponíveis atualmente. Além disso, outra vantagem é que o NIOS II conta com uma IDE *user-friendly* e aceita o mesmo código em **C** escrito para o SAPHO.

Para realizar o teste de desempenho, foram utilizados três modelos do processador NIOS II [101]. Dentre estes modelos, o mais simples é o **NIOS II/e**, que possui barramentos de 32 bits, apenas 1 estágio de *pipeline* e frequência máxima de operação de 200 MHz, tendo como foco a otimização de recursos. O modelo intermediário é o **NIOS II/s**, que tem barramentos de 32 bits, 5 estágios de *pipeline* e frequência máxima de operação de 165 MHz, além de possuir *hardware* de multiplicação e de divisão. Por fim, foi testado também o modelo **NIOS II/f**, que possui barramentos de 32 bits, 6 estágios de *pipeline*, frequência máxima de operação de 185 MHz e *hardware* dedicado para multiplicação e divisão, tendo como foco a otimização de performance. As implementações analisadas foram realizadas usando o Quartus e a FPGA EP4CE115 da família Cyclone IV E.

Para cada um destes três modelos do NIOS II, foi possível optar entre utilizar ou não um adicional de *hardware* para ponto-flutuante, totalizando 6 configurações possíveis de processadores a serem analisados e comparados com o SAPHO. Os processadores gerados foram submetidos a testes de desempenho, utilizando o simulador Modelsim.

Primeiramente, foram feitos programas em **C** para realizar simples operações aritméticas de soma, produto e de divisão, combinadas com as instruções de leitura e de escrita em memória. Com este teste, é possível verificar a quantidade mínima de recursos e o número necessário de *clocks* para realização das operações básicas, tanto em ponto-fixa, quanto em ponto-flutuante.

O segundo teste de desempenho foi a implementação de um algoritmo de Transformada de Fourier, o FFT [102], de 8 pontos, cuja parte principal do código pode ser vista na Figura 54. No caso do SAPHO, a implementação da FFT foi realizada somente com a ULA em ponto-flutuante. Os resultados dos testes são apresentados nas Tabelas 6, 7 e 8.

Diferentemente dos processadores convencionais, o SAPHO, visando otimização de recursos, não realiza aritmética de números inteiros quando a ULA instanciada é em ponto-flutuante e vice-versa. Do ponto de vista prático, todas as variáveis, mesmo as usadas para indexação ou contadores, podem ser computadas em ponto-flutuante quando este tipo de ULA é utilizada. Desta forma, os testes realizados para o SAPHO apresentam somente resultados para o tipo de dados referente ao respectivo tipo de ULA. No caso da ULA em ponto-fixa, o SAPHO foi configurado para 32 bits, para proporcionar uma comparação justa com o NIOS II.

Tabela 6 – Custo lógico e frequência (*hardware* para ponto-flutuante indicado por *)

Implementação em FPGA		Elementos	Bits de	DSP	Freq.	
(Custo Lógico)		Lógicos	Memória	(9 bits)	(MHz)	
NIOS II/e		1.440	141.312	0	136,28	
NIOS II/e*		7.969	142.393	7	205,34	
NIOS II/s		2.225	175.616	4	173,25	
NIOS II/s*		8.828	176.697	11	182,88	
NIOS II/f		3.024	193.216	4	132,52	
NIOS II/f*		9.615	194.297	11	158,08	
SAPHO	Soma	int	186	288	0	120,66
		float	617	171	0	54,08
	Produto	int	191	288	2	77,47
		float	447	171	2	71,62
	Divisão	int	1.424	288	0	8,55
		float	912	171	0	15,19
	FFT		884	9.025	2	49,61

```

109     m = 1;
110     while (m < mmax)
111     {
112         i = m;
113         while(n >= i)
114         {
115             j = i + mmax;
116
117             tempr      = wr*data[j]  - wi*data[j+1];
118             tempi      = wr*data[j+1] + wi*data[j];
119
120             data[j]   = data[i]  - tempr;
121             data[j+1] = data[i+1] - tempi;
122             data[i]   = data[i]  + tempr;
123             data[i+1] = data[i+1] + tempi;
124
125             i = i + istep;
126         }
127
128         wtemp = wr;
129         wr = wtemp*wpr - wi*wpi + wr;
130         wi = wi*wpr + wtemp*wpi + wi;
131
132         m = m+2;
133     }

```

Figura 54 – IDE do SAPHO com a parte principal do código da FFT.

Tabela 7 – Quantidade de *clocks* (*hardware* para ponto-flutuante indicado por *)

Simulação (<i>clocks</i>)	Soma		Produto		Divisão		FFT
	int	float	int	float	int	float	
NIOS II/e	117	2.703	263	8.246	401	3.822	813.538
NIOS II/e*	117	1.454	263	1.456	489	1.481	161.041
NIOS II/s	41	911	45	1.040	183	1.214	118.378
NIOS II/s*	41	529	45	537	186	554	40.605
NIOS II/f	28	771	28	872	161	1.008	80.654
NIOS II/f*	34	465	40	467	162	490	27.884
SAPHO	13	13	13	13	13	13	1.981

Tabela 8 – Tempo de execução (*hardware* para ponto-flutuante indicado por *)

Tempo (μs)	Soma		Produto		Divisão		FFT
	int	float	int	float	int	float	
NIOS II/e	0,85	19,83	1,93	60,51	2,94	28,05	5969,61
NIOS II/e*	0,57	7,08	1,28	7,09	2,38	7,21	784,26
NIOS II/s	0,23	5,26	0,26	6,00	1,06	7,01	683,28
NIOS II/s*	0,22	2,89	0,25	2,94	1,02	3,03	222,03
NIOS II/f	0,21	5,37	0,21	6,59	1,21	7,61	608,62
NIOS II/f*	0,22	2,94	0,25	2,95	1,02	3,10	176,39
SAPHO	0,11	0,24	0,17	0,18	1,52	0,86	39,93

Como pode ser visto na Tabela 8, dentre as configurações do NIOS II, o de maior desempenho (**NIOS II/f** em ponto-flutuante) executa o código mais rápido que as demais configurações, totalizando 176,39 μs . O SAPHO executa o mesmo código em 39,93 μs .

Ao gerar um processador NIOS II, o número de Elementos Lógicos (EL), bits de memória e multiplicadores (DSPs) será fixado, independentemente do programa a ser executado. No SAPHO, tais parâmetros são auto-escaláveis (Tabela 6).

No caso das operações aritméticas, o maior número de elementos lógicos para o SAPHO ocorre quando o mesmo detecta a operação de divisão. Para os testes do SAPHO em ponto-flutuante a ULA foi configurada com 19 bits.

O **NIOS II/e**, que é o mais simples das instâncias testadas, utiliza 1.440 elementos lógicos para divisão, um número maior do que o SAPHO configurado para executar a FFT (884 elementos). Além disso, esta mesma versão do NIOS II necessita de 813.583 *clocks* para esta mesma tarefa, que é executada com apenas 1.981 *clocks* pelo SAPHO.

Mesmo a versão mais eficiente do NIOS II necessita de uma quantidade muito maior de *clocks* (27.884) do que o SAPHO (1.981) para executar a FFT.

A quantidade de bits de memória e o número de blocos de DSP também é expressivamente maior no NIOS II. Estes números são auto-escaláveis no SAPHO, evitando o desperdício destes dois recursos tão importantes na maioria das aplicações.

Quanto à frequência de operação, devido ao fato de a ULA do SAPHO ser combinacional, a desvantagem é que o mesmo opera, em média, três vezes mais lento que o NIOS II. No entanto, a ULA combinacional possui a vantagem de permitir uma arquitetura em três estágios sem quebra de *pipeline*, o que faz com que uma sequência de operações seja realizada em menos ciclos de *clock*.

8 CONCLUSÕES

Nesse trabalho foram apresentadas diversas topologias para implementações que visam baixo consumo de recursos lógicos em FPGA, tanto no formato de ponto-fixa quanto no de ponto-flutuante, de processadores capazes de operar um algoritmo baseado em Redes Neurais Artificiais e um método iterativo de deconvolução baseado em Representação Esparsa de dados, visando realizar a estimação de energia em Calorímetros de Altas Energias, com foco no Calorímetro Hadrônico do ATLAS.

A abordagem por RNA realiza a estimação de energia levando em conta as não linearidades do problema, enquanto o SSF é uma técnica linear que consiste em operações matriciais de soma e multiplicação, onde grande parte dos termos das matrizes utilizadas são nulos, reduzindo o número de operações necessárias para implementar o algoritmo.

Um destaque na implementação do SSF foi a otimização na inicialização do vetor iterativo com o uso do pré-processamento com a matriz pseudo-inversa. Apesar da necessidade de um cálculo matricial adicional, foi possível diminuir o número de iterações de 150 para 18, diminuindo assim o número de ciclos de *clock* em uma ordem de grandeza. Em relação ao tempo de atraso, ao comparar o método baseado em RNA com o SSF, a diferença foi de duas ordens de grandeza. Outro destaque foi que, na implementação usando RNA, a substituição da LUT pela série de Taylor reduziu o consumo de memória em duas ordens de grandeza.

