UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS DE QUIXADÁ

CURSO DE ENGENHARIA DE COMPUTAÇÃO

ANA VICTÓRIA ARAÚJO MAIA

UM GERADOR DE ARQUITETURAS SOC PARA EXECUÇÃO DE REDES NEURAIS CONVOLUCIONAIS EM FPGAS

QUIXADÁ

2020

ANA VICTÓRIA ARAÚJO MAIA

UM GERADOR DE ARQUITETURAS SOC PARA EXECUÇÃO DE REDES NEURAIS CONVOLUCIONAIS EM FPGAS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Campus de Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Computação.

Orientador: Prof. Dr. Cristiano Bacelar de Oliveira

QUIXADÁ

2020

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

M184g Maia, Ana Victória Araújo.
Um gerador de arquiteturas SoC para execução de Redes Neurais Convolucionais em FPGAs / Ana Victória Araújo Maia. – 2020.
44 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Engenharia de Computação, Quixadá, 2020.
Orientação: Prof. Dr. Cristiano Bacelar de Oliveira.

1. Rede neural convolucional. 2. Arranjos de lógica programável em campo. 3. Sistemas programáveis em chip. I. Título.

CDD 621.39

ANA VICTÓRIA ARAÚJO MAIA

UM GERADOR DE ARQUITETURAS SOC PARA EXECUÇÃO DE REDES NEURAIS CONVOLUCIONAIS EM FPGAS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Campus de Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Computação.

Aprovada em: _____/_____/_____

BANCA EXAMINADORA

_____

Prof. Dr. Cristiano Bacelar de Oliveira (Orientador)
Universidade Federal do Ceará (UFC)

_____

Prof. Dr. André Ribeiro Braga
Universidade Federal do Ceará (UFC)

_____

Prof. Dr. Paulo de Tarso Guerra Oliveira
Universidade Federal do Ceará (UFC)

Dedico esse trabalho a minha família, vocês são e vão ser sempre o motivo da minha força. E dedico também aos meus amigos por terem me ajudado a enfrentar todas as dificuldades.

AGRADECIMENTOS

Agradeço primeiramente a Deus por ter me dado o presente da vida.

Aos meu pais e a toda a minha família seja de sangue ou não, que me deram total apoio durante todo o curso e me ajudaram a enfrentar todos os tipos de dificuldade.

A todos os professores que já me ensinaram, em especial aos que me ensinaram durante a graduação e aqueles que me orientaram em projetos e bolsas, por ter me proporcionado conhecimento tanto específicos do curso, mas também conhecimento para a vida.

Ao meu orientador Prof. Dr. Cristiano Bacelar de Oliveira pela excelente orientação e pelos momentos de ensinamento durante esse trabalho.

Aos meus avaliadores Prof. Dr. André Ribeiro Braga e Prof. Dr. Paulo de Tarso Guerra Oliveira pelos elogios, sugestões, incentivos e principalmente as críticas que fizeram esse trabalho ficar ainda melhor.

A todos os meus amigos e colegas que me ajudaram durante toda a graduação, os que passaram noites em claro comigo, os que me deram atenção apenas durante alguns minutos do dia ou da noite e todos que ficavam dizendo "Vai dar certo". Em especial João Lucas, Matheus Fernandes, Marisa Silva, Marianna Pinho, Mateus Sousa, Marcelo Martins, Gabriel Uchoa por terem estado ao meu lado durante o curso.

Para a conclusão deste trabalho agradeço carinhosamente a Johnny Marcos por parar o que estava fazendo e me ajudar a resolver um problema de biblioteca, João Henrique por me ouvir e me ajudar a pensar em como resolver os problemas de implementação, Arthur Antunes por estar sempre disponível pra ajudar a entender o conteúdo de redes neurais, Carlos Alberto e Francisca Beatriz por me ajudar no TCC1 e em especial a João Paulo por sempre querer conversar sobre os nossos trabalhos e Iana Mary por sempre me incentivar.

