View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Pantheon

UMA ARQUITETURA RECONFIGURÁVEL PARA PROCESSAMENTO IN SITU UTILIZANDO REDES NEURAIS SEM PESOS

Victor da Cruz Ferreira

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientadores: Felipe Maia Galvão França
Alexandre Solon Nery

Rio de Janeiro
Março de 2018

UMA ARQUITETURA RECONFIGURÁVEL PARA PROCESSAMENTO IN SITU UTILIZANDO REDES NEURAIS SEM PESOS

Victor da Cruz Ferreira

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

____
Prof. Felipe Maia Galvão França, Ph.D.

____
Prof. Alexandre Solon Nery, D.Sc.

____
Prof. Claudio Luis de Amorim, Ph.D.

____
Prof. Cristiana Barbosa Bentes, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
MARÇO DE 2018

Ferreira, Victor da Cruz

Uma arquitetura reconfigurável para processamento In Situ utilizando Redes Neurais sem Pesos/Victor da Cruz Ferreira. – Rio de Janeiro: UFRJ/COPPE, 2018.

X, 41 p.: il.; 29, 7cm.

Orientadores: Felipe Maia Galvão França
Alexandre Solon Nery

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2018.

Referências Bibliográficas: p. 36 – 39.

1. FPGA. 2. in-situ. 3. Dataflow. I. França, Felipe Maia Galvão *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Agradecimentos

Agradeço a Coordenação de Aperfeiçoamento de Pessoal de Nível Superior(CAPES) pelo suporte financeiro. Agradeço também tanto a Universidade Federal do Rio de Janeiro quanto a Universidade Estadual do Rio de Janeiro, por disponibilizar os recursos e espaço para execução deste trabalho. Agradeço aos professores Alexandre Nery, por todo apoio e conhecimento compartilhado durante todo o processo de implementação e redação desta dissertação de Mestrado, professor Felipe França pelas oportunidades, experiencias, confiança e conhecimentos repassadas para mim que me ajudaram me tornar um profissional mais apto e a trilhar o caminho deste trabalho e ao professor Leandro Marzulo que sempre esteve presente desde a minha graduação e me guiou com ideias e motivações a este trabalho e caminho acadêmico no qual faço parte. Agradeço a minha família por sempre ter me apoiado em minhas decisões de carreira e suporte financeiro e emocional. Por fim, agradeço meus amigos por sempre estarem comigo quando preciso e estarem dispostos me dar suporte necessário em momentos complicados.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

UMA ARQUITETURA RECONFIGURÁVEL PARA PROCESSAMENTO IN SITU UTILIZANDO REDES NEURAIS SEM PESOS

Victor da Cruz Ferreira

Março/2018

Orientadores: Felipe Maia Galvão França
Alexandre Solon Nery

Programa: Engenharia de Sistemas e Computação

Há alguns anos atrás, limitações em energia e espaço entre transistores começou a mover a industria de circuitos integrados para uma nova fronteira que envolvia arquiteturas heterogêneas de computadores e designs voltados para um menor consumo de energia. Hoje em dia, Field Programmable Gate Arrays (FPGAs) são de enorme importância para estes sistema por conta de sua alta flexibilidade para programação e baixo consumo de energia. Porém, por conta do aumento do número de aceleradores e dispositivos paralelos, existe um grande aumento na movimentação de dados na rede. Este aumento no trafego pode resultar em problemas na performance do sistema. Uma forma de minimizar a movimentação de dados é processar ele in-situ, isto é, mais próximo de onde o dado se encontra. Com este conceito em mente, este trabalho desenvolveu e avaliou um sistema de disco inteligente que realiza reconhecimento facial através de um rede neural sem peso implementada em uma FPGA através da ferramente High-Level Synthesis (HLS). Resultados preliminares em performance, area de circuito e consumo de energia demonstram que o co-processador desenvolvido pode aprender e reconhecer faces mais rápido do que o microprocessador ARM da mesma FPGA, enquanto reduz significativamente o trafego na rede.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

A RECONFIGURABLE ARCHITECTURE FOR IN SITU PROCESSING USING WEIGHTLESS NEURAL NETWORKS

Victor da Cruz Ferreira

March/2018

Advisors: Felipe Maia Galvão França
Alexandre Solon Nery

Department: Systems Engineering and Computer Science

A couple years ago, limitations in power and transistor area began to charge the integrated circuit industry into a new frontier that involved heterogeneous computer architectures and energy-efficient designs. Now-a-days, Field Programmable Gate Arrays (FPGAs) play an important role on these systems due to their high flexibility and energy-efficient processing capabilities. However, due to the increasing number of accelerators and new devices embedded with parallel capabilities there is an increase in data movement through the network. This increase in traffic may result in overall system performance depreciation. One way to decrease the data movement is to process the data in-situ, which means processing the data near it's location. With this concept in mind, this work aims at designing and evaluating a smart disk system for in-situ face recognition through a Weightless artificial Neural Network (WNN) co-processor designed in High-Level Synthesis (HLS) and implemented in the system's FPGA. Preliminaries results in performance, circuit-area and energy consumption results shows that the co-processor can efficiently learn and recognize faces faster than the system's embedded ARM processor, while also significantly reducing network traffic.

Sumário

Lista de Figuras ix

Lista de Tabelas x

1 Introdução 1

1.1 Motivação 3
1.2 Objetivos 3
1.3 Contribuições 4
1.4 Organização 5

2 Fundamentação Teórica 6

2.1 Arquiteturas Reconfiguráveis 6
2.2 High-Level Synthesis 8
2.3 O Modelo Dataflow 9
2.4 Reconhecimento Facial 12
2.5 Trabalhos Relacionados 15

3 O Disco Inteligente 17

3.1 Co-Processador Neural 17
3.2 Arquitetura do Sistema 19
3.3 Comunicação e Cartão de Memória 20
3.4 Suporte para Sucuri 22

4 Resultados Experimentais 25

4.1 Caso de teste 25
4.2 Execução do Caso de Teste 28
4.3 Avaliação do microprocessador ARM 29
4.4 Tempo de reposta do File Server 30
4.5 Área do Circuito 31
4.6 Consumo de Energia 32

5 Conclusão 34

Referências Bibliográficas **36**

A Código Fonte da Wisard **40**

Lista de Figuras

1.1 Visão geral de um sistema heterogêneo moderno. 2

2.1 Representação da arquitetura FPGA divida em blocos lógicos, uma rede de interconexão e portas de entrada e saída. 7
2.2 Exemplo de um programa dataflow (A) e seu grafo de dependência correspondente (B), reproduzido de [1] 9
2.3 Componentes internos da arquitetura da Sucuri. Reproduzido de: [2] 11
2.4 Exemplo de especialização da classe `Node` com o nó `Feeder` 12
2.5 Mapeamento de uma imagem preto e branco 4x3 na Wisard, onde cada *pixel* é um quadrado preto com valor 1 ou branco com valor zero 13

3.1 Interface do Co-Processador Neural. 19
3.2 O Co-processador Wisard implementado no `Block Design do Vivado` 20
3.3 Exemplo de como ficaria o grafo de dependências utilizando os nós especializados 23
3.4 Arquitetura final do Sistema Co-Processador 24

4.1 Estes são os três grafos de dependência modelados de acordo com o caso de teste 27
4.2 Média dos tempos medidos (em segundos) quando executando o caso de teste e variando a latência. 29
4.3 Relatório de consumo de energia gerado pelo Vivado 32

Lista de Tabelas

4.1 Microprocessador ARM versus Co-processador Neural em ciclos do ARM . . . 30
4.2 Média dos tempos de resposta do `File Server` em milissegundos . . . 30
4.3 Consumo de recursos na FPGA ZC706 gerado pelo Vivado 2017.3 . . 31

Capítulo 1

Introdução

A densidade de transistores em um circuito integrado é hoje muito maior do que há décadas atrás graças aos avanços da tecnologia de semicondutores, que possibilitou uma revolução da indústria de micro-processadores e circuitos integrados ao longo dos últimos anos. As arquiteturas de múltiplos núcleos (*multi-core*), por exemplo, trouxeram a tona o paradigma da programação paralela, a fim de proporcionar e viabilizar o uso dos vários núcleos de processamento agora disponíveis. Atualmente, existe uma gama de dispositivos aceleradores que são geralmente baseados em diferentes arquiteturas de múltiplos núcleos e que podem trabalhar em conjunto na execução de uma ou várias tarefas simultaneamente. Estes sistemas são chamados de Sistemas Heterogêneos. Em geral, cada um dos dispositivos aceleradores é especializado em algum tipo de tarefa. As unidades de processamento gráfico (*Graphics Processing Units - GPUs*) são conhecidas pela sua natureza SIMD (*Single-Instruction Multiple-Data*), que consegue executar uma instrução em vários núcleos com dados diferentes, o que é ideal para execução de aplicações com alto paralelismo de dados. Por outro lado, as unidades de processamento tradicionais (*Central Processing Units - CPUs*) são otimizadas para a execução eficiente de código sequencial de natureza SISD (*Single-Instruction Single-Data*). Para tanto, elas possuem unidades especializadas em hardware para exploração de paralelismo em nível de instrução. Juntas, CPU e GPU podem compor um sistema heterogêneo para execução de aplicações com diferentes níveis de paralelismo a serem explorados.

As FPGAs (*Field Programmable Gate Arrays*) são outra classe de dispositivo acelerador que tem se destacado recentemente por sua alta flexibilidade e baixo consumo de energia [3]. As FPGAs são compostas por blocos lógicos reconfiguráveis (*Configurable Logic Blocks - CLBs*) que juntos podem ser usados para implementar diferentes tipos de sistemas digitais em hardware conectados entre si também por interconexões reconfiguráveis. Logo, uma FPGA pode ser usada para implementar um caminho de dados especializado para execução eficiente do caminho crítico de uma aplicação ou de um domínio de aplicações. A chave para o alto desempenho

em FPGAs está no seu hardware reconfigurável abundante, que permite a execução de várias operações em paralelo, ainda que a uma menor frequência de operação (50 a 200 MHz) se comparada à frequência de operação média de uma GPU (1 a 1.5 GHz). A Figura 1.1 apresenta uma visão geral de um sistema heterogêneo moderno, composto de diferentes unidades de processamento e que podem executar em diferentes configurações de sistemas operacionais, linguagens, etc.

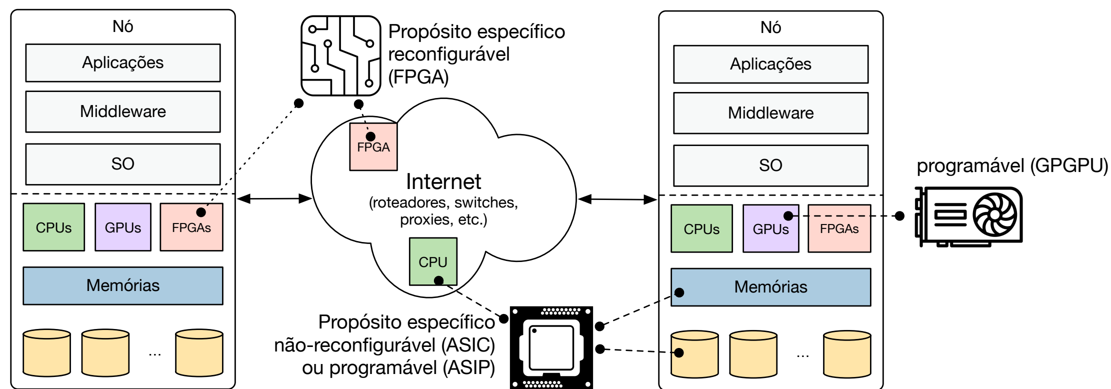


Figura 1.1: Visão geral de um sistema heterogêneo moderno.

Um dos problemas de todo esse potencial de processamento e paralelismo presente nos sistemas heterogêneos está no uso eficiente dos diferentes recursos computacionais simultaneamente, cada um especializado para execução de um tipo de tarefa. Além disso, a divisão de uma aplicação em tarefas é de responsabilidade do programador. Otimizações feitas pelo compilador e processador procuram automatizar certos aspectos. Porém, a comunicação e a divisão do programa ainda são feitas exclusivamente pelo desenvolvedor, que precisa levar em consideração os acessos a memória em regiões críticas que não devem ser escritas e lidas ao mesmo tempo, podendo gerar condições de corrida. Em resumo, programação paralela não é uma tarefa trivial.

O modelo *Dataflow* é uma alternativa promissora ao tradicional modelo de *Von Neumann*. Proposto na década de 70, o modelo Dataflow executa as operações do programa segundo seu fluxo de dados, o que acaba por expor o paralelismo de operações de forma natural. Para tanto, o programador precisa descrever o programa segundo um grafo de dependências. O modelo Dataflow também pode ser utilizado em conjunto com as arquiteturas tradicionais, sendo capaz de obter desempenho equivalente ao de outras bibliotecas de paralelização como OpenMP e PThreads [4], que também procuram facilitar a escrita de programas paralelos. A programação Dataflow consiste em dividir a aplicação em blocos conectados de acordo com suas dependências. Desta forma, blocos que não possuem dependências entre si podem executar em paralelo, desde que existam unidades funcionais suficientes.