Com as modificações na estrutura do SAPHO, foi possível implementar os algoritmos propostos, para processamentos realizados tanto em ponto-flutuante quanto em ponto-fixa, com o objetivo de comparar e definir qual a melhor implementação em relação a frequência operacional, tempo de atraso e consumo lógico. As análises feitas para o SAPHO mostram que ele possui ótimo desempenho, sendo comparado com o NIOS II, um PSC comercial robusto e já consolidado no mercado. Ressalta-se que o SAPHO tem a vantagem de possuir código aberto, o que facilita a sua personalização e adição de novas funcionalidades.

Assim, é possível concluir que o presente trabalho contribuiu com diversas questões como: (i) a estimação *online* de energia no primeiro nível de *trigger* do TileCal; (ii) publicamos 7 artigos no decorrer deste trabalho; (iii) a implementação em FPGA de forma dedicada dos algoritmos que foram propostos; (iv) uma otimização na inicialização do método SSF; (v) um aprimoramento da RNA com o uso de Séries de Taylor para aproximação da função Sigmóide para o neurônio que demanda maior precisão; (vi) otimizações na estrutura do SAPHO; (vii) o desenvolvimento de uma estrutura *multicore* de processamento capaz de operar na frequência de 40 MHz do LHC; (viii) comparações entre as diversas implementações propostas em FPGA do processador que foi customizado e aprimorado; (ix) análises de desempenho do SAPHO.

REFERÊNCIAS

- [1] PERKINS, D. H. *Introduction to High Energy Physics*. Cambridge University Press, April 2000.
- [2] WILLE, K. *The Physics of Particle Accelerators: An Introduction*. Clarendon Press, 2000.
- [3] LINDE, A.; LINDE, D.; MEZHLUMIAN, A. *From the Big Bang Theory to the Theory of a Stationary Universe*. Physical Review D, v. 49, n. 4, p. 1783, 1994.
- [4] THE ATLAS COLLABORATION. *Measurements of Higgs boson Production and Couplings in Diboson Final States with the ATLAS Detector at the LHC*. Physics Letters B, vol. 726, pp. 88-119, Oct 2013.
- [5] ABDALLA, B. *O Maior Acelerador de Partículas do Mundo passa por um Upgrade*. Jornal da USP, 2019.
- [6] VIEIRA, C. L. *História da Física*. Rio de Janeiro, CBPF, 2015.
- [7] BEZRUKOV, F. and SHAPOSHNIKOV, M. *The Standard Model Higgs Boson as the Inflaton*. Physics Letters B, v. 659, n. 3, p. 703-706, 2008.
- [8] CERN. *About CERN*. CERN Document Server, 2012.
- [9] NAKAHAMA, Y. *The ATLAS Trigger System: Ready for Run-2*. CERN Document Server, ATL-DAQ-PROC-2015-006, Geneva, 2015.
- [10] ANDRADE FILHO, L. M.; PERALVA, B. S.; SEIXAS, J. M.; CERQUEIRA A. S. *Calorimeter Response Deconvolution for Energy Estimation in High-Luminosity Conditions*. IEEE Transactions on Nuclear Science, 62(6):3265–3273, Dec 2015.
- [11] ANTHONY, K. *Celebrating the First of a Kind*. Sep 2014.
- [12] WIGMANS, R. *Calorimetry*. Oxford University Press, 2018.
- [13] DUARTE, J. P. B. S. *Técnicas de Deconvolução Aplicadas à Estimação de Energia Online em Calorimetria de Altas Energias em Condições de Alta Taxa de Eventos*. Anais do XX Congresso Brasileiro de Automática, pp. 1-6, 2016.
- [14] MITRA, S. K. *Digital Signal Processing : A Computer-Based Approach*. McGraw-Hill Higher Education, New York, 2001.
- [15] BARBOSA, D. P.; ANDRADE FILHO, L. M.; CERQUEIRA, A. S.; SEIXAS, J. M. *Sparse Representation for Signal Reconstruction in Calorimeters Operating in High Luminosity*. IEEE Transactions on Nuclear Science, 64(7):1942-1949, Jul 2017.
- [16] FARIA, M. H. M. d.; FILHO, L. M. A.; DUARTE, J. B. S.; SEIXAS, J. M. *Redes Neurais para Filtragem Inversa com Aplicação em Calorímetros Operando a Alta Taxa de Eventos*. XXXV Simpósio Brasileiro de Telecomunicações e Processamento de Sinais, p. 403–407, Sep, 2017.

- [17] DAUBECHIES, I.; DEFRISE, M.; DE MOL; C. *An Iterative Thresholding Algorithm for Linear Inverse Problems with a Sparsity Constraint*. Communications on Pure and Applied Mathematics: A Journal Issued by the Courant Institute of Mathematical Sciences, vol 57, pp. 1413-1457, 2004.
- [18] MEYER, B. U. *Digital Signal Processing with Field Programmable Gate Arrays*. Heidelberg, 2007.
- [19] CALLAN, C.; DICKE, R. H.; PEEBLES, P. J. E. *Cosmology and Newtonian Mechanics*. American Journal of Physics, v. 33, n. 2, p. 105-108, 1965.
- [20] SAKURAI, J. J.; COMMINS, E. D. *Modern Quantum Mechanics*. Revised Edition. 1995.
- [21] AFSHAR S. S. *Paradox in Wave-Particle Duality*. Foundations of Physics, v. 37, n. 2, p. 295-305, 2007.
- [22] BUSCH, P.; HEINONEN, T.; LAHTI, P. *Heisenberg's uncertainty principle*. Physics Reports, v. 452, n. 6, p. 155-176, 2007.
- [23] WILLIAMS, E. R. *Accurate Measurement of the Planck Constant*. Physical Review Letters, v. 81, n. 12, p. 2404, 1998.
- [24] EINSTEIN, A. *The Meaning of Relativity: Including the Relativistic Theory of the Non-Symmetric Field*. Princeton University Press, 2014.
- [25] ROONEY, A. *A História da Física*. São Paulo, M. Books do Brasil Editora Ltda, 2013.
- [26] BRAIBANT, S.; GIACOMELLI, G.; SPURIO, M. *Particles and Fundamental Interactions*. New York, Springer-Verlag GMBH, 2011.
- [27] CERN. *The Large Hadron Collider*. CERN Accelerating Science. Disponível em: <https://home.cern/science/accelerators/large-hadron-collider>
- [28] MOUCHE, P. *Overall View of the LHC*. CERN Document Server, OPEN-PHO-ACCEL-2014-001, Jun, 2014.
- [29] BRICE, M. *LHC Restart Run 2*. CERN Document Server, April, 2015.
- [30] AAD, G. *Observation of a New Particle in the Search for the Standard Model Higgs boson with the ATLAS*. Phys. Lett. vol B716, pp. 1-29, 2012.
- [31] AGGLETON, R.; CMS COLLABORATION. *An FPGA Based Track Finder for the L1 Trigger of the CMS Experiment at the High Luminosity LHC*. Journal of Instrumentation, vol 12, p. P12019, 2017.
- [32] BOURGEOIS, D.; FITZPATRICK, C.; STAHL, S. *Using Holistic Event Information in the Trigger*. CERN Document Server, LHCb-PUB-2018-010, Geneva, 2018.
- [33] RONCHETTI, F.; BLANCO, F.; FIGUEIREDO, M.; KNOSPE, A. G.; XAPLAN-TERIS, L. *The ALICE Electromagnetic Calorimeter High Level Triggers*. JJ. Phys. Conf. Ser. vol 396, 2012.

- [34] AAD, G. *The ATLAS Experiment at the CERN Large Hadron Collider*. JINST, vol. 3, pp. S08003, 2008.
- [35] THE TOTEM COLLABORATION. *TOTEM Technical Design Report*. Technical Report, CERN/LHCC 2004-002, 2004.
- [36] THE LHCF COLLABORATION. *Technical Design Report of the LHCf Experiment*. Technical Report, CERN/LHCC 2006-004, 2006.
- [37] EVANS, L. R.; BRYANT, P. *LHC Machine*. Journal of Instrumentation, vol. 3, pp. S08001.164, 2008.
- [38] PEQUENAO, J. *Computer Generated Image of the Whole ATLAS Detector*. CERN Document Server, CERN-GE-0803012, Geneva, 2008.
- [39] ROS, E. *ATLAS Inner Detector*. Nuclear Physics B Proceedings Supplements, vol. 120, pp. 235–345, 2003.
- [40] PALESTINI, S. *The Muon Spectrometer of the ATLAS Experiment*. Nuclear Physics B Proceedings Supplements, vol. 125, pp. 237–345, 2003.
- [41] ALEKSA, M.; CLELAND, W.; ENARI, Y. *ATLAS Liquid Argon Calorimeter Phase-I Upgrade Technical Design Report*. CERN Document Server, CERN-LHCC-2013-017 and ATLAS-TDR-022, Geneva, 2013.
- [42] CARRIO, F.; KIM, H. Y.; MORENO, P.; REED, R.; SANDROK, C. *Design of an FPGA-Based Embedded System for the ATLAS Tile Calorimeter Front-End Electronics Test-Bench*. CERN Document Server, ATL-TILECAL-PROC-2013-017, Geneva, 2013.
- [43] PEQUENAO, J. *Computer Generated Image of the ATLAS Calorimeter*. CERN Document Server, CERN-GE-0803015, 2008.
- [44] ANDRADE FILHO, L. M. *Deteção e Reconstrução de Raios Cósmicos Usando Calorimetria de Altas Energias*. Tese de Doutorado, COPPE/UFRJ, Março, 2009.
- [45] PEQUENAO, J.; SCHAFFNER, P. *How ATLAS Detects Particles: Diagram of Particle Paths in the Detector*. CERN Document Server, CERN-EX-1301009, Geneva, Jan 2013.
- [46] PASCHOALIN, T. C. *Reconstrução de Energia em Calorímetros Operando em Alta Taxa de Luminosidade Usando Estimadores de Máxima Verossimilhança*. Tese de Mestrado, UFJF, Março, 2016.
- [47] PERALVA, B. S. M.; ANDRADE FILHO, L. M. A.; CERQUEIRA, A. S. *The TileCal Energy Reconstruction for Collision Data Using the Matched Filter*. CERN Document Server, ATL-TILECAL-PROC-2013-023, Geneva, 2013.
- [48] KLOUS, S. *Event Streaming in the Online System*. CERN Document Server, ATL-DAQ-PROC-2010-017, Geneva, 2010.
- [49] WATTS, G. *Review of Triggering* Nuclear Science Symposium Conference Record, IEEE, v. 1, pp. 282-287, 2003.
- [50] THE ATLAS HLT/DAQ/DCS GROUP. *ATLAS High-Level Trigger, Data Acquisition and Controls*. ATLAS Technical Report, TDR-016, CERN, 2001.