A todos os técnicos-administrativo do Campus Quixadá, em especial, Abdul-Hamid Moreira por toda a ajuda durante o curso.

A todos os funcionários da UFC Quixadá pelo excelente trabalho exercido durantes todos esses anos e pela amizade divertida que conquistei com alguns fosse apenas para pedir para abrir uma porta de uma sala, ou desejando bom dia, ou até mesmo só jogando conversa fora pelos corredores. Vocês fazem o campus ficar ainda mais bonito.

Todos vocês citados direto ou indiretamente fizeram esse trabalho acontecer. Muito obrigada a todos.

“Tudo depende do tipo de lente que você utiliza para ver as coisas.”

(Jostein Gaarder)

RESUMO

Convolutional Neural Networks (CNNs) são bastante utilizadas em diversas aplicações. Atualmente, há um interesse no desenvolvimento e execução dessas redes em sistemas embarcados. Para facilitar esse desenvolvimento para aplicações em sistemas embarcados esse trabalho apresenta um sistema automático para geração de arquiteturas *System on Chip* para a execução de CNNs em *Field Programmable Gate Arrays* (FPGAs). Utilizou-se na implementação a técnica de *Co-Design* para gerar a estrutura de *System on Chip*, tendo como resultado um sistema composto de subsistemas de *software* e *hardware*. No desenvolvimento do subsistema de *hardware* utilizou-se a técnica de *High Level Synthesis* (HLS) para gerar um coprocessador em forma de *Intellectual Property* (IP) específico para a camada de convolução. Essa é a camada responsável por cerca de 63% do tempo total de execução por *software* da CNN usada como caso de teste. Os resultados mostram que, na arquitetura gerada, a convolução foi executada 15,2 vezes mais rápido que a convolução implementada em *software*.

Palavras-chave: CNN. *System on Chip*. HLS. FPGA.

ABSTRACT

Convolutional Neural Networks (CNNs) are widely used in several applications. Currently, there is an interest in the development and execution of these networks in embedded systems. To facilitate this development for applications in embedded systems, this work presents an automatic system for the generation of System on Chip architectures for the execution of CNNs in Field Programmable Gate Arrays (FPGAs). It was used in the implementation of a Co-Design technique to generate a System on Chip structure, resulting in a system composed of software and hardware subsystems. In the hardware subsystem development uses a High Level Synthesis (HLS) technique to generate a coprocessor in the form of Intellectual Property (IP) specific to the convolution layer. This is a layer responsible for about 63 % of the total running time of CNN software used as a test case. The results presented that, in the generated architecture, the convolution was executed 15.2 times faster than the convolution implemented in software.

Keywords: CNN. System on Chip. HLS. FPGA.

LISTA DE FIGURAS

Figura 1 – Etapas de processamento de imagem 17
Figura 2 – Passos do processo de convolução 18
Figura 3 – Passos do processo de *maxpool* com região *pooling* de tamanho 2x2 19
Figura 4 – Etapas de desenvolvimento de um sistema utilizando *Co-Design* 22
Figura 5 – Arquitetura básica de um FPGA 23
Figura 6 – Etapas do processo de HLS 24
Figura 7 – Arquitetura RTL 25
Figura 8 – Algoritmo da Soma de Diferenças Absolutas (SDA) 26
Figura 9 – Máquina de estados de alto nível do algoritmo SDA 27
Figura 10 – Bloco operacional SAD e máquina de estados finita do bloco de controle . . 28
Figura 11 – Esquemático de uma CNN implementada para a execução em FPGA 29
Figura 12 – Diagrama do processo de geração do IP 30
Figura 13 – Diagrama do *CNN Core IP* 30
Figura 14 – Arquitetura *Zynq APSoC* 32
Figura 15 – Exemplo de imagens retiradas do banco de dados MNIST *digits* 33
Figura 16 – Rede CNN implementada 34
Figura 17 – Diagrama representando as etapas de desenvolvimento do sistema implementado 34
Figura 18 – Etapa I - Geração de de arquivos de cabeçalho 35
Figura 19 – Etapa II - Implementação da rede em C++ 36
Figura 20 – Etapa III - Geração do IP 36
Figura 21 – Integração entre *hardware* e *software* 37
Figura 22 – Processo de ativação do IP durante a execução do código C++ da rede . . . 38
Figura 23 – Arquitetura do sistema final 38
Figura 24 – Gráfico com valores de acurácia do treino e validação durante o treinamento da rede 39
Figura 25 – Gráfico com valores de *cross entropy loss* do treino e validação durante o treinamento da rede 40
Figura 26 – Comparação das predições feitas em *python* e em C++ 41