1.1 Motivação

Com o aumento no poder de processamento e de recursos é natural que aplicações consigam processar mais dados em menos tempo, gerando cada vez uma maior quantidade de informações que são frequentemente acessadas. Além disso, existe uma grande variedade de recursos de armazenamento disponíveis no mercado, com diferentes velocidades de leitura/escrita, diferentes capacidades de armazenamento e interligados em rede. Logo, o local onde os dados serão armazenados pode fazer uma grande diferença no desempenho geral de uma aplicação, dado que os dispositivos na hierarquia de memória não tem a mesma velocidade de acesso. Custos de movimentação frequente para um lugar fisicamente distante ou acessos frequentes a dispositivos com menor tempo de resposta podem impactar negativamente o desempenho do sistema como um todo. O processamento *In-Situ* procura diminuir este tipo de movimentação trazendo o processamento para próximo de onde o dado se encontra, ou seja, incorporando capacidade de processamento nos dispositivos de armazenamento para que sejam capazes de processar dados localmente [5–7]. Ao adicionar "inteligencia" nessas extremidades do sistema é possível diminuir a quantidade de dados que seriam transmitidos e até mesmo realizar operações inteiramente no dispositivo de armazenamento, liberando o processo que requisitou o dado para a execução de outras operações.

1.2 Objetivos

Diante da crescente necessidade por contenção de custos de comunicação em sistemas distribuídos e da evolução dos sistemas heterogêneos, este trabalho tem como objetivo implementar um disco inteligente, isto é, um disco com capacidades de processamento, em uma plataforma FPGA que realizará reconhecimento facial usando redes neurais sem peso. Com isso, pretende-se reduzir o custo de comunicação entre o disco e seus usuários, de forma que o reconhecimento facial possa ser executado in-situ, isto é, próximo de onde os dados estão armazenados. Este trabalho apresenta resultados experimentais de desempenho, custo de área do circuito, consumo de energia e custo de comunicação.

Além disso, este trabalho tem como objetivo a implementação de extensões da biblioteca de programação Dataflow Sucuri [2] em Python. A biblioteca procura trazer as facilidades da programação Dataflow para a programação de alto nível em Python, onde o programador deve montar um grafo de dependências a partir da sua aplicação, onde blocos seriam equivalentes a nós e as dependências seriam as arestas do grafo. A Sucuri também possui suporte para execução em `clusters` onde é criada uma abstração, permitindo a utilização do mesmo programa que seria executado

em apenas uma máquina para um ambiente de multiprocessamento. O modelo de programação Dataflow faz muito sentido quando aplicado ao processamento in-situ, de forma que um nó do grafo pode ser facilmente instanciado para execução na FPGA.

1.3 Contribuições

O uso de redes neurais para o reconhecimento facial é um exemplo de aplicação que pode ser implementada na FPGA como uma especialização de hardware (acelerador). Redes neurais são um conjunto de técnicas computacionais que simulam as conexões feitas pelos neurônios humanos para conseguir reconhecer padrões em diversos tipos de dados: faces, objetos, voz, etc. As redes neurais sem peso são um tipo importante de rede neural [8]. Estas redes são baseadas em memórias de acesso randômica (RAMs) e são reconhecidas por sua simplicidade e agilidade. O reconhecimento facial é muito utilizado em redes sociais, onde a identificação de rostos em fotos postadas pelos seus usuários é frequente. Algumas redes sociais são globalmente famosas, possuindo milhares de usuários em todo o mundo, o que gera uma quantidade massiva de dados em seus bancos de dados. Este tipo de cenário é ideal para avaliar as características do processamento in-situ. Com o intuito de implementar uma rede neural sem peso em um contexto de redes sociais e utilizando o reconhecimento facial em um dispositivo FPGA, as contribuições deste trabalho são:

- Implementação de um IP (*Intelectual Property*) da rede neural sem peso *Wisard* em HLS (*High Level Synthesis*). Para este fim foi usado o compilador Vivado HLS da Xilinx, que produz uma arquitetura RTL em VHDL ou Verilog a partir de uma especificação de um programa em linguagem de alto nível (C/C++), que neste caso é a rede neural sem peso Wisard. O IP utiliza a versão 4 do protocolo AXI (*Advanced eXtensible Interface*) para comunicação com outros componentes do sistema.

- Implementação da arquitetura do disco inteligente na plataforma SoC FPGA Xilinx ZC706, modelo XC7Z045-FFG900–2. O sistema do disco inteligente consiste de um processador ARM, conectado ao IP de rede neural implementando em FPGA. O ARM é responsável pela comunicação ethernet e pelo acesso ao cartão SD, onde os dados são armazenados. O sistema operacional FreeRTOS é usado para facilitar a programação da comunicação ethernet, permitindo o uso da API Sockets.

- Integração do dispositivo FPGA conectado na rede com o modelo Dataflow através da biblioteca Sucuri. Para tanto, novas classes e funcionalidades foram

incluídas para permitir que um nó FPGA seja instanciado e executado para fins de comunicação, treinamento e teste da rede neural.

- Realização de testes variando latência de rede para avaliar a partir de que ponto a movimentação constante de dados se torna um fator depreciativo sendo melhor utilizar processamento in-situ dos dados.

1.4 Organização

Os capítulos desta dissertação estão organizados da seguinte forma:

O **Capítulo 2** contém a fundamentação teórica onde serão abordados todos os aspectos necessários para a compreensão deste trabalho. Detalhes sobre a rede neural escolhida, arquitetura e o processo de desenvolvimento de um hardware em FPGAs e também detalhes sobre o Modelo Dataflow e a biblioteca Sucuri.

O **Capítulo 3** apresenta detalhes da implementação de todo os sistema co-processador, incluindo as dificuldades e detalhes da implementação do co-processador neural, o uso do processador ARM para facilitar a comunicação em rede através da interface Ethernet e detalhes sobre a implementação de nós especializados para a Sucuri.

O **Capítulo 4** possui detalhes sobre o caso de teste utilizado como experimentos, as especificações sobre a máquina utiliza nos testes e também uma discussão sobre os resultados obtidos nos mesmos.

O **Capítulo 5** apresenta a conclusão do trabalho e sugestões de trabalhos futuros, indicando o que pode ser feito adiante para otimizar os resultados e o design do sistema como um todo..

Capítulo 2

Fundamentação Teórica

Este capítulo descreve os tópicos necessários para compreensão desta dissertação. Serão abordados aspectos sobre as arquiteturas de FPGAs, detalhes do modelo Dataflow, da rede neural sem peso e da aplicação de reconhecimento facial com foco em redes sociais.

2.1 Arquiteturas Reconfiguráveis

As FPGAs são circuitos integrados formados por blocos lógicos reconfiguráveis e que se comunicam por interconexões também reconfiguráveis. Diferente dos circuitos dedicados a uma aplicação (*Application-Specific Integrated Circuit - ASIC*), as FPGAs permitem que os usuários reprogramem seus componentes reconfiguráveis para melhor atenderem aos requisitos de desempenho e consumo de energia de uma dada aplicação ou conjunto de aplicações. A frequência de operação de um circuito mapeado em FPGA é geralmente menor que a de um ASIC. No entanto, a grande quantidade de recursos reconfiguráveis disponíveis permitem a execução de várias operações em paralelo, a um custo menor de consumo de energia.

A Figura 2.1 apresenta a arquitetura geral de uma FPGA. Os blocos lógicos (CLBs) podem ser configurados para constituir diferentes tipos de sistemas digitais. A rede de interconexão, também disposta na imagem, consiste de conexões e *switches* reprogramáveis. Por fim as portas de entrada e saída conectam a FPGA com o mundo externo, permitindo, por exemplo, a troca de dados com dispositivos de memória: RAMs, SSDs, etc.

O fluxo de desenvolvimento de projetos em FPGAs costuma ser muito mais simples e baratas se comparado ao fluxo de desenvolvimento de ASICs, possibilitando que soluções sejam disponibilizadas ao mercado em menor tempo (*time-to-market*). Tal fluxo de desenvolvimento costuma ser divido em etapas que consistem desde a descrição do circuito até sua execução dentro de um sistema com outros dispositivos.

Normalmente, a primeira etapa é a especificação do circuito, *e.g.* co-processador,

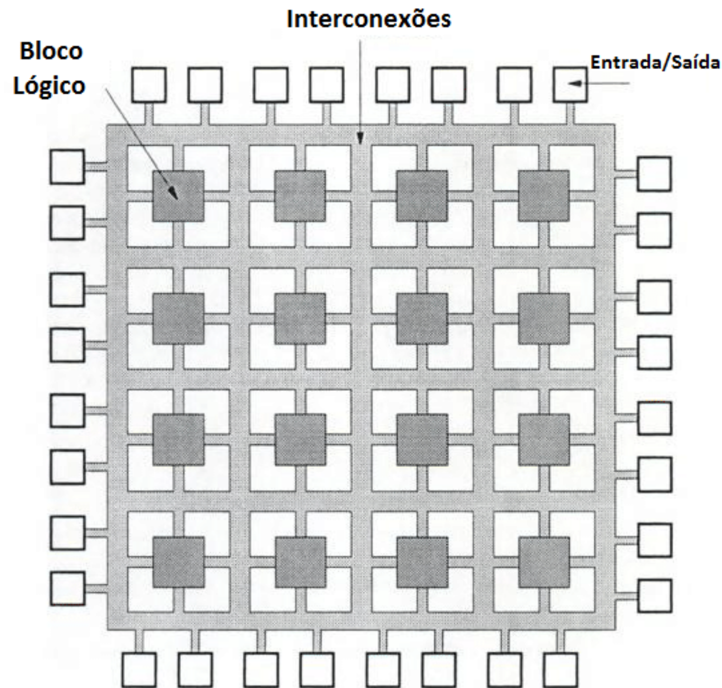


Figura 2.1: Representação da arquitetura FPGA divida em blocos lógicos, uma rede de interconexão e portas de entrada e saída.

em nível RTL, usando uma linguagem de descrição de hardware, como VHDL ou Verilog. Desta forma, o comportamento do circuito é descrito através de estruturas bem definidas e conhecidas. Também, já é possível realizar teste funcionais, capazes de verificar se os componentes estão funcionando corretamente antes de prosseguir o fluxo de desenvolvimento. Isso é feito através da observação dos sinais de saída produzidos pelo estímulo dos sinais de entrada do circuito escolhidos pelo usuário.

O próximo passo no fluxo de desenvolvimento é a síntese do circuito, que irá mapear a arquitetura RTL para uma representação inicial no nível de portas lógicas [9] e componentes básicos da FPGA (BlockRAMs, DSPs, etc.). Nesta etapa, algumas otimizações pontuais podem ser feitas pelo sintetizador a fim de, por exemplo, reduzir o consumo de recursos. Após a síntese, é possível obter uma estimativa inicial de consumo de recursos da FPGA, consumo de energia, atraso do caminho crítico, etc.

A última etapa é a de implementação e roteamento [10]. Nesta fase, o compilador irá buscar diferentes formas de mapear o circuito sintetizado nos recursos físicos da FPGA. O mesmo será feito para as interconexões entre os blocos lógicos. A quantidade de combinações é exorbitante então encontrar a melhor implementação não é possível. Por conta disso, a maioria das ferramentas utilizam de algoritmos heurísticos para encontrar uma solução viável em um tempo adequado. Por fim, o *bitstream* do projeto completo será gerado para programação (configuração) da

FPGA através de um cabo JTAG (*Joint Test Action Group*) ou até mesmo de um cartão de memória.

2.2 High-Level Synthesis

Recentemente, compiladores HLS (*High-Level Synthesis*) têm se destacado cada vez mais no mercado de FPGAs e de circuitos integrados em geral, pois permitem que um código descrito em C/C++ possa ser convertido para uma linguagem de descrição de hardware automaticamente, facilitando a programação, ainda que com algumas poucas limitações.

Embora o fluxo de desenvolvimento em FPGAs seja muito mais simples que o de ASICs, a especificação em nível RTL não é tão pratica quanto a programação estruturada proporcionada por linguagens de alto nível, o que torna esta etapa uma das mais demoradas no fluxo de desenvolvimento. Em nível RTL, não existem chamadas a um sistema operacional ou conjuntos extensos de bibliotecas que auxiliam a programação de uma aplicação complexa. Sendo assim, os compiladores HLS possibilitam a descrição de uma aplicação, geralmente na linguagem C/C++, facilitando a transformação da mesma em uma arquitetura RTL descrita em VHDL ou Verilog. Os compiladores HLS normalmente incluem suporte a um conjunto de bibliotecas otimizadas para manipulação de imagens, operações em ponto flutuante, barramentos de comunicação, etc. Com isso, além da redução drástica no tempo de especificação, desenvolvimento e prototipação, consequentemente há também uma redução drástica no *time-to-market* do produto final.