- [51] HADLEY, D. R. *Digital Filtering Performance in the ATLAS Level-1 Calorimeter Trigger*. Real Time Conference (RT), 7th IEEE-NPSS, pp. 1–6. IEEE, 2010.
- [52] RUGGIERO, F. *LHC Accelerator R&D and Upgrade Scenarios*. The European Physical Journal C - Particles and Fields, v. 34, pp. 433-442, 2004.
- [53] CEPEDA, M.; GORI, S.; ILTEN, P. *Report from Working Group 2: Higgs Physics at the HL-LHC and HE-LHC*. CERN Yellow Rep. Monogr., v. 7, pp. 221-584. 364 p., 2018.
- [54] CLELAND, W. E.; STERN, E. G. *Signal Processing Considerations for Liquid Ionization Calorimeters in a High Rate Environment*. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, vol. 338, pp. 467–497, Jan 1994.
- [55] CHAPMAN, J. *ATLAS Simulation Computing Performance and Pile-up Simulation in ATLAS*. LPCC Detector Simulation Workshop CERN, 2011.
- [56] BARBOSA D. P. *Estudo de Técnicas de Otimização para Reconstrução de Energia de Jatos no Primeiro Nível de Seleção de Eventos do Experimento ATLAS*. Tese de Mestrado, Juiz de Fora, MG, 2012.
- [57] PERALVA, B. S. M. *Detecção de Sinais de Estimação de Energia para Calorimetria de Altas Energias*. Tese de Mestrado, Juiz de Fora, MG, 2012.
- [58] DUARTE, J. P. B. S. *Estudo de Técnicas de Deconvolução para Reconstrução de Energia Online no Calorímetro Hadrônico do ATLAS*. Tese de Mestrado, UFJF, Juiz de Fora, MG, 2015.
- [59] BARBOSA D. P. *Estimação de Energia para Calorimetria em Física de Altas Energias Baseada em Representação Esparsa*. Tese de Doutorado, Juiz de Fora, MG, 2017.
- [60] GILAT, A. *Matlab com Aplicações em Engenharia* Bookman Editora, 2009.
- [61] GARVEY, J.; REES, D. *Bunch Crossing Identification for the ATLAS Level-1 Calorimeter Trigger*. CERN Document Server, CERN-ATL-DAQ-96-051, Geneva, 1996.
- [62] KAY, S. M. *Fundamentals of Statistical Signal Processing*. Prentice Hall, vol 1, New Jersey, 1993.
- [63] TEIXEIRA, T. A.; DUARTE, J. P. B. S.; ANDRADE FILHO, L. M.; SEIXAS, J. M. *Implementação em FPGA de um Método Recursivo de Deconvolução Aplicado em Calorímetros Operando a Alta Taxa de Eventos*. XXXVI Simpósio Brasileiro de Telecomunicações e Processamento de Sinais, Campina Grande, Paraíba, 2018.
- [64] ELAD, M. *Sparse and Redundant Representations*. Springer New York, New York, 2010.
- [65] FARIA, M. H. M. d. *Estimação de Energia no Primeiro Nível de Trigger do Calorímetro Hadrônico do ATLAS Utilizando Redes Neurais Artificiais*. Dissertação, PPEE/UFJF, 2017.
- [66] MATOUSEK, J.; GARTNER, B. *Understanding and Using Linear Programming*. Springer-Verlag Berlin Heidelberg, 1st Ed., Heidelberg, 2007.

- [67] TEIXEIRA, T. A. *Implementação de Métodos Iterativos de Deconvolução para Processamento On-line no Calorímetro Hadrônico do ATLAS*. Tese de Doutorado, UFJF, Juiz de Fora, MG, 2019.
- [68] YPMA, T. J. *Historical Development of the Newton–Raphson Method*. SIAM Review, v. 37, n. 4, p. 531–551, 1995.
- [69] AGUIAR, M. S.; RESENDE, M. O.; TEIXEIRA, T.; ANDRADE FILHO, L. M.; SEIXAS, J. M. *Implementação em FPGA de um Método Iterativo de Deconvolução para Operar no Sistema de Trigger do Experimento ATLAS*. XXII Encontro Nacional de Modelagem Computacional e X Encontro de Ciências e Tecnologia de Materiais, Juiz de Fora, MG, 2018.
- [70] ROCKAFELLAR, R. T. *Lagrange Multipliers and Optimality*. SIAM Review, v. 35, n. 2, p. 183–238, 1993.
- [71] KOVÁCS, Z. L. *Redes Neurais Artificiais*. Editora Livraria da Física, 2002.
- [72] STANFIELD, C. L. *Fisiologia Humana*. Person Education do Brasil, 2014.
- [73] MCCULLOCH, W.; PITTS, W. *A Logical Calculus of the Ideas Immanent in Nervous Activity*. The Bulletin of Mathematical Biophysics, Springer, 1943.
- [74] HAYKIN, S. *Neural Networks, a Comprehensive Foundation*. Pearson Education, 1999.
- [75] SANTOS, V. A. M.; SILVA, L. R. M.; ANDRADE FILHO, L. M. *Implementação de Circuitos Aritméticos em Ponto-Flutuante Utilizando Formato com Número de bits Configurável*. Congresso Brasileiro de Automática, pp. 1–2, João Pessoa, 2018.
- [76] YIANNACOURAS, P., STEFFAN, J. G., ROSE, J. *Exploration and Customization of FPGA-Based Soft Processors*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, v. 26, n. 2, pp. 266–277, Feb, 2007.
- [77] MAKNI, M.; BAKLOUTI, M.; NIAR, S.; JMAL, M. W.; ABID, M. *A Comparison and Performance Evaluation of FPGA Soft-Cores for Embedded Multi-Core Systems*. 11th International Design Test Symposium (IDT), pp. 154–159, Dec, 2016.
- [78] INTEL. *NIOS II Processor Reference Guide*. 2019.
- [79] STALLINGS, W. *Arquitetura e Organização de Computadores*. Pearson Universidades, Ed.10, 2017.
- [80] PATTERSON, D. A. *Reduced Instruction Set Computers*. Communications of the ACM, PP.8-21, 1985.
- [81] SCHILDT, H. *C Completo e Total*. Pearson Makron Books, 1997.
- [82] THOMAS, D.; MOORBY, P. *The Verilog Hardware Description Language*. Springer Science & Business Media, 2008.
- [83] CILETTI, M. D. *The Art of Assembly Language*. No Starch Press, 2003.
- [84] TOCCI, R. *Sistemas Digitais: Princípios e Aplicações*. Pearson Universidades, 2019.