LISTA DE TABELAS

Tabela 1 – Resultados da avaliação da rede 40
Tabela 2 – Resultados do *profiling* de cada camada da rede nos três testes 42
Tabela 3 – Comparação entre o tempo médio de computação da rede, com a convolução executada por *hardware* e por *software* 42

LISTA DE QUADROS

Quadro 1 – Comparativo dos trabalhos relacionados 31

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
CNN	*Convolutional Neural Network*
CPUs	*Central Processing Units*
FPGAs	*Field Programmable Gate Arrays*
GPUs	*Graphic Processing Units*
HDL	*Hardware Description Language*
HLS	*High Level Synthesis*
IP	*Intellectual Property*
MEF	Máquina de Estados Finita
PL	*Programmable Logic*
PS	*Processing System*
ReLU	*Rectifier Linear Unit*
RTL	*Register Transfer Level*
SDA	Soma de Diferenças Absolutas
SoC	*System on Chip*
TPUs	*Tensor Processing Units*
VHSIC	*Very High Speed Integrated Circuits*

SUMÁRIO

1 INTRODUÇÃO 14
2 FUNDAMENTAÇÃO TEÓRICA 16
2.1 Visão Computacional 16
2.1.1 Convolutional Neural Network 17
2.1.2 Computação de Borda 20
2.2 *System on Chip* 21
2.2.1 Field Programmable Gate Arrays (FGPAs) 22
2.2.2 High Level Synthesis 23
3 TRABALHOS RELACIONADOS 29
4 PROCEDIMENTOS METODOLÓGICOS 32
4.1 Rede CNN Modelo 33
4.2 Geração da Arquitetura SoC 34
4.2.1 Etapa I - Geração de de arquivos de cabeçalho 35
4.2.2 Etapa II - Implementação da rede em C++ 36
4.2.3 Etapa III - Geração do IP 36
4.2.4 Etapa IV - Integração entre hardware e software 37
5 RESULTADOS 39
5.1 Avaliação da rede em *python* 39
5.2 Resultado da rede implementada em C++ 41
5.3 Resultado do *profiling* feito da rede em C++ 41
5.4 Resultado do teste de tempo feito sobre o IP gerado 42
6 CONCLUSÃO E TRABALHOS FUTUROS 43
REFERÊNCIAS 44

1 INTRODUÇÃO

Podemos ver o crescente uso das técnicas de Visão Computacional nas mais variadas aplicações. Exemplos de trabalhos podem ser vistos em diversas áreas, como o de Fan *et al.* (2020), na indústria, onde tais técnicas são usadas para ajudar a identificar maçãs defeituosas, ou em trabalhos médicos, como para melhorar a identificação automatizada de indivíduos desconhecidos por comparação de radiografias dentárias panorâmicas (HEINRICH *et al.*, 2020), ou, ainda, no processamento e reconhecimento de leitura labial (MAHADEVASWAMY *et al.*, 2021) ou mesmo detecção de pedestres (YU *et al.*, 2021). Percebe-se, portanto, que os algoritmos de Visão Computacional tiveram um rápido progresso nos últimos anos. Em particular, a combinação de Visão Computacional com o aprendizado de máquina contribuiu para o desenvolvimento de sistemas e aplicações práticas envolvendo o uso e a interpretação de imagens e vídeos, de forma a tentar, com várias limitações, reproduzir a percepção humana.