Portanto, o objetivo principal de um compilador HLS é fornecer ao desenvolvedor uma versão limitada do C/C++ que, além de permitir a programação habitual, fornece uma gama de bibliotecas que são comumente utilizadas em aplicações complexas de processamento de vídeo, áudio, etc. Isto pode parecer simples do ponto de vista do programador C/C++, porém costuma ser muito trabalhoso do ponto de vista do programador VHDL/Verilog. Além de tudo, muitos compiladores HLS auxiliam na implementação de protocolos de comunicação complexos. O protocolo AXI costuma ser o mais utilizado em arquiteturas SoC (*System-on-Chip*) baseadas em ARM, oferecendo três principais variações: AXI-Full, AXI-Lite e AXI-Stream.

O AXI-Lite é um protocolo simples, limitado ao envio e recebimento de 32 ou 64 bits por transação, dependendo do tamanho do barramento. Este protocolo é geralmente utilizado para enviar e receber sinais de controle ou registradores de estado. Já o protocolo AXI-Stream implementa um modelo de comunicação ponto-a-ponto, enviando e recebendo dados em fluxo com o auxílio de um DMA (*Direct Memory Acess*). O protocolo AXI-Full, por fim, permite o envio e o recebimento de rajadas de até 256 dados de 32 bits entre dispositivos mapeados em memória.

2.3 O Modelo Dataflow

O modelo Dataflow [11, 12] surgiu na década de 70 como uma alternativa ao modelo tradicional de Von Neumann, cujo desempenho é fortemente limitado pela natureza intrinsecamente sequencial do contador de programa, ainda que não existam dependências entre algumas das instruções. Uma arquitetura Dataflow, por outro lado, tem a execução do programa guiada pelo fluxo de dados, o que significa que para uma operação ser executada basta que seus operandos já estejam prontos e que exista uma unidade funcional disponível para sua execução. Logo, com vários operandos prontos e unidades disponíveis, o paralelismo flui naturalmente.

A forma de se representar a linguagem base do modelo Dataflow é através de um grafo de dependências [13]. Os vértices, também chamados de nós, são equivalentes a operações ou tarefas a serem executadas, enquanto as arestas preenchem as dependências entre os vértices, guiando quando a operação ou tarefa poderá ser executada. Tal execução produz resultados que podem ser levados até outros nós, caracterizando esta aresta como de saída para o nó que foi executado e de entrada para o nó a ser executado. Em geral, nós possuem uma aresta de saída e podem ter várias arestas de entrada. A Figura 2.2 representa um trecho de código no quadro A, com as instruções e declarações. O seu grafo de dependência correspondente está representado no quadro B. É possível visualizar que a operação de soma e multiplicação são independentes entre si. Então, desde que haja recursos suficientes, elas poderão ser executadas em paralelo, enquanto a operação de subtração deve esperar os resultados correspondentes para ser inicializada.

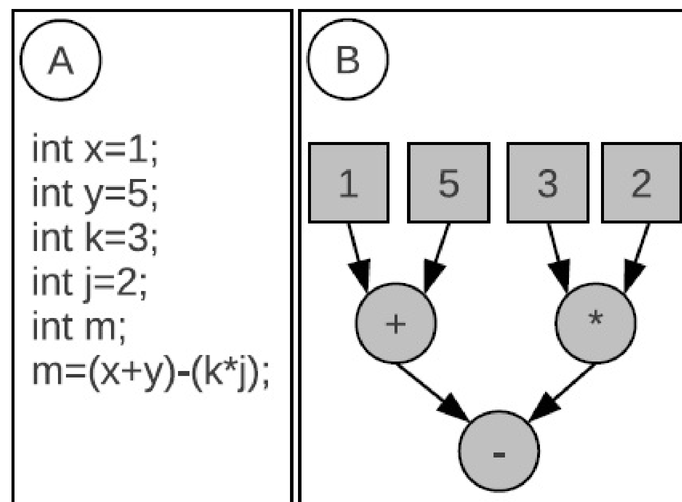


Figura 2.2: Exemplo de um programa dataflow (A) e seu grafo de dependência correspondente (B), reproduzido de [1]

Embora o modelo Dataflow pareça solucionar a maioria dos problemas em relação à identificação de paralelismo, na prática existem dificuldades em sua im-

plementação. Uma delas é que o modelo assume que se as instruções não possuem dependências, elas devem ser executas em paralelo. No entanto, isto pode implicar em um número impraticável de unidades funcionais necessárias para execução do programa dataflow. Outra dificuldade esta no fato de que na teoria a quantidade de dados passante pelas arestas devem ter tamanhos ilimitados, criando *buffers* e filas com tamanhos também impraticáveis. Por conta destes problemas, uma possível solução é a utilização híbrida de ambos os modelos como, por exemplo, usando bibliotecas que emulam o ambiente Dataflow em cima da arquitetura já estabelecida de Von Neumann. Neste trabalho utilizaremos exatamente este ambiente híbrido, através da biblioteca Sucuri [2].

A Sucuri é uma biblioteca minimalista desenvolvida em Python que permite a programação Dataflow em alto nível. A biblioteca também permite que o mesmo código executado em uma máquina possa ser utilizado para execução em um aglomerado (*cluster*) de máquinas, através do MPI (*Message Passing Interface*). A Sucuri é dividida em quatro componentes principais: `Scheduler, Worker, Node e Graph`.

A classe `Node` representa os nós do grafo. A cada nó é associado uma função declarada pelo programador e que será executada como uma tarefa. Ao instanciar o nó o programador deve indicar o nome da função, o número de arestas de entrada e conectar suas arestas de saída utilizando a função `add_edge`. A classe `Graph` funciona como a base do problema, consistindo no grafo de dependências da aplicação. Esta classe é utilizada como um contêiner onde todos os nós declarados devem ser adicionados. O `Worker` é equivalente a um processo que irá executar diversas tarefas durante seu ciclo de vida. O número de processos é definido quando o escalonador é declarado. Quando o `Worker` está ocioso, ele enviará um marcador indicando que pode receber novas tarefas. Quando o marcador é recebido pelo escalonador, uma tarefa é enviada e a função do nó correspondente é executada. Também é possível forçar com que algum grupo de `Workers` sejam os únicos que podem executar um certo nó, o que é bastante útil em um ambiente de clusters.

A classe `Scheduler` é responsável por gerenciar os operandos, operações e escalonar os processos ociosos em operações prontas para a execução. O `Scheduler` recebe os operandos enviados pelas tarefas já finalizadas que possuem retorno. Isto significa que seus nós correspondentes possuíam arestas de saída. Os operandos são armazenados em uma unidade especial chamada `Matching Unit`, que é responsável por identificar quando um nó já tem seus operandos disponíveis. Em caso positivo, uma tarefa é criada e designada para algum `Worker` disponível ou colocada em uma fila de espera..

A Figura 2.3 apresenta uma visão geral da arquitetura interna da Sucuri e como seus componentes são organizados quando utiliza-se o MPI para comunicação entre processos em máquinas remotas. É importante frisar que todos os componentes

são replicados para todas as máquinas com exceção do escalonador. Isto se dá por conta do escalonador ser centralizado, ou seja, todas as tarefas são designadas e transmitidas por uma máquina, enquanto os escalonadores das demais funcionam apenas como retransmissores de informação.

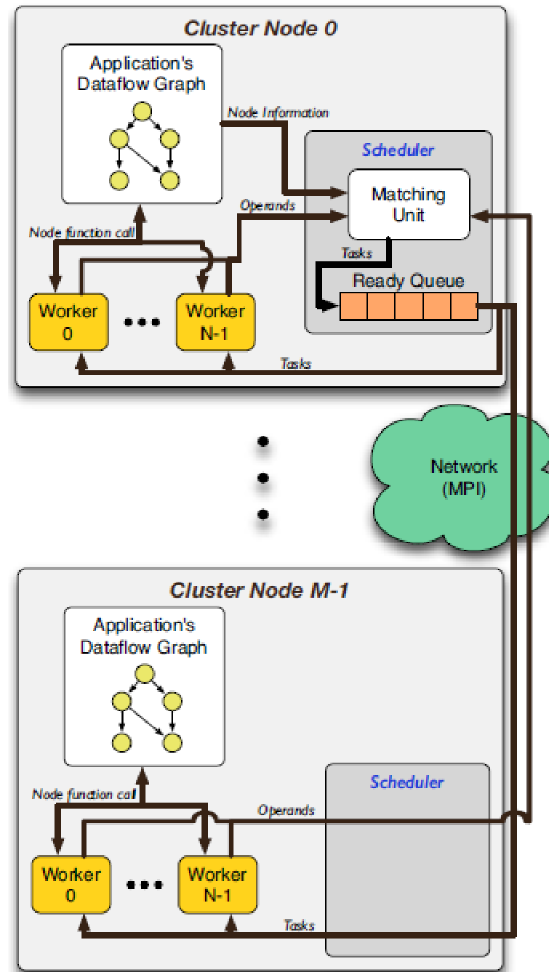


Figura 2.3: Componentes internos da arquitetura da Sucuri. Reproduzido de: [2]

Uma característica importante da Sucuri é que ela segue os princípios de Programação Orientada a Objetos (POO). Logo, é possível criar um novo tipo de nó que herda os atributos e funcionalidades da classe original `Node` a fim de criar uma biblioteca de nós específicas para uma dada aplicação. Esta funcionalidade permite que funções muito utilizadas não precisem ser declaradas e instanciadas em um nó toda vez que um programa novo for feito. Essa funcionalidade já se provou muito útil para adesão de novos dispositivos como GPUs [14].

Um exemplo de especialização que pode ser encontrado na biblioteca é o chamado `Feeder`, disposto na Figura 2.4. O objetivo do nó `Feeder` é abolir a necessidade de criar uma função que não teria arestas de entrada, estas que geralmente são utilizadas para inicializar o grafo. Para a aplicação da Figura 2.4 seria necessária a criação de duas funções que apenas retornariam os valores utilizados pela tarefa seguinte.

Porém, com esta especialização, foi possível apenas instanciar dois `Feeders` e passar os valores de retorno como parâmetros. Outra especialização muito utilizada é o chamada `Source` que recebe um ponteiro para arquivo e envia linha por linha do arquivo com uma tag especial. Esta tag especial permite execução de varias instancias paralelas do nó seguinte. Uma técnica similar a execução de varias interações independentes de um loop em paralelo.

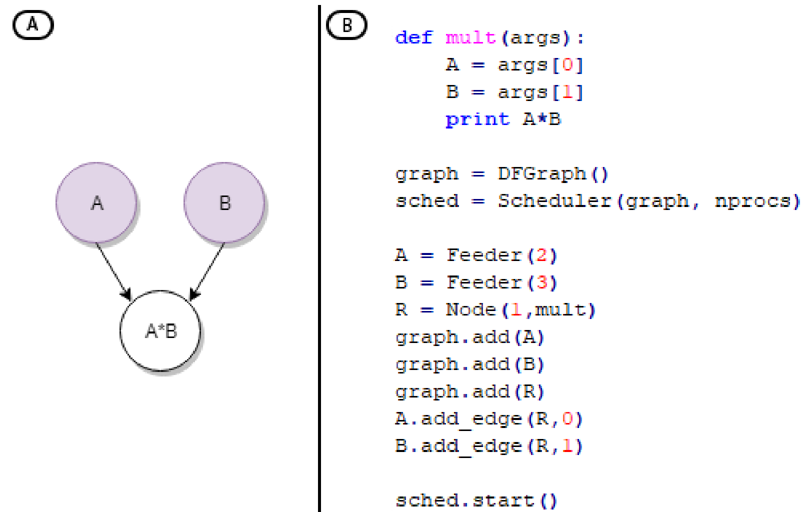


Figura 2.4: Exemplo de especialização da classe `Node` com o nó `Feeder`

2.4 Reconhecimento Facial

O reconhecimento facial sempre teve um papel muito importante no cotidiano humano, seja para relembrar do rosto de familiares, amigos ou até mesmo ajudar em investigações policiais. Com o avanço da tecnologia foi possível a criação de grandes banco de dados com rostos de diversas pessoas do mundo. No entanto, foi necessária a criação de ferramentas automáticas que podem fazer o trabalho de busca e identificação de faces de forma rápida e precisa.

As redes neurais são uma classe de algoritmos de aprendizado de máquina muito usados em aplicações de visão computacional e reconhecimento de voz. Elas simulam as conexões feitas pelos neurônios no cérebro, procurando reproduzir sua capacidade de aprendizado. Cada neurônio individual recebe valores de entrada, aplica um tipo de processamento e produz um valor de saída [15]. Em geral, as redes neurais podem ser dividas em dois tipos: sistemas analógicos que utilizam redes com peso e sistemas digitais que utilizam redes sem peso.