- [85] SANTOS, V. A. M. *Implementação de Circuitos Aritméticos em Ponto-Flutuante*. Novas Edições Acadêmicas, 2019.
- [86] LEVINE, J. *Flex & Bison: Text Processing Tools*. O'Reilly Media, Inc., 2009.
- [87] *IEEE Standard for Floating-Point Arithmetic*. IEEE Std 754-2019 (Revision of IEEE 754-2008), 2008.
- [88] SANTOS, V. A. M.; KAPISCH, E. B.; FILHO, L. M. A. *Implementação de Circuitos Aritméticos em Ponto-Flutuante, utilizando Formato com Número de Bits Configurável*. Anais do XXII Congresso Brasileiro de Automática, 2018.
- [89] MODELSIM. *Intel FPGA Edition Simulation Quick-Start*. Intel, UG-01102, 2019.
- [90] AGUIAR, M. S.; RESENDE, M. O.; VICCINI, L. O. F.; DIAS, K.; TEIXEIRA, T. A.; ANDRADE FILHO, L. M.; SEIXAS, J. M. *Arquitetura Multi-Core de Processadores Reconfiguráveis para Reconstrução Online de Energia no Calorímetro Hadrônico do ATLAS*. XXIII Congresso Brasileiro de Automática, CBA, 2020.
- [91] AGUIAR, M. S.; VICCINI, L. O. F.; SANTOS, D. F.; RESENDE, M. O.; FARIA, M.; ANDRADE FILHO, L. M.; SEIXAS, J. M. *Processamento Multicore para Reconstrução Online de Energia por meio de Redes Neurais*. XXXVIII Simpósio Brasileiro de Telecomunicações e Processamento de Sinais, SBrT, 2020.
- [92] RESENDE, M. O.; AGUIAR, M. S.; SANTOS, D. F.; VICCINI, L. O. F.; FARIA, M.; ANDRADE FILHO, L. M.; SEIXAS, J. M. *Implementação de Redes Neurais em FPGA para Estimação de Energia no Calorímetro Hadrônico do Experimento ATLAS*. XXXIX Simpósio Brasileiro de Telecomunicações e Processamento de Sinais, SBrT, 2021.
- [93] OLIPHANT, T. E. *Python for Scientific Computing*. Computing in Science & Engineering, v. 9, n. 3, pp. 10–20, 2007.
- [94] Intel Corporation. *Intel FPGAs and Programmable Devices*. Disponível em: <https://www.intel.com.br/content/www/br/pt/products/programmable.html>
- [95] AGUIAR, M. S.; ANDRADE FILHO, L. M.; SEIXAS, J. M. *Processamento Embarcado para Implementação de um Método Iterativo de Deconvolução Visando a Reconstrução de Energia em Calorímetros de Altas Energias*. XXXVII Simpósio Brasileiro de Telecomunicações e Processamento de Sinais, 2019, Petrópolis, RJ. SBrT, 2019.
- [96] AGUIAR, M. S.; ANDRADE FILHO, L. M.; SEIXAS, J. M. *Processamento Embarcado em FPGA para Implementação de Métodos Iterativos de Desconvolução no Sistema de Trigger do ATLAS*. XL Encontro Nacional de Física de Partículas e Campos (ENFPC) e XLII Reunião de Trabalho sobre Física Nuclear no Brasil (RTFNB), 2019, Campos do Jordão, SP. XL ENFPC e XLII RTFNB, 2019.
- [97] BAER, J. L. *Arquitetura de Microprocessadores - Do Simple Pipeline ao Multiprocessador em Chip*. LTC, 2013.
- [98] PADGETT, W. T.; ANDERSON, D. V. *Fixed-Point Signal Processing*. Morgan and Claypool Publishers, 2009.

- [99] PIAZZA, F.; UNCINI A.; ZENOBI, M. *Neural Networks with Digital LUT Activation Functions*. International Conference on Neural Networks (IJCNN-93-Nagoya, Japan). Vol. 2. IEEE, 1993.
- [100] TERMURTAS, F.; GULBAG, A.; YUMUSAK, N. *A Study on Neural Networks Using Taylor Series Expansion of Sigmoid Activation Function*. Lecture Notes in Computer Science, page 389-397. Springer, 2004.
- [101] ALTERA. *Nios II Core Implementation Details*. Altera Corporation, 2015.
- [102] PRESS, W. H. *Numerical Recipes*. Cambridge University Press, Ed.2, 1992.

APÊNDICE A – Produção Bibliográfica

Publicações em Congressos Nacionais

AGUIAR, M. S.; ANDRADE FILHO, L. M. ; SEIXAS, J. M. “*Processamento Embarcado para Implementação de um Método Iterativo de Deconvolução Visando a Reconstrução de Energia em Calorímetros de Altas Energias*”. **XXXVII Simpósio Brasileiro de Telecomunicações e Processamento de Sinais**, Petrópolis, RJ, 2019.

Resumo: Este trabalho propõe a implementação, em FPGA, de um processador customizado que opera um método iterativo, baseado em Gradiente Descendente Positivo, visando sua aplicação na reconstrução online de energia em calorímetros de altas energias, respeitando a latência de operação necessária. Este algoritmo é mais eficiente na reconstrução dos sinais que o atual método em uso, o qual não é tolerante ao efeito de empilhamento de sinais que ocorre em colisionadores de partículas modernos, que operam em elevada colimação dos feixes de colisão.

AGUIAR, M. S.; RESENDE, M. O. ; TEIXEIRA, T. ; ANDRADE FILHO, L. M.; SEIXAS, J. M. “*Implementação em FPGA de um Método Iterativo de Deconvolução para Operar no Sistema de Trigger do Experimento ATLAS*”. **XXII Encontro Nacional de Modelagem Computacional e X Encontro de Ciências e Tecnologia de Materiais**, Juiz de Fora, MG. Anais do XXII ENMC X ECTM,2019.

Resumo: Aceleradores de partículas, como o LHC, em Genebra, na Suíça, colidem partículas subatômicas cujos subprodutos (partículas fundamentais da natureza) são analisados por detectores ao redor do ponto de colisão. O LHC vem passando por um processo de atualização, em que parâmetros como a energia das colisões e a quantidade de partículas por *bunch* são aumentados, impactando diretamente nos sistemas de instrumentação dos detectores, como, por exemplo, o Experimento ATLAS presente neste acelerador. Este aumento na probabilidade de ocorrência de colisões adjacentes produz um efeito conhecido como *pile-up* (empilhamento de sinais) na eletrônica de leitura dos sub-detectores do ATLAS. Para lidar com este novo desafio, métodos iterativos de deconvolução de sinais, baseados em Representação Esparsa de Dados, como o SSF, vêm sendo propostos, apresentando resultados satisfatórios. No entanto, tais métodos apresentam um alto custo computacional, de modo que o principal desafio atual é o desenvolvimento de algoritmos capazes de operar de forma online. Este trabalho apresenta um aprimoramento proposto ao SSF, permitindo reduzir a quantidade de iterações deste algoritmo em uma ordem de grandeza, facilitando, assim, a sua implementação online.

AGUIAR, M. S.; ANDRADE FILHO, L. M. ; SEIXAS, J. M. “*Processamento Embarcado em FPGA para Implementação de Métodos Iterativos de Desconvolução no Sistema de Trigger do ATLAS*”. **XL Encontro Nacional de Física de Partículas e Campos (ENFPC) e XLII Reunião de Trabalho sobre Física Nuclear no Brasil (RTFNB)**, Campos do Jordão, SP. XL ENFPC e XLII RTFNB, 2019.

Resumo: O LHC (*Large Hadron Collider*) é o principal acelerador de partículas do CERN, sendo o maior e mais energético acelerador de partículas já construído. A aquisição dos dados resultantes das colisões no LHC é realizada pelos seus detectores. Dentre eles, o ATLAS (*A Toroidal LHC Apparatus*), que é composto por sub-detectores dispostos em camadas, sendo cada uma responsável por medir propriedades específicas das partículas geradas pelas colisões. O Calorímetro Hadrônico do ATLAS (*TileCal*) opera a uma alta taxa de eventos e a estimação de energia sofre com a sobreposição de sinais em seus canais de leitura, uma vez que o tempo de resposta (largura do pulso de 150 ns) é maior que o período de colisão (25 ns). Este fenômeno é conhecido como empilhamento de sinais (*pileup*) e a ocorrência do efeito se intensificará à medida que a luminosidade das colisões do LHC aumenta com as constantes atualizações que vêm sendo feitas. O sistema atual de processamento de sinais do detector não está preparado para lidar com o *pileup*, o que pode ocasionar baixa eficiência para a detecção (*trigger*) de eventos. Neste contexto, para lidar com o problema obtendo mais eficiência na reconstrução dos sinais, foi proposta uma abordagem de separação dos sinais empilhados usando métodos de desconvolução, muito empregados em comunicação digital de dados. Neste trabalho, a proposta é implementar em FPGA (*Field-Programmable Gate Array*) um processador customizado que opera um método iterativo de desconvolução baseado em Gradiente Descendente, realizando o estudo da viabilidade de sua aplicação na reconstrução de energia no Calorímetro Hadrônico do ATLAS, tendo baixo custo em elementos lógicos e melhor eficiência que os métodos baseados em filtros FIR. Dada esta validação da topologia, a proposta para trabalhos futuros é modificar a estrutura da Unidade Lógica Aritmética (ULA) do processador de forma a realizar os cálculos matriciais dentro do tempo requerido pelo sistema de aquisição do ATLAS.

AGUIAR, M. S.; RESENDE, M. O.; VICCINI, L. O. F. ; DIAS, K.; TEIXEIRA, T. A.; ANDRADE FILHO, L. M.; SEIXAS, J. M. “*Arquitetura Multi-Core de Processadores Reconfiguráveis para Reconstrução Online de Energia no Calorímetro Hadrônico do ATLAS.*” In: **XXIII Congresso Brasileiro de Automática**, CBA, novembro, 2020.

Resumo: Calorímetros são sistemas usados para medir a energia de partículas fundamentais que atravessam o seu material. Para tal, pulsos elétricos são gerados por sensores posicionados ao longo do material absorvedor do calorímetro. Técnicas de processamento digital de sinais são empregadas para detectar e estimar parâmetros destes pulsos de forma a inferir a energia das partículas. Tais técnicas, quando implementadas online, necessitam ser de baixa complexidade para que seja possível a sua implementação em *hardware* dedicado. No entanto, técnicas baseadas em teoria de Representação Esparsa (RE) de dados vêm se destacando quanto à eficiência de reconstrução, mas, devido ao seu alto custo computacional, ainda não são utilizadas como uma opção para processamento online. Neste trabalho, é apresentada a customização de um processador e sua utilização em uma arquitetura *multi-core* em FPGA, possibilitando a implementação online de técnicas baseadas em RE. Para demonstrar seu funcionamento, foi utilizado o calorímetro hadrônico do Experimento ATLAS como ambiente de aplicação.