O aprendizado de máquina é uma sub-área da Inteligência Artificial que permite que os computadores aprendam a partir dos dados sem serem programados explicitamente. Dentre os algoritmos de Aprendizagem de Máquina utilizados no campo de Visão Computacional existem as *Convolutional Neural Networks* (CNNs), que se mostraram muito eficazes em áreas como reconhecimento e classificação de imagens (KHAN *et al.*, 2018).

Convencionalmente, as CNNs são executadas em *Central Processing Units* (CPUs), *Graphic Processing Units* (GPUs) e nos últimos anos em *Tensor Processing Units* (TPUs)[1]. Porém, junto com o crescente uso de aplicações utilizando CNNs, há o crescente interesse em utilizar essas aplicações em sistemas embarcados, em um cenário de computação de borda (Seção 2.1.2). Assim, outra plataforma de execução promissora bastante frequente nos trabalhos atuais são os *Field Programmable Gate Arrays* (FPGAs), como em (BONNARD *et al.*, 2020), onde é mostrado o uso de FPGAs em um projeto de câmeras inteligentes sincronizadas com um sistema de visão de múltiplas visualizações para reconhecimento de objetos. Mittal (2020) afirma que, geralmente, FPGAs fornecem maior eficiência energética do que GPUs e CPUs e melhor desempenho do que CPUs. Além disso, a reconfigurabilidade dos FPGAs permite a exploração rápida do vasto espaço de projeto de configurações e parâmetros utilizados em uma CNN.

Sistemas Embarcados são sistemas que podem ser amplamente definidos como dispositivos que contém componentes de *hardware* e *software* fortemente acoplados para executar

[1] https://cloud.google.com/blog/products/gcp/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu

uma única função. Podem fazer parte de um sistema maior, não destinados, necessariamente, a serem independentemente programáveis pelo usuário, atuando com o mínimo ou sem interação humana. Eles operam em ambientes restritos onde a memória, a capacidade de computação e o fornecimento de energia são limitados (JIMÉNEZ *et al.*, 2014).

Dadas as limitações impostas pelos sistemas embarcados, há pesquisas que buscam conciliar os requisitos para execução de uma CNN em tais dispositivos. Um exemplo de solução desta natureza pode ser visto no trabalho de Zhang *et al.* (2020), onde os autores desenvolveram uma estrutura híbrida de CNN com alterações para reduzir a complexidade computacional, provando que a implantação de CNNs em dispositivos portáteis é possível e vantajosa.

Dado o exposto, este trabalho apresenta o desenvolvimento de um sistema cujo objetivo é gerar, automaticamente, a partir de um modelo CNN pré-treinado, uma arquitetura *System on Chip* (Seção 2.2) para a execução de Redes Neurais Convolucionais em FPGAs. Para isto foi necessário analisar a implementação de uma CNN, usada como caso de teste, e seu desempenho em uma plataforma *System on Chip*. Por fim, a arquitetura de *hardware* gerada mostrou um ganho de desempenho de cerca de 2,3 vezes em relação ao tempo de execução do modelo puramente em *sofware*. Isto foi conseguido devido à implementação de um coprocessador para realização da convolução, cujo desempenho foi melhorado em torno de 15,2 vezes.