As redes baseadas em sistemas analógicos possuem um parâmetro de peso que aumenta ou diminui a força da resposta dada pelo neurônio. Para realizar o treinamento em tais sistemas é necessário um conjunto de imagem chamadas de positivas,

que constituem padrões que se quer reconhecer. No caso de reconhecimento de objetos, imagens positivas seriam todos os objetos que a rede deve reconhecer. Já o conjunto de imagens chamado de negativos seriam os objetos que não deveriam ser reconhecidos, retornando uma resposta negativa. Conforme os padrões são treinadas os pesos são ajustados, favorecendo certos caminhos entre os neurônios para que dados de entradas parecidos procurem passar por caminhos parecidos, retornando a resposta correta. Redes neurais sem peso são baseadas em memórias de acesso randômico (RAMs) e, diferente das redes neurais com peso, são muito mais simples e velozes. A força de um sinal de entrada depende de qual altura na árvore dendrítica a conexão de entrada sináptica é colocada, bastante semelhante ao modo como a decodificação do endereço da RAM é executada. Por serem baseadas em um dispositivo de hardware já conhecido, elas se tornam o alvo perfeito para serem programadas em um hardware reconfigurável.

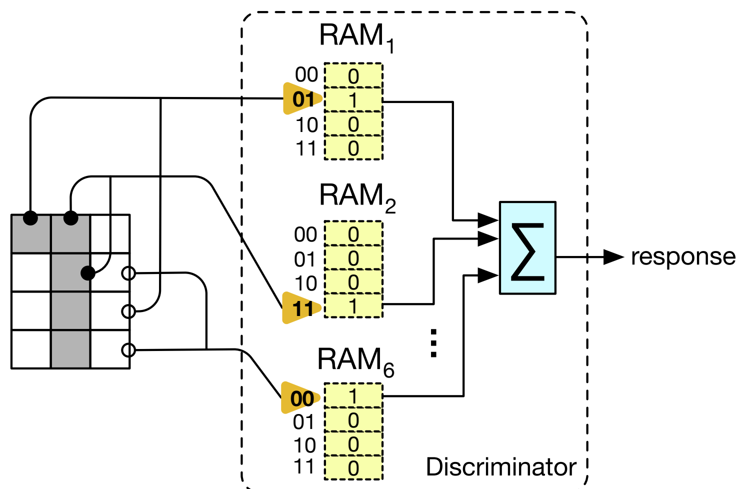


Figura 2.5: Mapeamento de uma imagem preto e branco 4x3 na Wisard, onde cada *pixel* é um quadrado preto com valor 1 ou branco com valor zero

A Wisard [8] é uma rede neural sem peso constituída por um conjunto de discriminadores, cada qual formado por um conjunto de x RAMs de até n entradas e um somador. O padrão de entrada de cada posição da RAM está associado diretamente a um mapeamento pseudo-aleatório.

A Figura 2.5 mostra como uma imagem preto e branca de 4×3 *pixels* pode ser mapeada em um conjunto de RAMs. Todos os *pixels* são selecionados aleatoriamente e mapeados, também aleatoriamente, em uma das RAMs disponíveis. O endereço que será acessado na RAM selecionada se dá pelos valores concatenados dos *pixels* obtidos. Como a imagem é preto e branco, os valores dos pixels variam de 0 (branco) e 1 (preto). A quantidade de *pixels* selecionadas por vez é intrisicamente relacionado ao tamanho de cada RAM. No caso da figura em questão são necessários dois *pixels* por vez para formar um endereço binário correspondente e preencher as 5 RAMs,

visto que as RAMs possuem até 4 linhas de endereçamento (00, 01, 10 e 11). Para as próximas imagens mapeadas o procedimento continua o mesmo, porém os *pixels* acessados e em qual RAMs são mapeados deve se manter o mesmo, por isso é chamado de mapeamento pseudo-aleatório.

Como a maioria das redes neurais, a Wisard necessita de treinamento para se adaptar a padrões desconhecidos. Para inicializar uma rotina de treinamento é necessário que todos os valores em cada posição das RAMs estejam zerados. Cada imagem utilizada para treino deve possuir um tamanho de $n \times x$ bits de forma que seja possível preencher todas as RAMs de cada discriminador em alguma posição. Na fase de treino os endereços obtidos através das imagens são acessados e atualizados com o valor 1. Em geral cada discriminador é especializado em um tipo de padrão, sendo sempre treinado na mesma classe de *inputs*. É importante notar que o somador não é utilizado no treinamento, sendo exclusivo para os testes, também conhecidos como previsões.

Quando treinando uma imagem é possível que o endereço gerado pelo mapeamento já tenha o valor 1 escrito por conta de algum outro conjunto de *pixels* de outra imagem que foi mapeado no mesmo lugar, isto é chamado de conflito. No Wisard clássico ao testar uma imagem este endereço não terá um valor especial que reflita para o somador que diversas imagens diferentes já tentaram escrever neste endereço. Para refletir que ocorreu algum conflito na posição da RAM existem técnicas como o `bleaching` [16, 17] que aumenta o valor escrito para mais do que apenas 1. Um somador especial irá levar em conta estes conflitos e pode melhorar substancialmente a qualidade da resposta.

Quando um padrão precisa ser reconhecido a Wisard irá executar uma rotina de teste. Este padrão será mapeado nas RAMs da mesma forma que ocorre em um treinamento, porém os valores serão apenas acessados. O valor de cada posição acessada será somado pelo somador que irá gerar uma resposta r de discriminador. Cada resposta dada indica o quão similar o discriminador em questão é do padrão testado. É importante notar que ao testar uma imagem que foi utilizada no treinamento, o somador retornaria o maior valor r possível, dado que o mapeamento do treino deve mapear os mesmos *pixels* nas mesmas posições. Após todos os discriminadores testados, os valores de similaridade são comparados e avaliados de acordo com um valor de confiança relativa, retornando a provável classe no qual a imagem testada pertence. Caso a rede esteja sendo utilizada para reconhecimento facial e cada discriminador tenha sido treinado em uma pessoa, o discriminador com maior resposta indicaria a pessoa mais provável de estar na imagem testada.

2.5 Trabalhos Relacionados

No artigo descrito em [18] um sistema similar ao deste trabalho é construído. A biblioteca Sucuri também é utilizada para emular um ambiente Dataflow. No artigo uma versão modificada da Sucuri descrita em [19] faz o uso de um escalonador distribuído e é utilizada para avaliar, em aplicações simples, quais os melhores momentos para processar os dados de forma in situ levando em consideração as condições da rede. O dispositivo que simulou um disco inteligente é chamado de Parallella. As aplicações utilizam um sistema operacional Linux e o microprocessador ARM da placa Parallella para executar o escalonador da Sucuri, diferentemente do trabalho atual onde o processador ARM apenas irá auxiliar o co-processador implementado na FPGA repassando as informações enviadas pelo *host*.

Aplicações voltados para o meio cientifico costumam produzir e consumir uma quantidade massiva de dados. Porém a distancia de desempenho entre os dispositivos na hierarquia de memória ainda é muito grande, sendo necessário recorrer a técnicas que agilizem o acesso ao disco como a indexação de dados, que já é muito utilizada com sucesso em banco de dados. O problema da indexação é que todo o processo de construção é muito custoso. Em [20] é proposto a utilização de processamento in situ para indexar os dados no caminho entre a memória e o disco, agilizando o processo de criação em até 10 vezes e diminuindo o tempo de leitura no disco em até 200 vezes.

Simulações cientificas de grande escala também possuem uma grande produção de dados que pode chegar até aos terabytes em cada execução. Por conta destes tipos de simulações produzirem uma quantidade de dados massiva é necessário diminuir a carga em que se acessa os dispositivos de entrada e saída. Por conta disso, os dados acabam sendo apenas recuperados entre intervalos de tempo. O problema deste tipo de abordagem para compressão é que existe muita perda de informação. In-situ Sort-And-B-spline Error-bounded Lossy Abatement (ISABELA) [21] é um método especializado na compressão de dados entre estes intervalos para simulações que possuem uma grande aleatoriedade e ruídos em seus dados, permitindo que o usuário defina a precisão que a dado compresso deve possuir. O método divide os dados em janelas aplicando uma ordenação seguido de um B-spline. Os erros relativos entre o modelo criado com o B-spline e o dado original são comparados. No final apenas os índices permutados dos blocos ordenados e os erros relativos são guardados no dispositivo.

Em [22] é introduzido um sistema chamado de ADIOS que procura resolver parcialmente os problemas envolvendo o gargalo gerado entre dispositivos de entrada e saída em sistemas `exascale` através do monitoramento e gerenciamento de dados e do processamento in situ. O sistema oferece uma arquitetura orientada a serviço

(ASO) que facilita a programação de *plugins*, permitindo a criação de aplicações in situ customizadas que podem facilitar a analise a visualização de dados. No artigo também é apresentado um formato de arquivo chamado de ADIOS-BP otimizados para aplicações cientificas que possuem um paralelismo massivo.

Em [23] são discutidos aspectos e problemas sobre a implementação de redes neurais com peso em FPGAs. Embora mais adequadas do que ASICs e DSPs por terem maior flexibilidade, os cálculos em neurônios com diversas entradas geram números e operações de ponto flutuante que custam muito em recursos da FPGA. Por conta disso, a maioria dos designs optam por limitar o paralelismo e executar neurônios sequencialmente. Implementações de neurônios que usam funções lineares e não lineares são avaliadas e os principais aspectos que influenciam a relação velocidade e consumo de recursos são explicitados.

As `Convolutional neural networks` (CNNs) são redes neurais artificias utilizadas para reconhecimento facial que se baseiam nos nervos óticos dos seres vivos. Por conta de seu padrão especifico de processamento CNNs não costumam ter boa performance em processadores de proposito geral, sendo necessário utilizar aceleradores. Embora FPGAs sejam os aceleradores preferidos para sua implementação por conta da flexibilidade e consumo de energia, a relação do design entre a estrutura lógica e de memória pode trazer até 90% de diferença de performance entre implementações. Em [24] a vazão entre computação de dados e resposta da memória são analisados. Um algorítimo que procura otimizar o design de um acelerador CNN também é apresentado. Resultados preliminares comparam a implementação de uma CNN utilizando o algorítimo desenvolvido e outras implementações da mesma rede em outros dispositivos FPGA. Cerca de um *throughput* de 61,62 GFLOPS e um `speedup` de 3,62 se comparado a outras implementações de CNNs em FPGAs são reportados.

Em [25] uma rede neural com peso é desenvolvida em uma FPGA da Xilinx onde técnicas de *backpropagation* e *pipelining* são implementadas permitindo treinamento paralelo [26]. A técnica de *backpropagation* é dividia em 3 fases que são executadas a cada padrão que é inserido na rede. Neste algorítimo é possível executar todas as fases simultaneamente onde cada una executa um padrão inserido em outro momento. A rede é implementada através do VHDL em um dispositivo FPGA, onde as sinapses dos neurônios foram implementadas através da ferramenta *High-Level Synthesis*. Resultados preliminares comparam a versão com e sem *pipelining* conseguindo bons resultados.

Capítulo 3

O Disco Inteligente

Este capítulo abrange a implementação de todo o sistema do disco inteligente em hardware e em software. Sua inteligência se deve a um co-processador de redes neurais sem peso (Wisard) implementando na lógica reconfigurável da FPGA, cujo objetivo é possibilitar a execução *in-situ* de operações de treinamento e teste de imagens armazenadas no cartão de memória da placa. Com isto, pretende-se reduzir o custo de comunicação com máquinas remotas que façam uso do disco para reconhecimento facial em um cenário de redes sociais.

3.1 Co-Processador Neural

O Co-processador Neural foi desenvolvido com base na rede neural sem peso Wisard [8] sem *bleaching*, apresentada na Seção 2.4. Contudo, ele permite o uso de fotos em tons de cinza, diferente da rede neural descrita na seção supracitada, a fim de melhorar a taxa de acertos da rede. O trabalho principal do co-processador é receber uma matriz de endereços válidos para acessar suas memórias em seus discriminadores e escrever valores no caso de treino, ou somá-los no caso de teste. Podemos perceber que essa implementação independe da forma do dado de entrada, desde que este tenha uma conversão válida para representar os endereços. A propriedade intelectual (IP) do co-processador neural se deu através do compilador Vivado HLS da Xilinx, que utilizou o *High Level Synthesis* para produzir uma arquitetura RTL em VHDL a partir da especificação em linguagem C/C++. Um IP pode ser interpretado como uma biblioteca, que pode ser patenteada e é utilizada pelo usuário como uma caixa preta.O código fonte HLS da rede neural Wisard com apenas dois discriminadores está representado no Algoritmo 3.1. O Apêndice A apresenta a especificação completa da rede neural, ou seja, com quatro discriminadores.