AGUIAR, M. S.; VICCINI, L. O. F. ; SANTOS, D. F. ; RESENDE, M. O. ; FARIA, M. ; ANDRADE FILHO, L. M. ; SEIXAS, J. M. “*Processamento Multicore para Reconstrução Online de Energia por meio de Redes Neurais.*” In: **XXXVIII Simpósio Brasileiro de Telecomunicações e Processamento de Sinais**, SBrT, novembro, 2020.

Resumo: A reconstrução de energia em calorímetros é um processamento executado em pulsos gerados em seus eletrodos para estimar a energia de partículas subatômicas que atravessam seu material. Características não-gaussianas do ruído de calorímetros operando em altas taxas de evento demonstram que métodos não-lineares de estimação são mais indicados. No entanto, tais métodos tendem a ter um alto custo computacional, o que dificulta sua implementação online. Apresenta-se, com este trabalho, a implementação de uma rede neural embarcada em um processamento *multicore* em FPGA capaz de operar a uma taxa de aquisição acima de 40 MHz, bem como uma implementação no Calorímetro do Experimento ATLAS.

TEIXEIRA, T. A.; AGUIAR, M. S.; ANDRADE FILHO, L. M. ; SEIXAS, J. M. “*Método Iterativo de Representação Esparsa Implementado em FPGA para Aplicação em Calorimetria.*” In: **XXXVIII Simpósio Brasileiro de Telecomunicações e Processamento de Sinais**, SBrT, novembro, 2020.

Resumo: O acelerador de partículas LHC vem passando por um processo de atualização, produzindo o efeito conhecido como *pile-up* (empilhamento de sinais) na eletrônica de leitura dos detectores. Como o algoritmo atualmente utilizado para a estimação da amplitude dos sinais gerados nessas colisões não é tolerante a esse efeito, este trabalho propõe a implementação em *hardware* de um método iterativo baseado em uma variante do método *Separable Surrogate Functionals* que recupera a informação da amplitude dos sinais sobrepostos dentro de uma janela de aquisição. Tal implementação permite que dezenas de canais possam ser implementados em paralelo dentro de uma única FPGA, além de respeitar a latência de operação necessária a sua implementação *online*.

RESENDE, M. O. ; AGUIAR, M. S.; SANTOS, D. F. ; VICCINI, L. O. F. ; FARIA, M. ; ANDRADE FILHO, L. M. ; SEIXAS, J. M. “*Processamento Multicore para Reconstrução Online de Energia por meio de Redes Neurais.*” In: **XXXVIX Simpósio Brasileiro de Telecomunicações e Processamento de Sinais**, SBrT, setembro, 2021.

Resumo: Em experimentos de Física de Altas Energias é possível medir a energia das partículas fundamentais geradas por meio da estimação da amplitude dos sinais oriundos da eletrônica de leitura em calorímetros. Recentemente, foi proposta a implementação em FPGA de uma Rede Neural com função de ativação por *look-up table*, visando a estimação de amplitude do Calorímetro Hadrônico do Experimento ATLAS, utilizando grande quantidade de memória embarcada. Neste trabalho, propõe-se o desenvolvimento de um circuito para aproximação da função de ativação por Série de Taylor, reduzindo drasticamente a utilização de memórias internas, permitindo uma maior quantidade de canais por chip.

ANEXO A – Descrição dos Algoritmos do Método SSF

Nesse anexo serão descritos os algoritmos que contém os passos seguidos, de forma simplificada, para cada implementação que foi realizada no SAPHO, em linguagem C^+ .

Para o melhor entendimento desta etapa, nas Figuras 55, 56, 57, 58 e 59 podem ser observadas as equações onde está destacada a ordem de procedência a qual cada operação matricial foi calculada, nos respectivos algoritmos, de forma que as operações foram realizadas na direção dos blocos mais internos para os mais externos, ou seja, primeiro os de cor azul, seguidos pela cor vermelha, depois a cor roxo e, por fim, os de cor verde.

Vale ressaltar que devem ser adicionados zeros ao sinal que contém os dados de saída de forma a padronizá-lo, mais especificamente, são adicionados três zeros no início da janela de saída e quatro zeros no final da mesma, de forma que, ao observar o sinal de saída contendo diversas janelas em sequência, ocorrem 48 intervalos temporais de 25 ns com colisões e 7 intervalos temporais de 25 ns sem colisões, de acordo com o padrão de colisões mais recente adotado pelo LHC, conhecido como 48b7e [63]. Com isso, as janelas de saída e de entrada de dados possuem o mesmo tamanho (55 termos).

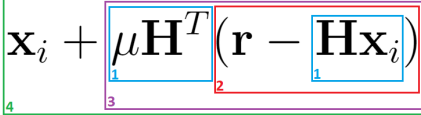
$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mu \mathbf{H}^T (\mathbf{r} - \mathbf{H} \mathbf{x}_i)$$


Figura 55 – Operações do Algoritmo 1.

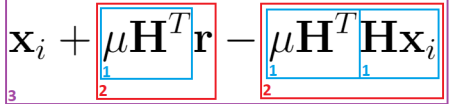
$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mu \mathbf{H}^T \mathbf{r} - \mu \mathbf{H}^T \mathbf{H} \mathbf{x}_i$$


Figura 56 – Operações do Algoritmo 2.


$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mu (\mathbf{H} \mathbf{s} - \mathbf{A} \mathbf{x}_i)$$


Figura 57 – Operações do Algoritmo 3.


$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mu (\mathbf{H} \mathbf{s} - \mathbf{A} \mathbf{x}_i)$$


Figura 58 – Operações do Algoritmo 4.

$$\mathbf{x}_i = \mathbf{B}^T \mathbf{x}_i$$

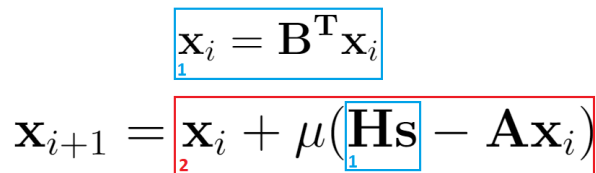
$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mu (\mathbf{H} \mathbf{s} - \mathbf{A} \mathbf{x}_i)$$


Figura 59 – Operações do Algoritmo 5.

Algoritmo 1: Implementação em ponto-flutuante (versão 1.0).

```

1 int i;
2 int mi = 0.25;
3 float x(48);
4 float aux(48);
5 float y(55);
6 float d(55);
7 float h(55,48) = load(matrizH);
8 float mih(55,48);
9 for (i = 0:1:47) do
10 | x(i) = 0;
11 end
12 for (i = 0:1:54) do
13 | d(i) = in(0);
14 | %% vetor d recebeu os dados de entrada vindos da porta de índice 0.
15 end
16 for (i = 0:1:149) do
17 | for (i = 0:1:54) do
18 | | y(i) = 0;
19 | end
20 | for (i = 0:1:47) do
21 | | aux(i) = 0;
22 | end
23 | mih = mi * hT;
24 | y = d - h * x;
25 | aux = mih * y;
26 | x = x + aux;
27 | for (i = 0:1:47) do
28 | | if (x(i) < 0) then
29 | | | x(i) = 0;
30 | | end
31 | end
32 end
33 %% abaixo está a janela de saída, no formato padrão, na porta de índice 0.
34 out(0, 0);
35 out(0, 0);
36 out(0, 0);
37 for (i = 0:1:47) do
38 | out(0, x(i));
39 end
40 out(0, 0);
41 out(0, 0);
42 out(0, 0);
43 out(0, 0);

```

Algoritmo 2: Implementação em ponto-flutuante (versão 1.1).

```

44 int i;
45 float mi = 0.25;
46 float x(48);
47 float aux(48);
48 float mihd(48);
49 float y(55);
50 float d(55);
51 float h(55,48) = load(matrizH);
52 float mih(55,48);
53 for (i = 0:1:47) do
54 |   x(i) = 0;
55 end
56 for (i = 0:1:54) do
57 |   d(i) = in(0);
58 |   %% vetor d recebeu os dados de entrada vindos da porta de índice 0.
59 end
60 mih = mi * hT;
61 mihd = mih * d;
62 for (i = 0:1:149) do
63 |   for (i = 0:1:54) do
64 | |   y(i) = 0;
65 |   end
66 |   for (i = 0:1:47) do
67 | |   aux(i) = 0;
68 |   end
69 |   y = h * x;
70 |   y = d - y;
71 |   aux = mih * y;
72 |   x = x + mihd - aux;
73 |   for (i = 0:1:47) do
74 | |   if (x(i) < 0) then
75 | | |   x(i) = 0;
76 | |   end
77 |   end
78 end
79 %% abaixo está a janela de saída, no formato padrão, na porta de índice 0.
80 out(0, 0);
81 out(0, 0);
82 out(0, 0);
83 for (i = 0:1:47) do
84 |   out(0, x(i));
85 end
86 out(0, 0);
87 out(0, 0);
88 out(0, 0);
89 out(0, 0);