Os demais capítulos deste trabalho apresentam o desenvolvimento deste projeto, estando organizados da seguinte forma:

- **Capítulo 2**: apresenta os principais conceitos utilizados no desenvolver do trabalho;
- **Capítulo 3**: apresenta uma visão geral dos trabalhos que estão relacionados com este;
- **Capítulo 4**: apresenta os detalhes da abordagem proposta e as etapas que foram realizadas para a conclusão deste trabalho;
- **Capítulo 5**: apresenta e discute os resultados;
- **Capítulo 6**: apresenta a conclusão e possíveis trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este Capítulo apresenta os conceitos utilizados no desenvolvimento da solução proposta. Inclui uma introdução à Visão Computacional e a Redes Neurais Convolucionais para entender a teoria por trás da implementação, uma visão geral de *System on Chip* e a metodologia de desenvolvimento chamada *Co-Design*, que será utilizada na construção da arquitetura proposta.

2.1 Visão Computacional

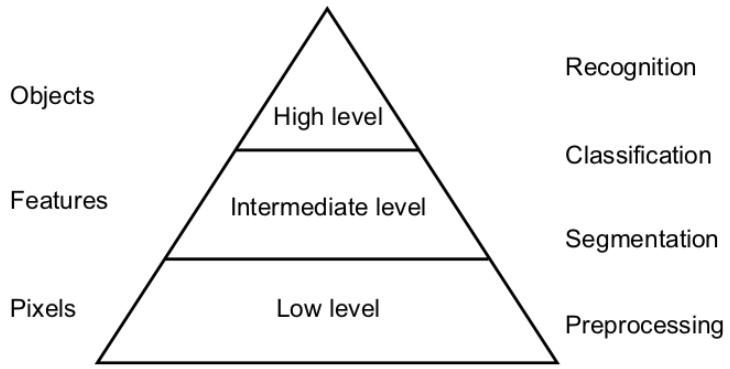
Khan *et al.* (2018) explicam que a Visão Computacional procura desenvolver métodos que são capazes de replicar uma das capacidades mais incríveis do sistema visual humano, ou seja, inferir características do mundo real puramente usando a luz refletida de vários objetos, diferente de aplicações puras de processamento de imagens, focadas principalmente no tratamento de imagens brutas sem fornecer nenhum *feedback* de conhecimento sobre elas.

Com base no nível de abstração das informações de saída, as tarefas de Visão Computacional podem ser divididas em três categorias diferentes: *low-level*, *mid-level*, e *high-level*. Khan *et al.* (2018) afirmam que a visão de *low-level* identifica correspondências de imagens para recuperação de geometria e movimento. Também calcula o padrão e analisa o movimento aparente de objetos, superfícies e bordas em uma cena visual. Visão de *mid-level* tenta responder à pergunta: "Como o objeto se move?". Para isso é utilizado a geometria de múltiplas visualizações, que inferem as informações da cena 3D a partir de imagens 2D, de modo que a reconstrução 3D possa ser possível. Visão de *high-level* fornece uma interpretação coerente da imagem. Ela determina quais objetos estão presentes na cena e interpreta suas inter-relações, com o reconhecimento de padrões respondendo a perguntas como "Existe um tigre na imagem?" Ou "Esse vídeo é um drama ou uma ação?".

O processamento de imagem pode ser considerado como um passo de pré-processamento para Visão Computacional, por conseguir extrair primitivas de imagens fundamentais, incluindo arestas e cantos, operações de filtragem, morfologia, etc. As operações de processamento de imagens geralmente podem ser agrupadas em uma pirâmide como mostrado na Figura 1.

Bailey (2011) explica que na base da pirâmide estão as operações de pré-processamento, que melhoram as informações relevantes da imagem enquanto suprime qualquer informação irrelevante. Operações como: correção de distorção, aprimoramento de contraste e filtragem para

Figura 1 – Etapas de processamento de imagem



Fonte: Bailey (2011)

a redução de ruídos ou detecção de bordas são exemplos de operações de pré-processamento. Acima da base estão as operações de segmentação que detectam objetos ou regiões, na imagem, que possuem alguma propriedade comum, tendo como resultado criação de regiões a partir da imagem. Operações como detecção de cor, crescimento de região e rotulagem de componentes conectados são exemplos de operações de segmentação. Após a segmentação estão as operações de classificação, que utilizam as regiões formadas anteriormente transformando-as em rótulos para identificar objetos ou classificá-los em categorias predefinidas.