Algoritmo 3.1: Implementação da Wisard com dois discriminadores usando HLS.

```
1 #define IMG_SIZE 8000
2
3 static bool disc0[IMG_SIZE * 64];
```

```
4  static bool disc1[IMG_SIZE * 64];
5
6  void wisard(bool mode, int disc_id, volatile int *img_addr, int *res) {
7      #pragma HLS INTERFACE m_axi depth=8000 port=img_addr offset=slave bundle=image
8      #pragma HLS INTERFACE s_axilite port=img_addr bundle=control
9      #pragma HLS INTERFACE s_axilite port=mode bundle=control
10     #pragma HLS INTERFACE s_axilite port=disc_id bundle=control
11     #pragma HLS INTERFACE s_axilite port=res bundle=control
12     #pragma HLS INTERFACE s_axilite port=return bundle=control
13
14   int img[IMG_SIZE];
15
16   memcpy(img, (const int*) img_addr, IMG_SIZE * sizeof(int));
17
18   int counter0 = 0;
19   int counter1 = 0;
20   int index = 0;
21
22   if (mode == 0) //treino
23     {
24     switch (disc_id) {
25     case 0:
26       for (int i = 0; i < IMG_SIZE; i++) {
27         index = 64 * i + img[i];
28         disc0[index] = 1;
29       }
30       *res = -1;
31       break;
32     case 1:
33       for (int i = 0; i < IMG_SIZE; i++) {
34         index = 64 * i + img[i];
35         disc1[index] = 1;
36       }
37       *res = -2;
38       break;
39     default:
40       *res = -5;
41     }
42   } else //teste
43   {
44     for (int i = 0; i < IMG_SIZE; i++) {
45       index = 64 * i + img[i];
46       counter0 = counter0 + disc0[index];
47       counter1 = counter1 + disc1[index];
48     }
49     *res = max(counter0, counter1);
50   }
51 }
```

Como os discriminadores devem ser sempre utilizados e devem manter seu estado entre chamadas, dado que não se quer perder os treinamentos anteriores, foi necessária a sua declaração como uma variável global utilizando a palavra chave `static`. A função Wisard recebe como parâmetros um vetor que representa a imagem já convertida em endereços e uma `flag` indicando se esta imagem deve inicializar uma rotina de treino ou de teste. No caso de treinamento mais um parâmetro deve ser enviado, este é a `ID` do discriminador que se deseja treinar. Os valores do discriminador correspondente serão acessados de acordo com o vetor da imagem e o treinamento realizado. No caso do teste os valores da imagem são acessados para todos os discriminadores e somados retornando a `ID` do discriminador que deu a melhor resposta.

Conforme descrito na Seção 2.2, os compiladores HLS apresentam em geral algumas limitações. Uma delas é que não existe suporte à alocação dinâmica, visto que o hardware precisaria ser reconfigurado dinamicamente para tal finalidade. Por

conta disso, o tamanho das memórias RAMs e a quantidade de discriminadores tem um tamanho fixo. Esta limitação impõe que as imagens devem sempre possuir um conjunto de endereços que abrange o mesmo tamanho. A limitação quanto a quantidade de discriminadores pode ser contornada, pois é possível especificar o máximo de discriminadores que os recursos da FPGA conseguem suportar e utilizando apenas os discriminadores devidamente treinados. Isto é semelhante à implementação de estruturas de dados circulares com uso de `arrays`, cujos limites são controlados a cada inclusão e remoção de dados da estrutura. Neste trabalho, no máximo quatro discriminadores são especificados no co-processador neural, permitindo a síntese e implementação do mesmo em FPGAs mais singelas, ou seja, com menos recursos de blocos lógicos e blocos de memória.

O compilador HLS do Vivado gera uma especificação VHDL ou Verilog em nível RTL a partir de uma hierarquia de funções descritas em C/C++, sendo a função raiz aquela que irá encapsular a funcionalidade de todas as demais funções. No caso da especificação VHDL, utilizada ao longo deste trabalho, cada função é mapeada em uma entidade, com as portas de entrada e saída mapeadas segundo os parâmetros de entrada da função e o seu retorno. A quantidade de bits definida em cada porta respeitará o tipo de dados do parâmetro e do retorno. A Figura 3.1 mostra a interface do co-processador produzido a partir da especificação HLS descrita no Algoritmo 3.1. Para a comunicação com o ARM são utilizados os protocolos de comunicação AXI4-Lite e AXI4-Full. Tais protocolos nas interfaces destacadas são automaticamente implementados através de diretivas de compilação (*pragmas*) indicadas nos parâmetros de entrada/saída da função raiz.

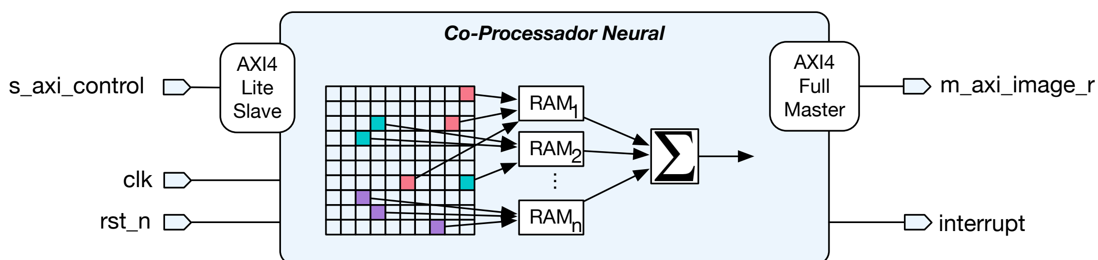


Figura 3.1: Interface do Co-Processador Neural.

3.2 Arquitetura do Sistema

O próximo passo na construção do sistema foi utilizar a ferramenta Vivado para fazer o chamado `block design`. Nesta etapa a Xilinx oferece suporte a uma ferramenta gráfica embutida dentro do próprio vivado que permite conectar diversos designs com o próprio ARM. Para realizar este feito, bastou exportar o Wisard como

um IP e conecta-lo com o IP do microprocessador ARM através de `switches` que também podem ser considerados como propriedades intelectuais. Essas propriedades intelectuais são um conjunto de designs já prontos que a Xilinx oferece como uma biblioteca a ser utilizada no `block design`, permitindo o re-uso de designs já feitos anteriormente por terceiros.

Figura 3.2: O Co-processador Wisard implementado no `Block Design do Vivado`

Na Figura 3.2 podemos visualizar como ficou o design completo da Wisard através da ferramenta. Temos o ARM, representado como ZYNQ, conectado com o co-processador neural através do protocolo AXI4, os outros componentes funcionam como multiplexadores e *switches* para auxiliar no funcionamento do barramento. Ao investigar a implementação do design na FPGA, descobrimos que ao declarar os discriminadores como vetores separados, eles foram otimizados para terem suas posições acessadas simultaneamente criando um paralelismo para qualquer tipo de acesso simultâneo feito neles.

Após definido o `Block Design`, os IPs foram sintetizados e implementados para as FPGA ZC706, Zybo e ZedBoard. Por consumir muita memória dado que os discriminadores são de tamanho elevado as FPGAs Zybo e ZedBoard não tiveram recursos suficientes para comportar o co-processador, deixando apenas a placa ZC706 como opção definitiva. É valido citar que os valores do consumo de recursos da placa ZC706 serão explicitados no capítulo 4.

O passo final é gerar o `bitstream` que foi utilizado para programar a FPGA através de um cabo J-TAG. O projeto do Vivado foi então exportado para outra ferramenta da Xilinx chamada xSDK que permite programar o ARM através da linguagem C/C++, e utilizar o projeto exportado para executar o co-processador.

3.3 Comunicação e Cartão de Memória

O objetivo do microprocessador ARM no sistema é mediar a comunicação entre um *host* externo que irá utilizar uma linguagem de alto nível e o co-processador neural. Denominamos esta parte do sistema de `File Server` dado que será um servidor que gerenciara os arquivos do cartão de memória. Esta comunicação será feita ponto

a ponto através de um cabo Ethernet través de um protocolo de comunicação já pré-definido. Este tipo de mediação foi realizada para facilitar a implementação da comunicação de rede, dado que para realizar uma comunicação direta com o co-processador neural seria necessária a programação da entrada de rede e de todo o protocolo de comunicação TCP em linguagem RTL. Quando utilizamos o microprocessador ARM, a ferramenta xSDK da Xilinx possui suporte a um conjunto de bibliotecas que procura facilitar diversas funcionalidades, como as de comunicação. Uma das bibliotecas disponíveis é a chamada lwIP(*Lightweight TCP/IP stack*) [27].

O lwIP é uma implementação mais simples e leve do protocolo TCP/IP que permite utilizar todas as vantagens que uma conexão TCP iria proporcionar diretamente no ARM, sem o auxilio de um sistema operacional. Porém ainda é necessário controlar pacotes e janelas explicitamente através de ponteiros e estruturas. Por conta disso, foi utilizado o sistema operacional FreeRtOS disponibilizado também no xSDK. O FreeRtOS é bem leve e limitado e será apenas utilizado para auxiliar o uso do lwIP através de uma abstração de `sockets` permitindo uma comunicação de alto nível com um *host*.

É importante citar que a combinação FreeRtos e LwIP permite a execução de um servidor que irá criar uma nova *thread* e *socket* para cada comunicação aberta com o dispositivo. O problema é que existe um limite baixo na quantidade de dados que cada *thread* pode possuir por conta do FreeRTOS. Esta limitação obrigou a utilização de uma variável global que pode ser acessada por todas as *thread*, tornando inviável o uso de *thread* simultâneas no `File Server`. Outra limitação encontrada na combinação FreeRtos e LwIP foi relacionada ao recebimento de dados por parte do *socket*, dado que a função implementada não possuía uma de garantia que iria ter uma espera bloqueante até todos os bytes serem recebidos, obrigando a programação de laços extras que continuam a checar o *buffer* por novas informações até chegada completa dos dados.

Além da comunicação com o ambiente externo o microprocessador ARM ainda deve se comunicar com o cartão de memória inserido na placa para gravar ou ler imagens. Para realizar estas operações foi utilizada uma outra biblioteca disponibilizada pelo xSDK chamada `Xilffs` que permite montar e acessar um disco formatado em `Fat32` através de uma árvore de diretórios como o de costume em sistemas operacionais tradicionais.

Para este trabalho foram desenvolvidas duas versões que utilizam o co-processador neural no `File Server`. A primeira foi feita especialmente para o caso de testes de processamento in-situ dispostos no Capítulo 4, enquanto a outra foi feita para ser utilizada de forma genérica em trabalhos futuros.

A primeira versão ao iniciar o dispositivo irá buscar todos os arquivos de treino em um diretório especifico e todos os discriminadores serão treinados. O próximo

passo do `File Server` será iniciar o servidor LwIP que irá esperar conexões. Quando uma conexão é feita, uma *thread* e *socket* TCP são criados. O servidor deve receber um endereço de uma imagem que já está gravada no cartão SD. Esta imagem será buscada e testada nos discriminadores já treinados, o retorno para o *host* será a `id` do discriminador com a melhor resposta.

Na segunda versão ao iniciar a placa o servidor LwIP também é inicializado esperando por conexões. Quando uma conexão é feita uma *thread* e *socket* TCP são criados, estes devem receber primeiramente a flag indicando se deseja treinar ou testar os discriminadores e a imagem já convertida para o uso do Wisard que será treinada ou testada. Caso a `flag` indique que seja um treinamento a segunda mensagem deve conter a `id` do discriminador e uma mensagem de confirmação será retornada no final. No modo de teste a `id` do discriminador com melhor resposta será retornado ao *host*. Todos os discriminadores começam zerados e precisam ser treinados antes dos testes.

Esta etapa de implementação do `File Server` conclui a parte de programação voltada para FPGA, restando apenas a implementação da comunicação por parte do *host*.

3.4 Suporte para Sucuri

Um dos objetivos deste trabalho é implementar o suporte ao co-processador neural na Sucuri. Esse suporte foi feito através da habilidade de especializar nós proporcionada pela biblioteca. Os dois nós principais foram denominados de: `WSFTrainer` e `WSFPredict`.

A especialização `WSFTrainer` possui duas variações, porém o seu objetivo principal continua o mesmo: acionar o processador neural para treinar uma determinada imagem. Ao instanciar o nó deve-se passar o número de arestas de entrada que ele irá receber, o IP e a porta do dispositivo que está executando o `File Server`. Quando o nó estiver sendo executado na Sucuri um *socket* é criado e a comunicação estabelecida, as devidas mensagens são enviadas. A conexão só é fechada quando o sinal de finalização do treinamento é recebido, prosseguindo com o restante do grafo. As varianças apenas modificam o que será enviado ao dispositivo, a primeira versão envia um endereço indicando aonde a imagem se encontra, enquanto a segunda enviaria a `flag` indicando treinamento, a `id` do discriminador e a imagem inteira, em ambos os casos os parâmetros devem ser recebidos por outro nó do grafo em tempo de execução.

O nó `WSFPredict` funciona de forma similar ao `WSFTrainer`. No momento de instancia-lo também deve-se passar o numero de arestas, IP e porta do dispositivo. Em sua execução o nó estabelece uma conexão com o dispositivo e envia os dados

necessários. A conexão é encerrada apenas quando é recebida a mensagem informando o identificador do discriminador que deu a melhor resposta. As suas duas variações se comportam da mesma forma que as do `WSFTrainer` apenas mudando o conteúdo enviado nas mensagens.