```

Algoritmo 3: Implementação em ponto-flutuante (versões 2.0 e 2.1).

```

90 int i;
91 float mi = 0.25;
92 float Hs(48);
93 float x(48);
94 float aux(48);
95 float d(55);
96 float A(48,48) = load(matrizA);
97 %% a matriz A tem 13 termos não nulos em cada coluna.
98 float H(55,48) = load(matrizH);
99 for (i = 0:1:47) do
100 | x(i) = 0;
101 end
102 for (i = 0:1:47) do
103 | aux(i) = 0;
104 end
105 for (i = 0:1:54) do
106 | d(i) = in(0);
107 | %% vetor d recebeu os dados de entrada vindos da porta de índice 0.
108 end
109 Hs = HT * d;
110 for (i = 0:1:149) do
111 | %% o produto abaixo foi declarado/realizado termo a termo, ignorando as
    | multiplicações por zero. Na versão 2.0 da implementação os termos não nulos
    | de A são escritos como variáveis. Na versão 2.1 eles são escritos como
    | constantes, ou seja, seus valores literais.
112 | aux = A * x;
113 | aux = Hs - aux;
114 | x = x + mi * aux;
115 | for (i = 0:1:47) do
116 | | if (x(i) < 0) then
117 | | | x(i) = 0;
118 | | end
119 | end
120 end
121 %% abaixo está a janela de saída, no formato padrão, na porta de índice 0.
122 out(0, 0);
123 out(0, 0);
124 out(0, 0);
125 for (i = 0:1:47) do
126 | out(0, x(i));
127 end
128 out(0, 0);
129 out(0, 0);
130 out(0, 0);
131 out(0, 0);

```

Algoritmo 4: Implementação em ponto-flutuante (versão 2.2).

```

132 int i;
133 float Hs0, Hs1, Hs2, ... Hs46, Hs47;
134 float x0 = 0, x1 = 0, x2 = 0, ... x46 = 0, x47 = 0;
135 %% o sinal x é declarado como variáveis e inicializado com zeros termo a termo ao
    ser declarado.
136 float d0 = 0, d1 = 0, d2 = 0, ... d53 = 0, d54 = 0;
137 %% Para H, A e B abaixo, apenas os termos não nulos serão utilizados nas
    operações e os mesmos são escritos termo a termo de forma literal para todos os
    cálculos.
138 d0 = in(0);
139 d1 = in(0);
140 ...
141 d53 = in(0);
142 d54 = in(0);
143 %% variáveis d recebem os dados de entrada vindos da porta de índice 0.
144 Hs =  $H^T * d$ ;
145 %% no SAPHO, essa operação do cálculo de Hs é escrita termo a termo e os
    valores de  $H^T$  são inseridos diretamente, como constantes.
146 for (i = 0:1:149) do
147     aux = x + 0.25*(Hs - A*x);
148     %% no SAPHO, essa operação do cálculo de aux é escrita termo a termo e os
        valores de  $\mu$ , Hs e A são inseridos diretamente, como constantes.
149     if (x0 < 0) then
150         | x0 = 0;
151         | else x0 = aux0;
152     end
153     if (x1 < 0) then
154         | x1 = 0;
155         | else x1 = aux1;
156     end
157     ...
158     if (x47 < 0) then
159         | x47 = 0;
160         | else x47 = aux47;
161     end
162 end
163 %% abaixo está a janela de saída, no formato padrão, na porta de índice 0.
164 out(0, 0);
165 out(0, 0);
166 out(0, 0);
167 out(0, x0);
168 ...
169 out(0, x47);
170 out(0, 0);
171 out(0, 0);
172 out(0, 0);
173 out(0, 0);

```

Algoritmo 5: Implementação em ponto-flutuante (versões 3.0 e 3.1).

```

174 int i;
175 float Hs0, Hs1, Hs2, ... Hs46, Hs47;
176 float x0 = 0, x1 = 0, x2 = 0, ... x46 = 0, x47 = 0;
177 %% a função PSET (ou @) é responsável por zerar os termos negativos. A
    diferença entre as versões 3.0 e 3.1 é que a 3.1 tem o PSET implementado, a 3.0
    não tem.
178 float d0 = 0, d1 = 0, d2 = 0, ... d53 = 0, d54 = 0;
179 d0 = in(0);
180 d1 = in(0);
181 ...
182 d53 = in(0);
183 d54 = in(0);
184 %% variáveis d recebem os dados de entrada vindos da porta de índice 0.
185 Hs =  $H^T * d$ ;
186 %% os valores de  $H^T$  são inseridos diretamente, como constantes.
187 x0 = d3;
188 x1 = d4;
189 ...
190 x46 = d49;
191 x47 = d50;
192 aux =  $B^T * x$ ;
193 %% esta operação do calculo de aux é realizada termo a termo, e os valores de  $B^T$ 
    são inseridos diretamente, como constantes.
194 x0 @ aux0;
195 ...
196 x47 @ aux47;
197 for (i = 0:1:17) do
198     aux = x + 0.25*(Hs - A*x); %% esta operação do calculo de aux é realizada
        termo a termo, e os valores de Hs e A são inseridos diretamente, como
        constantes.
199     x0 @ aux0;
200     ...
201     x47 @ aux47;
202 end
203 %% abaixo está a janela de saída, no formato padrão, na porta de índice 0.
204 out(0, 0);
205 out(0, 0);
206 out(0, 0);
207 out(0, x0);
208 out(0, x1);
209 ...
210 out(0, x46);
211 out(0, x47);
212 out(0, 0);
213 out(0, 0);
214 out(0, 0);
215 out(0, 0);

```

ANEXO B – Tutorial: Criando um Projeto com o SAPHO

Neste tutorial são ilustrados os passos desde a criação de um projeto na interface de desenvolvimento do SAPHO até a configuração e visualização do processador no Quartus.

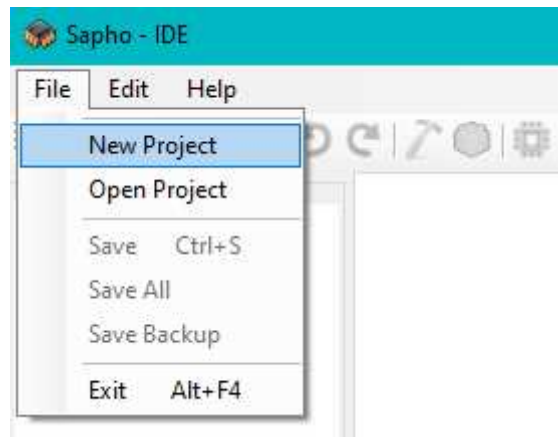


Figura 60 – Tutorial SAPHO: criando um novo projeto.

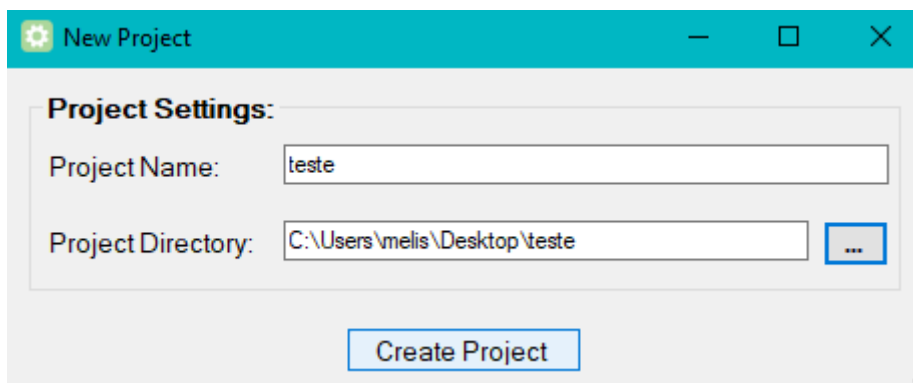


Figura 61 – Tutorial SAPHO: configurando o diretório do projeto.



Figura 62 – Tutorial SAPHO: criando um processador.

Configuration Wizard

General Settings:
Processor Name: teste

ULA Settings:
Processor Type: Fixed Point Floating Point
N bits: 16 Nb Mantissa: Nb Exponent:

Memory Stack Settings:
Data Stack Size: 4 Instruction Stack Size: 4

I/O Settings:
Number of Input Ports: 2 Number of Output Ports: 2

Gain
Gain: 5

Generate

Figura 63 – Tutorial SAPHO: configurando os parâmetros do processador.

Sapho - IDE

File Edit Help

Hierarchy:

```

1 #PRNAME teste
2 #DIRNAM "C:\Users\melis\Desktop\teste\teste\Hardware\teste_H"
3 #DATYPE 0
4 #NUBITS 16
5 #NDSTAC 4
6 #SDEPTH 4
7 #NUIOIN 2
8 #NUIOOU 2
9 #NUGAIN 32
10
11 void main()
12 {
13
14 }

```

Console Output

Figura 64 – Tutorial SAPHO: interface de desenvolvimento em C⁺.