No topo da pirâmide estão as operações de reconhecimento, onde se obtém uma descrição ou uma interpretação da cena. Estas operações incluem, ainda, tarefas de classificação dos objetos na imagem. No contexto de Visão Computacional, o uso de modelos de redes tipo *Convolutional Neural Network* é bastante comum em tarefas de classificação.

2.1.1 Convolutional Neural Network

Essa subseção foi baseada principalmente na explicação de Khan *et al.* (2018). Eles afirmam que *Convolutional Neural Network* (CNN) é uma categoria de Redes Neurais que provou ser muito eficaz em áreas como reconhecimento e classificação de imagens, sendo bastante utilizada para casos em que se deseja aprender padrões a partir de uma mídia de entrada de alta dimensão como por exemplo, imagens ou vídeos. A CNN é composta por várias camadas. Dentre elas estão as camadas convolucionais, *pooling* e *fully connected*.

Khan *et al.* (2018) explicam, ainda, que a camada convolucional é o componente mais importante da CNN. É nessa camada onde ocorre a convolução de um conjunto de filtros chamados *kernels* com a entrada, de forma a gerar um mapa de características como saída. Cada

filtro convolucional é uma matriz de números discretos menor que a matriz de entrada. Eles são inicializados com valores aleatórios e com o treino da rede os valores vão sendo definidos. Nessa camada é feita a operação de convolução entre os filtros e a entrada da camada.

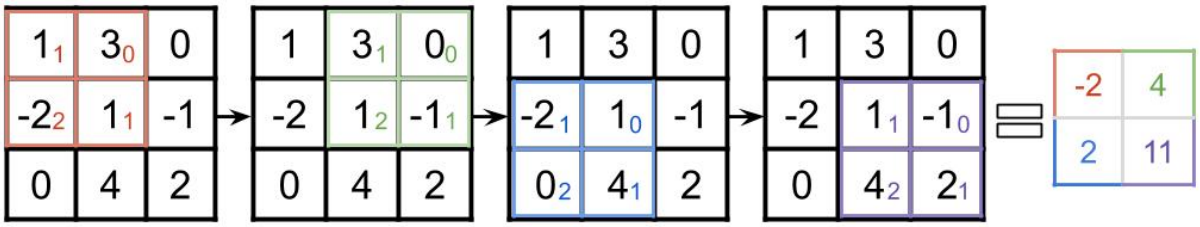
A convolução pode ser definida matematicamente utilizando a equação apresentada em (SZELISKI, 2011):

$$g(i,j) = \sum_{k,l} f(k,l)h(i-k,j-l) \tag{2.1}$$

onde g é o resultado da convolução, f é a entrada e h é o *kernel*. Essa equação pode ser interpretada como uma adição dos valores do filtro deslocados $h(i-k,j-l)$ multiplicados pelos valores dos *pixels* de entrada $f(k,l)$.

A Figura 2 mostra os passos da operação de convolução. À esquerda do símbolo de igualdade é mostrado o deslocamento do filtro (em destaque) pela imagem (representada pela matriz) e à direita do símbolo de igualdade é mostrado o resultado final da convolução após todos os deslocamentos do filtro, conforme equação 2.1.