É importante citar que além destes dois nós principais também foi criado um nó facilitador que tem o objetivo de facilitar o uso das versões que enviam a imagem para o dispositivo, dado que ambos os nós citados não fazem nenhum tipo de conversão. Este nó foi chamado de `WSFConverter` e utiliza a biblioteca OpenCV para manipular imagens. O trabalho desta especialização é receber uma imagem em sua porta de entrada, converte-la para uma imagem em tons de cinza e depois utilizar um algorítimo de conversão baseado em ranks utilizado em [28] para assimilar os valores dos pixeis a um endereço da memória RAM. O nó retorna um vetor de endereços já pronto para ser enviado ao dispositivo. O algorítimo irá dividir a imagem que está em tons de cinza em diversos conjuntos de tamanho N, para cada tupla será designado um valor chamado de *rank* que irá depender da forma que os valores do pixel dentro destas tuplas estão dispostos, este *rank* gerado corresponde ao endereço que será acessado na memória RAM do discriminador escolhido. É fácil perceber que o tamanho de N está diretamente conectado ao tamanho escolhido para cada memória RAM do Wisard.

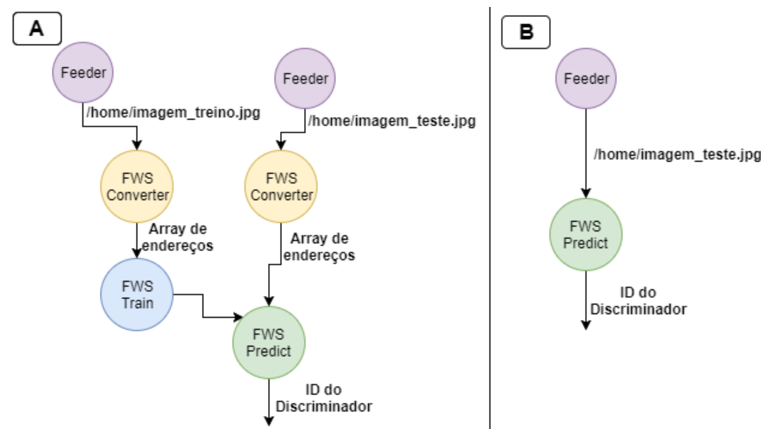


Figura 3.3: Exemplo de como ficaria o grafo de dependências utilizando os nós especializados

Na Figura 3.3 temos um exemplo de como seria o grafo de dependências quando utilizando os nós especializados. No quadro A temos a versão que utiliza o nó conversor e os nós que enviam a imagem por completo para serem treinadas ou testada. Existe uma dependência entre o nó `WSFPredict` e `WSFTrainer` porque a rede ainda não está treinada, então para o teste não ocorrer simultaneamente ao treino é forçada a dependência entre os nós. Podemos visualizar que a conversão de ambas as imagens pode ser feita em paralelo dado que são independentes. No quadro B temos a outra versão desenvolvida que será utilizada no teste, onde são

apenas enviados endereços para a FPGA, assumindo que os dados já estão no cartão de memória e os discriminadores são treinados ao inicializar o dispositivo. Embora a imagem mostre o nó `FWSPredict`, a versão equivalente do nó `FWSTrain` se comporta da mesma forma.

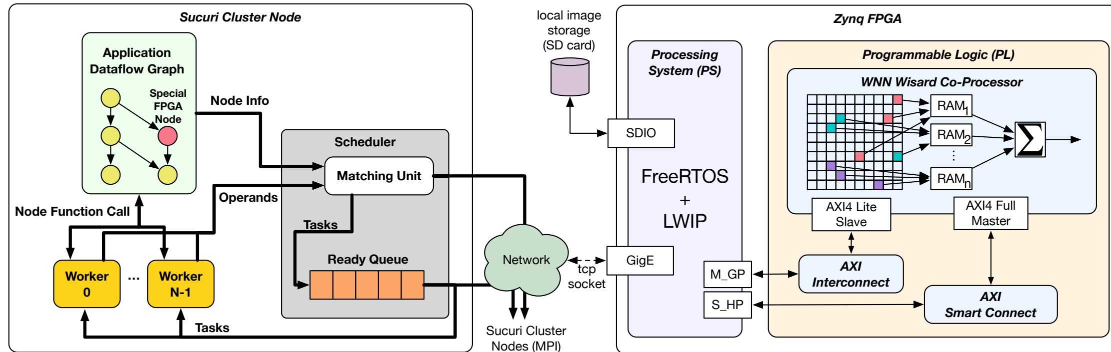


Figura 3.4: Arquitetura final do Sistema Co-Processador

Após a criação dos nós especializados para o uso da Sucuri com o dispositivo FPGA escolhido o sistema se encontra completo. A Figura 3.4 ilustra a arquitetura final do sistema mostrando todos os dispositivos e seus componentes internos. O trabalho se inicia com a Sucuri coordenando toda a comunicação feita dentro do *host* através do escalonador. Quando o nó especializado for chamado para execução ele irá criar uma conexão com o `File Server` que irá inicializar um novo *socket* ao aceitar a conexão através do processador ARM. Este irá acionar uma rotina de execução do co-processador neural. O Co-Processador retornará uma resposta para o `File Server` através barramento. Este por sua vez irá transferir via `socket` o valor para o nó Sucuri finalizando o ciclo.

Capítulo 4

Resultados Experimentais

Neste capítulo serão discutidos o caso de teste desenvolvido, os resultados referentes a implementação do co-processador neural na lógica reprogramável do circuito e também experimentos variando latência de rede.

Todos os experimentos foram conduzidos no servidor LabCad localizado na Universidade Estadual do Rio de Janeiro (UERJ). A máquina possui um Sistema Operacional Ubuntu 14.04, processador Intel Xeon E5-2609 1.90Ghz e 131GB de memória RAM. Os testes e a implementação do co-processador neural foram realizados em uma FPGA Xilinx ZC706 modelo XC7Z045-FFG900–2. A ferramenta Vivado versão 2017.3 foi responsável pela programação, implementação e resultados de consumo de energia e área do circuito do dispositivo.

4.1 Caso de teste

Um dos objetivos deste trabalho é testar as capacidades do sistema co-processador desenvolvido. Os testes irão comparar versões que não utilizam o processamento in-situ e avaliar a partir de que ponto valeria a pena enviar e processar os dados no *host*, que possui um processador de propósito geral com maior frequência de `clock`, ou enviar apenas uma requisição para um co-processador que executa a uma frequência mais baixa. Como caso de teste foi escolhido um cenário de reconhecimento facial em um contexto de redes sociais.

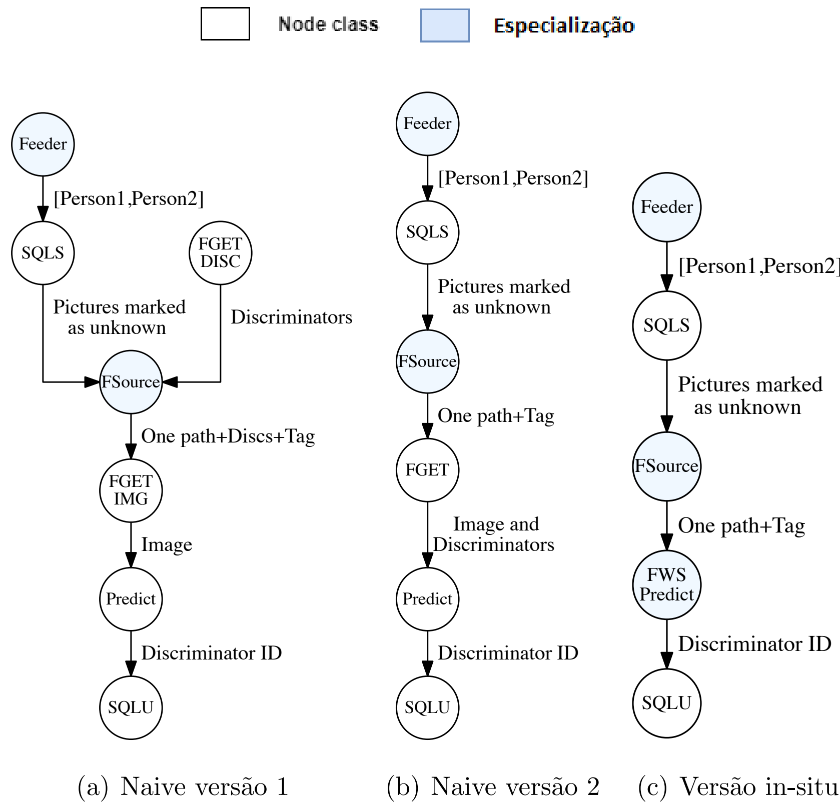
Redes sociais como o Facebook e o Instagram vem crescendo em popularidade nos últimos anos e uma das funções mais utilizados em ambas as redes sociais é a publicação de fotografias, muitas vezes da própria pessoa sozinha ou com vários amigos. Uma das funcionalidade disponibilizadas para fotos descarregadas é a habilidade de marcar rostos de pessoas que estão nas fotos, e depois indicando quem são pelo perfil correspondente da rede social. O Facebook além de permitir a marcação manual também faz a detecção e reconhecimento automático nas fotos publicadas de forma automática. É importante frisar que apenas amigos adicionados no per-

fil da rede social podem ser marcados, logo se alguém for um conhecido na vida real e estiver em uma foto publicada porém não é seu amigo na rede social ou não possui perfil criado esta pessoa provavelmente não será marcada automaticamente. Caso a amizade seja concretizada depois da foto esta também não será marcada automaticamente.

O objetivo do caso de teste é emular um ambiente onde existem duas pessoas com contas em uma rede social e suas fotos já armazenadas em um disco. Estas fotos já possuem marcações de outras pessoas ou rostos que foram detectados porém marcados como desconhecido. Estas duas pessoas finalmente se tornam amigas na rede social, o objetivo agora é buscar todas as fotos de ambas as pessoas que possuem uma marcação desconhecida e tentar reconhecer este novo amigo. É valido citar que por conta do Wisard implementado no co-processador neural não possuir um mecanismo de rejeição, ou seja, decidir que nenhum dos discriminadores em questão deu uma resposta valida o suficiente, todas as pessoas das fotos foram marcadas e apenas as duas pessoas que se emulará a amizade foram retiradas artificialmente.

Foram desenvolvidos três versões para o mesmo teste de caso. O primeiro utiliza do processamento in situ através do co-processador neural. É valido citar que neste cenário o *host* ficaria livre para processar outros dados enquanto o disco inteligente buscaria as fotos e tentaria marcar a nova pessoa adicionada. A segunda e terceira versão chamadas de `Naive 1` e `Naive 2` e emulam um disco comum conectado por rede, onde o processador ARM apenas recebe o endereço de onde a imagem se encontra no cartão de memória e retornar os dados desta imagem, todo o reconhecimento é feito no *host*. Para ambas as versões também é necessário o envio dos discriminadores já treinados dado que para o primeiro caso ao inicializar a placa estes são carregados na memória. A diferença é que para o Naive 1 os discriminadores são requisitados apenas uma única vez, enquanto que para a versão Naive 2 os arquivos dos discriminadores são requisitados toda vez que uma imagem nova precisa ser testada. Assume-se que todas as imagens salvas no cartão SD já estão convertidas em um array de ranks prontos para serem repassados para a Wisard, isto foi necessário pois não é possível utilizar o Opencv dentro da FPGA o que tornaria necessário o envio da imagem para o *host* de qualquer forma para a conversão em tons de cinza e execução de todo o algorítimo de ranks.

Na Figura 4.1 temos todos os grafos de dependências que foram programados através da Sucuri. Podemos visualizar que existem nós que executam as mesmas funções para todas as versões. O primeiro deles é o nó `Feeder` que envia o nome das duas pessoas que se tornaram amigas. Em seguida nó `SQLS` faz uma busca em um banco de dados `sqlite`. O banco de dados possui 2 tabelas. A primeira tabela possui o endereço da foto completa e quem postou esta foto. A segunda tabela possui uma chave externa conectada diretamente ao endereço da foto da primeira tabela.



(a) Naive versão 1 (b) Naive versão 2 (c) Versão in-situ

Figura 4.1: Estes são os três grafos de dependência modelados de acordo com o caso de teste

Esta chave indica que para existir uma entrada na tabela 2 esta deve corresponder a uma foto postada por alguém na tabela 1. A tabela 2 possui, além desta chave externa, o endereço de onde apenas o rosto da imagem original está, e a `id` de qual discriminador reconheceu este rosto. É neste campo que se encontra a marcação de desconhecido. O objetivo do nó `SQLS` é achar todas as fotos que possuem a marcação de desconhecido que foram postadas pela Pessoa 1 ou Pessoa 2. Todo o conjunto de imagens é então passado para outra especialização chamada `FSource`. Este nó irá pegar o vetor com os endereço de todas as imagens e libera-los um por um com uma tag especial, permitindo que varias instancias do próximo nó executem em paralelo, cada um calculando uma imagem. Similar ao nó `Source` explicado na secção 2.3. O ultimo nó em comum entre as versões desenvolvidas é o nó `SQLU`. Ele é responsável por atualizar o banco de dados preenchendo o valor desconhecido com a resposta dada pela Wisard, que será a `id` do discriminador com maior similaridade a foto testada.