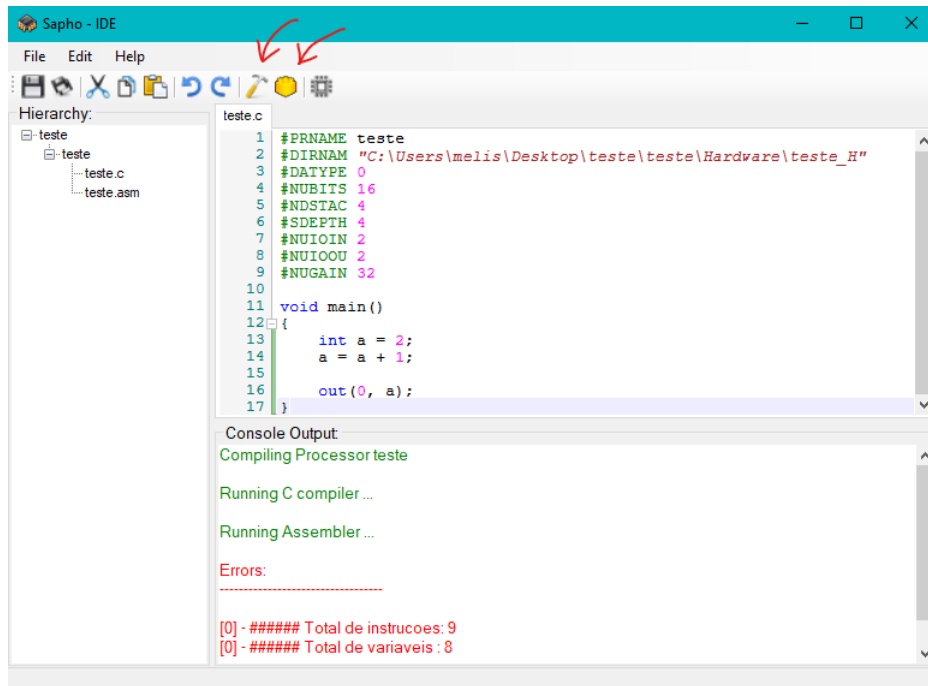


Figura 65 – Tutorial SAPHO: compilando o código.

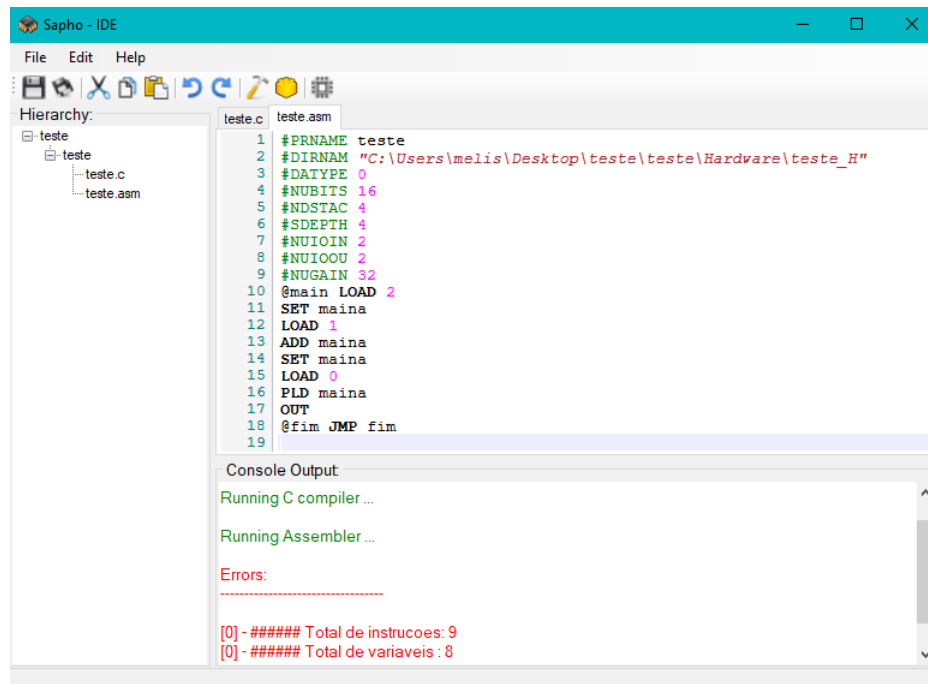


Figura 66 – Tutorial SAPHO: código em *Assembly* gerado.

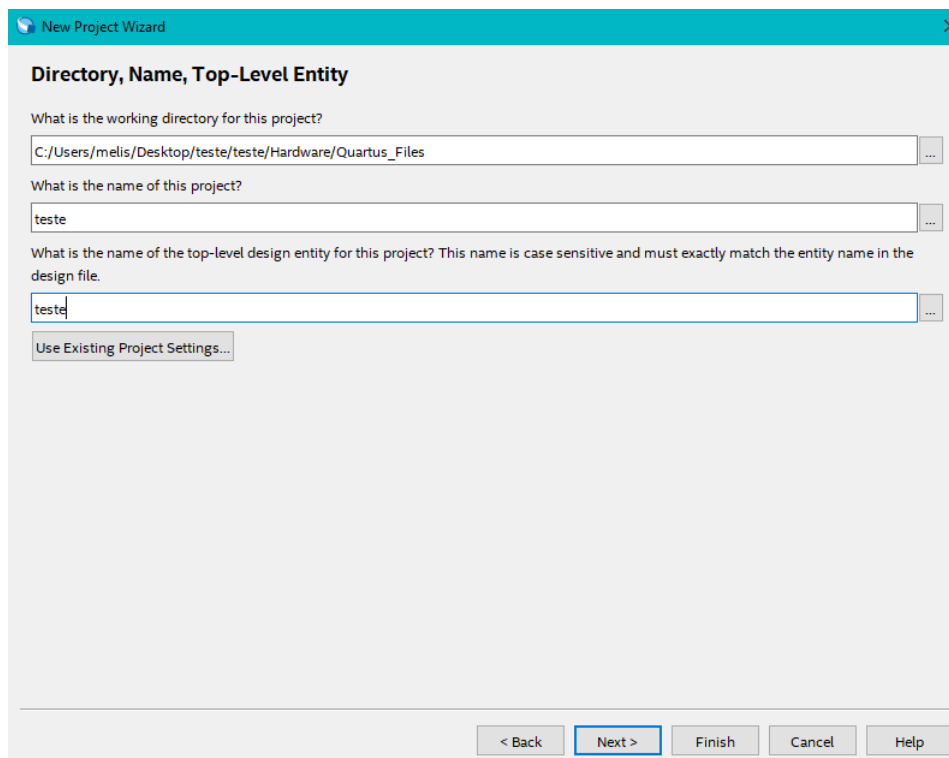


Figura 67 – Tutorial SAPHO: abrindo o projeto no Quartus.

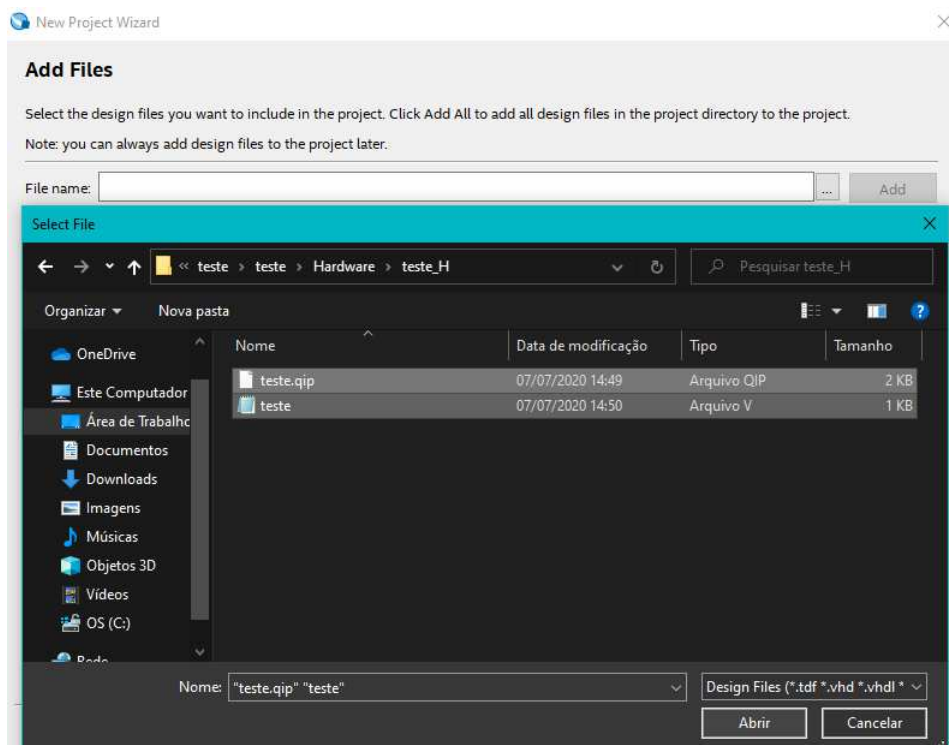


Figura 68 – Tutorial SAPHO: selecionando os arquivos de parametrização.