Figura 2 – Passos do processo de convolução

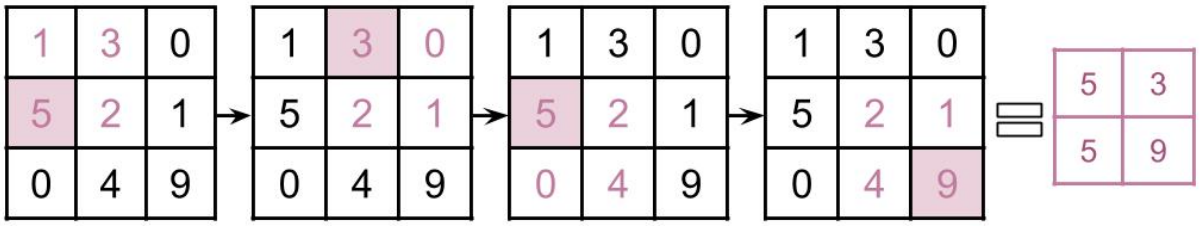


Fonte: Elaborado pelo autor

Khan *et al.* (2018) explicam também que a camada de *pooling* opera realçando características em regiões da entrada. Essa operação é definida por funções *pooling*, que podem ser a média, a função máximo ou a função mínimo. Semelhante a convolução, a região do *pooling* é deslizada por toda a entrada aplicando a função escolhida.

A Figura 3 mostra os passos da operação de *maxpool*. À esquerda do sinal de igualdade é mostrado o deslocamento da região de *pooling* (onde estão os números destacados) na imagem (matriz) e o maior valor dentro da área selecionada sendo destacado. À direita do sinal de igualdade está o resultado da seleção do maior valor dentro das regiões de *pooling* que se deslocaram pela imagem.

Figura 3 – Passos do processo de *maxpool* com região *pooling* de tamanho 2x2



Fonte: Elaborado pelo autor

A camada *fully connected* ou camada densa, geralmente é a última parte da arquitetura de uma CNN. Cada neurônio de uma camada densa é conectado com todos os neurônios da camada anterior. A operação feita nessa camada pode ser representada como uma multiplicação de matrizes seguida de uma adição de um vetor com valores de *bias* e aplicando uma função de ativação f. Podendo ser definida como:

$$y = f(W^T x + b), \tag{2.2}$$

onde x e y são vetores de entrada e saída, respectivamente, W é a matriz contendo os pesos dos neurônios das camadas, b representa o vetor de *bias*. O valor de *bias* é explicado por Haykin (2007), onde ele afirma que *bias* é um parâmetro externo ao neurônio que tem o efeito de aplicar uma transformada afim ao seu valor de saída.

Khan *et al.* (2018) afirmam que a função de ativação é usada como um mecanismo de seleção que decide se um neurônio será ou não acionado dadas todas as suas entradas. Uma função de ativação que é geralmente utilizada em redes neurais profundas é a *Rectifier Linear Unit* (ReLU). A ReLU tem uma importância prática especial por sua rápida computação. Ela mapeia a entrada para 0 se tiver um valor negativo ou mantém o valor caso contrário. Ela pode ser definida como:

$$f_{relu}(x) = max(0, x). \tag{2.3}$$

Ainda segundo Khan *et al.* (2018), existem funções de ativação específicas para cada tipo de problema. Para problemas de classificação com múltiplas classes usa-se a função *softmax*. A *softmax* calcula a probabilidade de cada classe de ser o valor de saída da rede. Pode ser definida como:

$$p_n = \frac{exp(p'_n)}{\sum_k exp(p'_k)}, \tag{2.4}$$

onde p_n é a probabilidade do neurônio n que representa uma classe, p'_n é o valor de saída do neurônio n e p'_k é o valor de saída do neurônio k, sendo esses neurônios os da camada a qual a função esta sendo aplicada.

Durante o treinamento da rede é utilizada uma função de perda (*loss function*) para determinar a qualidade das predições feitas pela rede com os dados de treinamento pelos quais as respostas de predição são conhecidas. A *loss function* quantifica a diferença entre a predição da rede e a predição correta. A *categorical cross-entropy* é a função de custo para problemas de classificação mais difusa adotada pela maioria das arquiteturas neurais. Podendo ser definida como:

$$L(p,y) = -\sum_{n} y_n \, log(p_n), \qquad n \in [1,N], \tag{2.5}$$

onde y é o valor de saída desejado, p é a probabilidade da categoria de saída, N é quantidade total de neurônios da camada de saída da rede e p_n é a probabilidade da categoria n calculado a partir da função *softmax*.