A Figura 4.1(c) mostra versão onde será feito o processamento in situ. Como para o teste realizado todos os discriminadores já estão treinados será apenas utilizado o `FSWPredict` que irá receber um endereço e enviar para a FPGA, esperando o retorno indicando a `id` do discriminador com a melhor resposta que irá direto para sua porta de saída. Na Figura 4.1(a) temos a versão com o disco comum. O primeiro

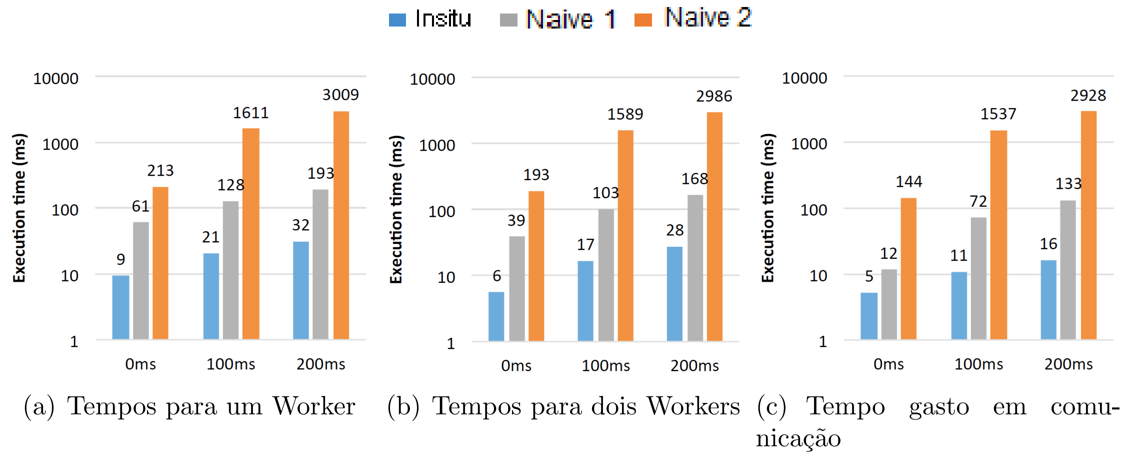
nó `FGETDISC` irá enviar uma mensagem para o disco pedindo os arquivos com os discriminadores treinados. O nó `FGETIMG` faz o pedido de apenas uma imagem que então é retornando para o próximo nó `PRED` que irá carregar os discriminadores já treinados e testar a imagem neles retornando também a `id` do melhor resultado para o nó `SQLU`. Para a versão Naive 2, apresentada na Figura 4.1(b) o nó `FGET` faz a requisição de uma imagem e dos discriminadores treinados. O nó `PRED` funciona da mesma forma que o da versão 1. Os nós que fazem acesso ao banco de dados e os nós `FGET`, `FGETDISC`, `FGETIMG` e `PRED` foram programados utilizando a classe `Node` padrão da Sucuri.

4.2 Execução do Caso de Teste

O objetivo deste experimento, é a execução deste grafo de dependências na Sucuri avaliando a performance de cada uma das versões. As latências da rede foram variadas entre 0, 100 e 200ms com intuito de indicar a partir de que ponto problemas na rede afetariam a comunicação com o `File Server` e tornaria a movimentação de dados para o *host* depreciativa. Para emular esta variação de latência foi utilizado a ferramenta Netem disponibilizada na maioria das distribuições Linux. Netem é uma ferramenta que permite a emulação de algumas propriedades de redes como delay, perda, reordenação e duplicação de pacotes [29]. Além de variar a latência, foram variados o número de Workers na Sucuri entre 1 e 2. A Figura 4.2 possui as médias dos tempos em segundos para a execução total do caso de teste com 55 rostos marcados como "desconhecidas"no banco de dados. Isto significa que serão retornadas 55 fotos no banco de dados e a Wisard será executada uma vez para cada uma dessas fotos.

É importante relembrar que por conta de limitações na implementação do `File Server` com o FreeRtOS e o LwIP apenas um socket pode ser instanciado, limitando uma conexão por vez. Para garantir esta execução, quando utilizando dois Workers foi utilizada uma função da Sucuri que permite forçar a execução de apenas um Worker em certos nós. Com isso foi possível executar apenas uma instancia por vez dos nós que tinham algum tipo de comunicação com o `File Server`. Entretando, todos os outros nós podem executar em paralelo.

Na Figura 4.2(a) temos os tempos em segundos quando executando o caso de teste com apenas 1 Worker. Ao utilizar apenas uma instancia da classe Worker, o grafo se comporta de maneira sequencial. Os resultados demonstram que mesmo sem nenhuma latência de rede, o sistema co-processador consegue ter o melhor desempenho do que as outras duas versões *Naive*. Como a versão in-situ não necessita transferir os discriminadores treinados por rede ela acaba tendo o melhor desempenho. Para versão *Naive* 1 os discriminadores são enviados apenas uma vez, enquanto



(a) Tempos para um Worker (b) Tempos para dois Workers (c) Tempo gasto em comunicação

Figura 4.2: Média dos tempos medidos (em segundos) quando executando o caso de teste e variando a latência.

que a versão 2 necessita transferir os discriminadores e monta-los do lado do host para cada uma das 55 imagens.Por conta dessa diferença na transferência de dados podemos visualizar que ao aumentar a latência a disparidade de tempos acaba aumentando cada vez mais.

Ao compara a Figura 4.2(c) com os outros dois gráficos, podemos inferir que a comunicação ocupa a maior parte do tempo total. Isto se dá pois o Wisard é um modelo simples que não exige muita computação e o outro tipo de computação realizado no caso de teste é a pesquisa e atualização de um banco de dados que também não deveria ocupar tanto tempo.

Quando executando o caso de teste com 2 Workers é possível que duas tarefas executem ao mesmo tempo. Escritas no banco de dados enquanto se testa uma imagem, a requisição de dados ao servidor enquanto uma previsão é feita no host, são todos cenários possíveis. Acompanhando pela Figura 4.2(b) e Figura 4.2(a) temos confirmação que os tempos diminuíram por conta do paralelismo que consegue mascarar um pouco o tempo de comunicação. Porém como só é possível estabelecer uma conexão por vez com o `File Server` esta ainda se demonstra o gargalo principal. Ao aumentar o número de Workers é bem provável que os tempos não tenham grandes mudanças dado que as tarefas que executariam em paralelo não são o gargalo.

4.3 Avaliação do microprocessador ARM

Neste subcapítulo serão discutidos os resultados no que diz respeito ao tempo de execução de rotinas de treino e de teste. O objetivo é comparar o co-processador neural com a mesma implementação do algorítimo da Wisard no microprocessador ARM. O processador ARM executa em uma frequência de aproximadamente 667 Mhz, enquanto o co-processador executa em 50 Mhz.

	Co-Processor	ARM com cache	Speedup
Por Treino	322067,20	489366,00	1.52
Por Teste	428560,20	2524919,00	5.89

Tabela 4.1: Microprocessador ARM versus Co-processador Neural em ciclos do ARM

A Tabela 4.1 dispõe das médias em ciclos de ARM de quando o algorítimo realiza uma rotina de treino ou teste de para apenas uma imagem. Podemos visualizar que o co-processador demora consideravelmente menos ciclos de ARM para executar ambas as rotinas, indicando que a implementação em na lógica da FPGA executa mais rápido do que no microprocessador. Este resultado demonstra que para esta FPGA vale a pena a implementação do algorítimo em hardware. Outra informação interessante é que os ciclos gastos no treino para os dois casos é menor. Isto se dá pois apenas um discriminador é acessado durante a rotina de treinamento, enquanto que para o teste todos devem ser acessados, aumentando o número de operações e a disparidade de tempo entre as versões. Na ultima coluna da Tabela 4.1 temos os *speedups* resultantes das medições dos ciclos. Podemos visualizar que o *speedup* em relação a rotina de teste é bem maior. Isto se dá pois os discriminadores do co-processador neural foram mapeados em grupos separados de *Block-RAMs*, permitindo seu acesso em paralelo.

4.4 Tempo de reposta do File Server

Esta seção irá avaliar o tempo de resposta do `File Server` quando uma requisição é feita pelo host. Todos os valores medidos se baseiam na execução de apenas uma imagem sendo testada ou treinada, e quanto tempo o Wisard demora pra executar esta rotina e/ou quanto tempo o `File Server` demora para realizar as operações nos arquivos que ocorrem antes da execução do Wisard.

	In-situ	Naive 1 e 2
Teste	0,69	401,86
FOp por Teste	79,87	3252,91
Treino	17,38	***
FOp por Treino	286,95	***

Tabela 4.2: Média dos tempos de resposta do `File Server` em milissegundos

A Tabela 4.2 dispõe das médias dos resultados em milissegundos obtidos através de medições realizadas no `File Server` desenvolvido. A primeira linha da tabela se refere ao tempo de execução de apenas um teste. Para o co-processador isto equivale a chamar a Wisard e esperar o seu retorno. Na versão in-situ o teste foi realizado em apenas 0,69 ms dado que está implementado em hardware. Para as

outras duas versões o teste é realizado no *host* através do Python o que o torna significativamente mais lento, demorando 401,86 ms. O próximo valor, chamado de FOp por teste, representa o tempo que demorou todas as operações de arquivo feitas no `File Server` antes da chamada da rede neural. No caso do in-situ as operações foram: Buscar, abrir, ler e fechar o arquivo da imagem. Para as outras duas versões o `File Server` deve buscar, abrir, ler e fechar as imagens e os arquivos contendo os discriminadores treinados, tornando-os bem mais lentos em suas respostas dado que são feitas uma quantidade maior de operações.

No treinamento foram utilizados 9 fotos por pessoa, totalizando 36 fotos dado que temos 4 discriminadores. As ultimas duas linhas da Tabela 4.2 representam o tempo para realizar o treinamento de todas as fotos. Estes valores não se aplicam para a versão Naive 1 e 2 pois os seus discriminadores estão salvos em arquivos que já foram previamente treinados pela mesma função. Para versão in-situ o treinamento é realizado ao inicializar a placa. O tempo total de treinamento equivale a soma das duas ultimas linhas da tabela. Podemos visualizar que embora este seja realizado apenas uma vez ele possui um tempo significativamente baixo.

4.5 Área do Circuito

Como citado previamente no subcapítulo 3.1, por não existir alocação dinâmica quando se trata de recursos físicos da FPGA, os discriminadores da Wisard devem possuir uma quantidade fixa. Para os experimentos deste trabalho foram implementados um total de 4 discriminadores com o intuito de avaliar o quanto estes ocupariam em recursos. A Tabela 4.3 dispõe dos resultados gerados pelo Vivado. Os recursos da FPGA são divididos em: Look-up Tables RAMs (LUTRAMs), Flip-Flops e Block RAMs (BRAM).

As LUTs são os recursos mais básicos da FPGA e costumam ser utilizados para a construção de lógica combinatorial. Os Flip-Flops são utilizados para construção de lógica sequencial, como registradores. As LUTRAMs são pequenas memórias construídas através de LUTs, ao contrário das BRAMs que são blocos especializados de memória.

Recursos	Utilizados	Disponível	Utilizados (%)
LUT	2449	218600	1.12
LUTRAM	289	70400	0.41
FF	2627	437200	0.60
BRAM	69.50	545	12.75

Tabela 4.3: Consumo de recursos na FPGA ZC706 gerado pelo Vivado 2017.3

Na Tabela 4.3 podemos visualizar que a maioria dos recursos lógicos como

`Look-Up Tables` (LUT) e Flip-Flops (FF) não foram bastante utilizados, alcançando em torno de 1% de consumo. Isto se dá porque as operações lógicas em si da Wisard são simples, apenas sendo necessário um somador para cada discriminador. No caso do componente `Block RAM` (BRAM) que representa blocos de memória especializados temos 13% de utilização para apenas 4 discriminadores implementados. Este resultado implica que seria possível multiplicar esse design em aproximadamente 7 vezes até estourar a capacidade máxima de BRAMs, permitindo um total de 28 discriminadores por cada placa ZC706. Com isso seria possível emular uma alocação dinâmica de discriminadores, mantendo uma lista de quais seriam os discriminadores validos, enquanto os outros não seriam levados em consideração até uma nova requisição de treino seja feita.

Por conta do consumo de recursos ser baixo, a Wisard pode ser implementada em FPGAs mais singelas, que possuem menos recursos do que a placa ZC706. Com esta área disponível também possível a implementação de novos recursos e até mesmo de outras versões do Wisard mais robustas.