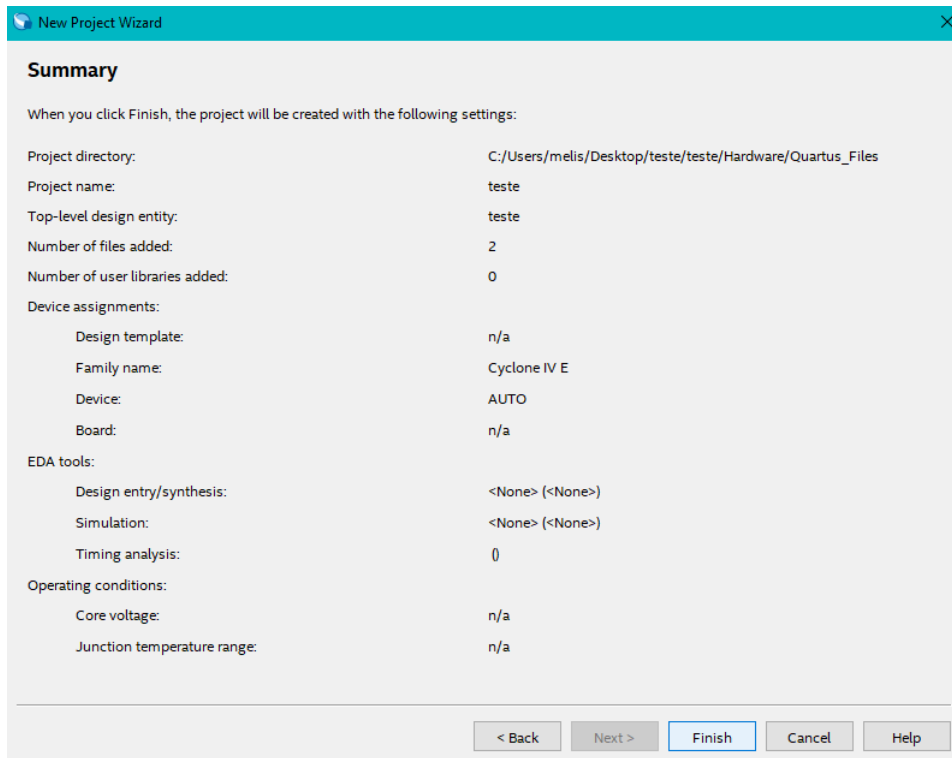


Figura 69 – Tutorial SAPHO: resumo do projeto criado no Quartus.

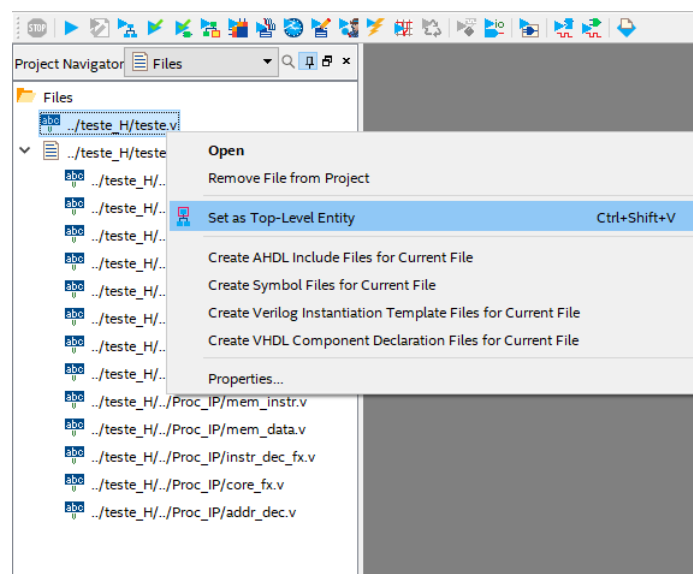
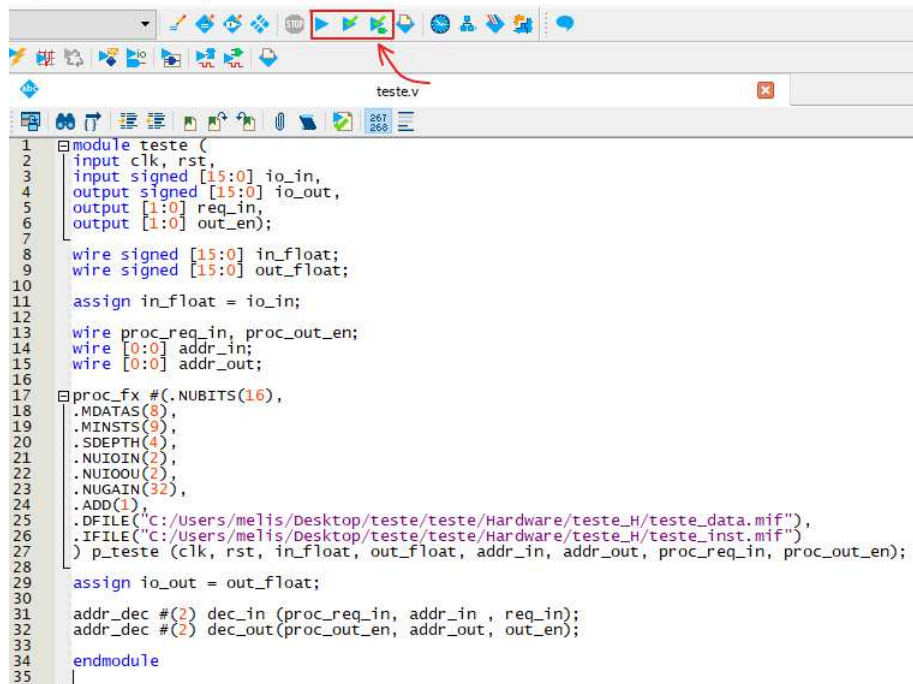


Figura 70 – Tutorial SAPHO: selecionando o arquivo principal.



```

1 module teste (
2   input clk, rst,
3   input signed [15:0] io_in,
4   output signed [15:0] io_out,
5   output [1:0] req_in,
6   output [1:0] out_en);
7
8   wire signed [15:0] in_float;
9   wire signed [15:0] out_float;
10
11   assign in_float = io_in;
12
13   wire proc_req_in, proc_out_en;
14   wire [0:0] addr_in;
15   wire [0:0] addr_out;
16
17   proc_fx #(C.NUBITS(16),
18     .MDATAS(8),
19     .MINSTS(9),
20     .SDEPTH(4),
21     .NUIOIN(2),
22     .NUIOOU(2),
23     .NUGAIN(32),
24     .ADD(1),
25     .DFILE("C:/users/melis/Desktop/teste/teste/Hardware/teste_H/teste_data.mif"),
26     .IFILE("C:/users/melis/Desktop/teste/teste/Hardware/teste_H/teste_inst.mif")
27   ) p_teste (clk, rst, in_float, out_float, addr_in, addr_out, proc_req_in, proc_out_en);
28
29   assign io_out = out_float;
30
31   addr_dec #(2) dec_in (proc_req_in, addr_in, req_in);
32   addr_dec #(2) dec_out (proc_out_en, addr_out, out_en);
33
34 endmodule
35

```

Figura 71 – Tutorial SAPHO: compilando o projeto.

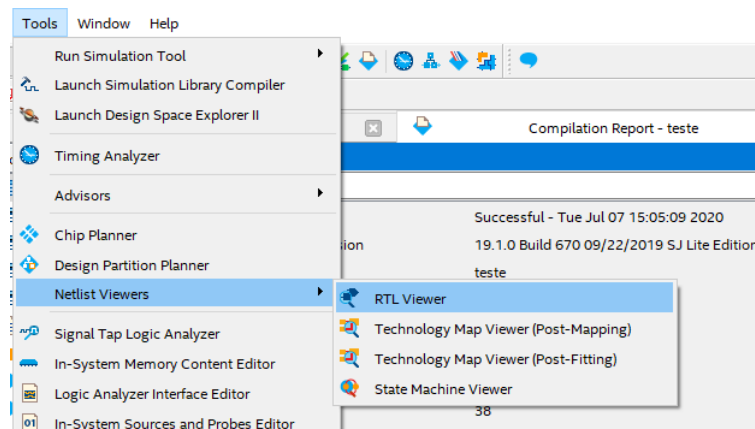


Figura 72 – Tutorial SAPHO: visualizando o processador criado.

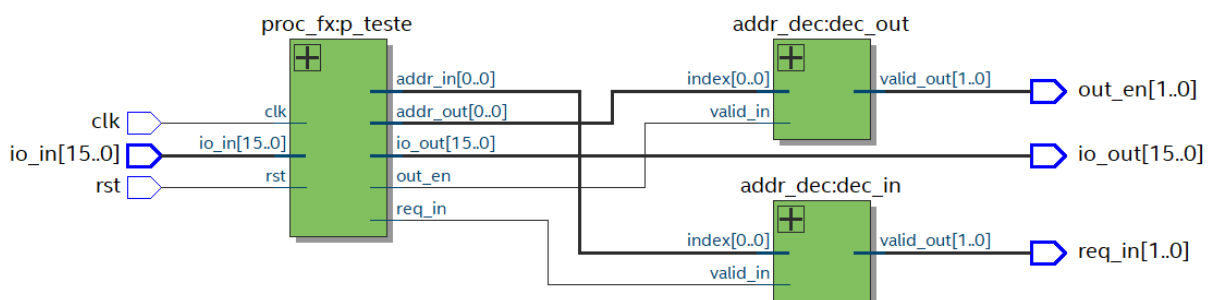


Figura 73 – Tutorial SAPHO: diagrama do *top level* do processador desenvolvido.