4.6 Consumo de Energia

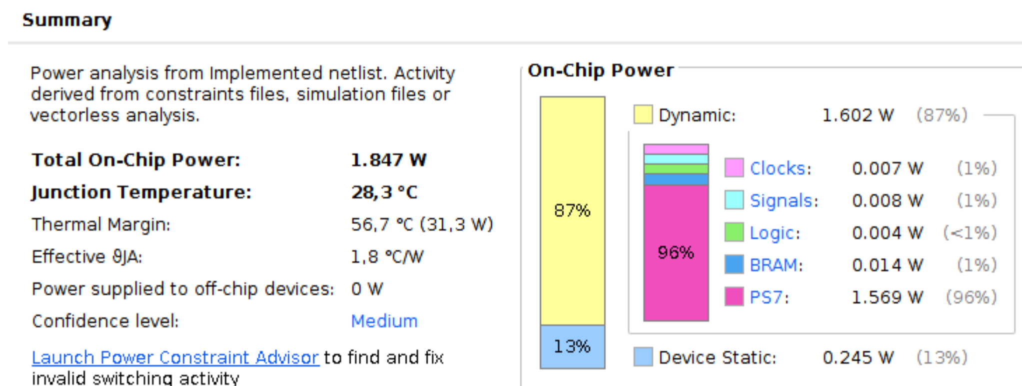


Figura 4.3: Relatório de consumo de energia gerado pelo Vivado

Embora não tenha sido possível medir o consumo de energia do dispositivo com um multímetro durante a execução do caso de teste, a ferramenta Vivado produz um relatório com uma estimativa de consumo previsto para o design. Na Figura 4.3 temos estimado um total de 1.847 Watts gastos por ciclo. Este consumo é divido no gráfico a direita em estático e dinâmico. O consumo estático representa os momentos em que o dispositivo está de repouso, exercendo uma computação mínima, totalizando 0.245 Watts que equivale a 13% do consumo total. O consumo dinâmico representa o caso em que os recursos estão todos sendo utilizados. Este é divido em diversos componentes como Lógica e BRAM que representam o consumo do co-processador em si, ou seja, do hardware especializado. Eles somados dão em

aproximadamente 3.48% do consumo de energia total. O chamado PS7 representa o processador ARM que nitidamente possui a maior porcentagem do consumo total equivalente a 83.52% com 1.569 Watts gastos.

Capítulo 5

Conclusão

Neste trabalho foi apresentado um sistema co-processador com capacidades de processamento in-situ. O sistema utilizou de recursos de uma FPGA para a implementação de uma rede neural sem peso e do microprocessador ARM para o auxilio de comunicações com o ambiente externo. Neste trabalho também foi incluso o suporte deste sistema em uma biblioteca que emula o ambiente Dataflow chamada Sucuri. A biblioteca se demonstrou uma ferramenta de alta flexibilidade, facilitando a implementação de varias especializações que irão facilitar o uso futuro do sistema. Experimentos foram realizados em cima do ARM e de um teste de caso aplicado na Sucuri. Os resultados preliminares demonstraram que a implementação em hardware teve um desempenho superior as outras versões implementadas que não faziam uso do processamento in-situ.

Embora o sistema tenha demostrado bom desempenho, alguns pontos ainda precisam ser reavaliados. O uso do sistema operacional FreeRtOS em conjunto com o LwIP, embora tenha facilitado a implementação da comunicação via *sockets*, se demonstrou muito limitado, sendo a maior causa do gargalo nos testes. Uma outra direção seria a otimização do design do co-processador utilizando *flags* e *pragmas* na compilação do Vivado HLS. Também seria possível aumentar a frequência de relógio do co-processador dado que ele esta executando a apenas 50 Mhz.

Pretendemos implementar também o sistema para uma placa Alpha Data conectada via PCI Express. Esta implementação ficaria ligada diretamente a um *host* que executa um sistema operacional, provavelmente com o ambiente Dataflow da Sucuri. Este tipo de cenário iria facilitar a comunicação direta do hardware, retirando o problema anterior de limitações com *threads* e *sockets*.

Em relação aos testes realizados, após retirada da limitação de uma conexão por vez, um cenário mais realista seria estabelecido, permitindo a execução de múltiplos `Workers`. Por termos escolhida a Sucuri como API, a adaptação do código para execução em um *cluster* com diversas placas conectadas, cada uma com seu próprio cartão de memória, seria transparente. Seria possível utilizar também a versão da

Sucuri que não possui o escalonador centralizado [19], caso este se demonstre um gargalo. Também temos em mente a execução de outros casos de testes, utilizando os outros nós desenvolvidos para caso geral. Um cenário voltado para segurança como de `surveillance` em câmeras também poderia ser modelado na Sucuri, dado que a Wisard foi implementada de forma genérica.

Referências Bibliográficas

[1] MARZULO, L. A. J. *Explorando Linhas de Execuçao Paralelas com Programaçao Orientada por Fluxo de Dados*. Tese de Doutorado, Universidade Federal do Rio de Janeiro, 2011.

[2] ALVES, T., GOLDSTEIN, B. F., FRANÇA, F. M., etal. "A Minimalistic Dataflow Programming Library for Python". In: *Computer Architecture and High Performance Computing Workshop (SBAC-PADW), 2014 International Symposium on*. IEEE, 2014.

[3] SUNDARARAJAN, P. "High performance computing using FPGAs", *Xilinx White Paper: FPGAs*, pp. 1–15, 2010.

[4] ALVES, T. A., MARZULO, L. A., FRANCA, F. M., etal. "Trebuchet: exploring TLP with dataflow virtualisation", *International Journal of High Performance Systems Architecture*, v. 3, n. 2/3, pp. 137, 2011. ISSN: 1751-6528. doi: 10.1504/IJHPSA.2011.040466.

[5] JUN, S. W., LIU, M., LEE, S., etal. "BlueDBM: An appliance for Big Data analytics". In: *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–13, June 2015. doi: 10.1145/2749469.2750412.

[6] KIM, J., ABBASI, H., CHACÓN, L., etal. "Parallel in situ indexing for data-intensive computing". In: *2011 IEEE Symposium on Large Data Analysis and Visualization*, pp. 65–72, Oct 2011. doi: 10.1109/LDAV.2011.6092319.

[7] "NGD Systems announces availability of industry's first Computational Storage". http://www.prnewswire.com/news-releases/ngd-systems-announces-availability-of-industrys-first-computational-storage-300493319.html. Accessado em: Dezembro 20, 2017.

[8] ALEKSANDER, I., DE GREGORIO, M., FRANÇA, F. M. G., etal. "A brief introduction to Weightless Neural Systems." In: *ESANN*, pp. 299–305, 2009.

[9] "Xilinx Vivado HLx Desing Suite - Synthesis ". https://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_2/ug901-vivado-synthesis.pdf, . Accesssado em: Dezembro 20, 2017.

[10] "Xilinx Vivado HLx Desing Suite - Implementation ". https://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_4/ug904-vivado-implementation.pdf, . Accesssado em: Dezembro 20, 2017.

[11] GURD, J. R., KIRKHAM, C. C., WATSON, I. "The Manchester prototype dataflow computer", *Communications of the ACM*, v. 28, n. 1, pp. 34–52, 1985.

[12] SAKAI, S., HIRAKI, K., KODAMA, Y., etal. "An architecture of a dataflow single chip processor". In: *ACM SIGARCH Computer Architecture News*, v. 17, pp. 46–53. ACM, 1989.

[13] DENNIS, J. B., MISUNAS, D. P. "A preliminary architecture for a basic dataflow processor". In: *ACM SIGARCH Computer Architecture News*, v. 3, pp. 126–132. ACM, 1975.

[14] FERREIRA, VICTOR C., M. L. A. J. "Computação de Alto Desempenho em GPUs com Execução Guiada por Fluxo de Dado". In: *Workshop on Microarchitectural Changes*, 2017.

[15] MCCULLOCH, W. S., PITTS, W. "A logical calculus of the ideas immanent in nervous activity", *The bulletin of mathematical biophysics*, v. 5, n. 4, pp. 115–133, 1943.

[16] GRIECO, B. P., LIMA, P. M., DE GREGORIO, M., etal. "Producing pattern examples from "mental" images", *Neurocomputing*, v. 73, n. 7, pp. 1057–1064, 2010.

[17] CARVALHO, D. S., CARNEIRO, H. C., FRANÇA, F. M., etal. "B-bleaching: Agile Overtraining Avoidance in the WiSARD Weightless Neural Classifier." In: *ESANN*, 2013.

[18] CARVALHO, C. B., FERREIRA, V. C., FRANCA, F. M., etal. "Towards a Dataflow Runtime Environment for Edge, Fog and In-Situ Computing". In: *2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, pp. 115–120. IEEE, 2017.

[19] RAFAEL J. N. SILVA, BRUNNO GOLDSTEIN, L. S. A. C. S. L. A. J. M. T. A. O. A. F. M. G. F. "Task Scheduling in Sucuri Dataflow Library". In: *IEEE 28th International Symposium on Computer Architecture and High Performance Computing Workshop*. IEEE, 2016.

[20] KIM, J., ABBASI, H., CHACON, L., etal. "Parallel in situ indexing for data-intensive computing". In: *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, pp. 65–72. IEEE, 2011.

[21] LAKSHMINARASIMHAN, S., SHAH, N., ETHIER, S., etal. "Compressing the incompressible with ISABELA: In-situ reduction of spatio-temporal data". In: *European Conference on Parallel Processing*, pp. 366–379. Springer, 2011.

[22] KLASKY, S., ABBASI, H., LOGAN, J., etal. "In situ data processing for extreme-scale computing", *Scientific Discovery through Advanced Computing Program (SciDAC'11)*, 2011.

[23] MUTHURAMALINGAM, A., HIMAVATHI, S., SRINIVASAN, E. "Neural network implementation using FPGA: issues and application", *International journal of information technology*, v. 4, n. 2, pp. 86–92, 2008.

[24] ZHANG, C., LI, P., SUN, G., etal. "Optimizing fpga-based accelerator design for deep convolutional neural networks". In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 161–170. ACM, 2015.

[25] GADEA, R., CERDÁ, J., BALLESTER, F., etal. "Artificial neural network implementation on a single FPGA of a pipelined on-line backpropagation". In: *Proceedings of the 13th international symposium on System synthesis*, pp. 225–230. IEEE Computer Society, 2000.

[26] GIRONÉS, R. G., SALCEDO, A. M. "Systolic implementation of a pipelined on-line backpropagation". In: *Microelectronics for Neural, Fuzzy and Bio-Inspired Systems, 1999. MicroNeuro'99. Proceedings of the Seventh International Conference on*, pp. 387–394. IEEE, 1999.

[27] DUNKELS, A. "Design and Implementation of the lwIP TCP/IP Stack", *Swedish Institute of Computer Science*, v. 2, pp. 77, 2001.

[28] KHAKI, K. M. *Weightless neural networks for face recognition.* Tese de Doutorado, Brunel University School of Engineering and Design PhD Theses, 2013.

[29] "Netem Networking - Linux Foundation". https://wiki.linuxfoundation.org/networking/netem. Accessado em: Fevereiro 02, 2018.

Apêndice A

Código Fonte da Wisard

Algoritmo A.1: Implementação HLS da Wisard utilizando o Xilinx Vivado HLS 2017.3.

```
#define IMG_SIZE 8000

static bool disc0[IMG_SIZE * 64];
static bool disc1[IMG_SIZE * 64];
static bool disc2[IMG_SIZE * 64];
static bool disc3[IMG_SIZE * 64];

void wisard(bool mode, int disc_id, volatile int *img_addr, int *res) {
    #pragma HLS INTERFACE m_axi depth=8000 port=img_addr offset=slave bundle=image
    #pragma HLS INTERFACE s_axilite port=img_addr bundle=control
    #pragma HLS INTERFACE s_axilite port=mode bundle=control
    #pragma HLS INTERFACE s_axilite port=disc_id bundle=control
    #pragma HLS INTERFACE s_axilite port=res bundle=control
    #pragma HLS INTERFACE s_axilite port=return bundle=control

  int img[IMG_SIZE];

  memcpy(img, (const int*) img_addr, IMG_SIZE * sizeof(int));

  int counter0 = 0;
  int counter1 = 0;
  int counter2 = 0;
  int counter3 = 0;
  int index = 0;

  if (mode == 0) //training
    {
    switch (disc_id) {
    case 0:
      for (int i = 0; i < IMG_SIZE; i++) {
        index = 64 * i + img[i];
        disc0[index] = 1;
      }
      *res = -1;
      break;
    case 1:
      for (int i = 0; i < IMG_SIZE; i++) {
        index = 64 * i + img[i];
        disc1[index] = 1;
```

```
40        }
41        *res = -2;
42        break;
43      case 2:
44        for (int i = 0; i < IMG_SIZE; i++) {
45          index = 64 * i + img[i];
46          disc2[index] = 1;
47        }
48        *res = -3;
49        break;
50      case 3:
51        for (int i = 0; i < IMG_SIZE; i++) {
52          index = 64 * i + img[i];
53          disc3[index] = 1;
54        }
55        *res = -4;
56        break;
57      default:
58        *res = -5;
59      }
60    } else //test
61    {
62      for (int i = 0; i < IMG_SIZE; i++) {
63        index = 64 * i + img[i];
64        counter0 = counter0 + disc0[index];
65        counter1 = counter1 + disc1[index];
66        counter2 = counter2 + disc2[index];
67        counter3 = counter3 + disc3[index];
68      }
69      *res = max4(counter0, counter1, counter2, counter3);
70    }
71  }
```