Raschka e Mirjalili (2017) afirmam que para verificar o desempenho do modelo da rede implementada dada a arquitetura e os hiper-parâmetros é importante comparar o valor de acurácia do treinamento e da validação calculados durante o treinamento da rede. A acurácia pode ser definida como:

$$ACC = \frac{TP+TN}{FP+FN+TP+TN}, \tag{2.6}$$

onde TP ou *true positive* são as predições positivas feitas corretamente, TN ou *true negative* as predições negativas feitas corretamente, FP ou *false positive* as predições positivas que foram feitas erroneamente, FN ou *false negative* as predições negativas que foram feitas erroneamente.

2.1.2 Computação de Borda

A computação de borda[1] se refere às tecnologias que permitem que a computação seja realizada na borda da rede, em dados de serviços em nuvem e dados de serviços IoT. No paradigma da computação de borda, os dispositivos não são apenas consumidores de dados, mas também atuam como produtores de dados. Os dispositivos podem não apenas solicitar serviço e conteúdo da nuvem, mas também podem executar as tarefas de computação da nuvem. Eles

[1] Em inglês *Edge Computing*

podem realizar armazenamento de dados, armazenamento em cache e processamento, bem como distribuir serviço de solicitação e entrega da nuvem para o usuário. Com esses trabalhos na rede, é necessário que o sistema com esse paradigma atenda aos requisitos de maneira eficiente em serviços como confiabilidade, segurança e proteção de privacidade (CAO *et al.*, 2018).

Como as CNNs podem ter várias camadas com vários neurônios, o treinamento pode necessitar de muito poder computacional. Para suprir essa necessidade o treinamento da rede geralmente é feito em sistemas com GPUs, TPUs ou em sistemas executados em nuvem. Porém, concluído o treinamento, os requisitos computacionais para a execução e obtenção de uma predição da rede caem drasticamente. Com isso pode-se utilizar uma rede treinada em dispositivos com recursos limitados, em um cenário de computação de borda, com sistemas embarcados ou dispositivos *System on Chip*.

2.2 *System on Chip*

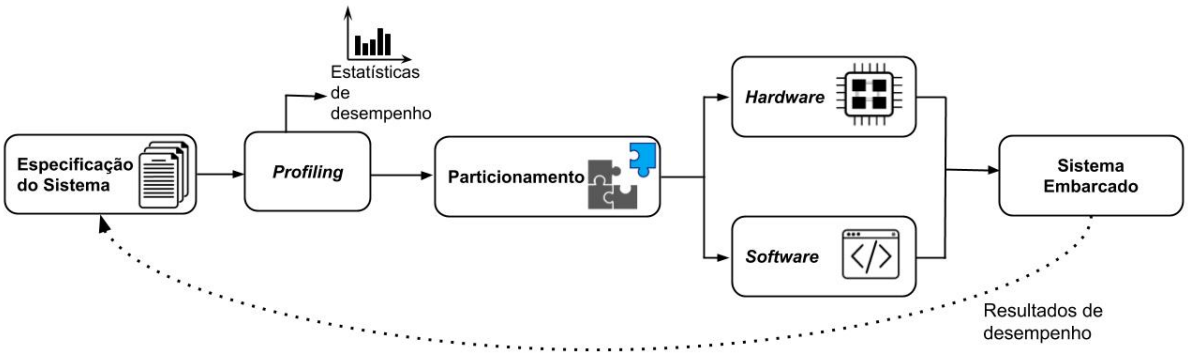
Segundo Pasricha e Dutt (2008) circuitos integrados em um único chip são comumente chamados de *System on Chip* (SoC). Normalmente são constituídos de vários componentes heterogêneos complexos, como: processadores, memórias, *hardware* personalizado, periféricos, blocos externos de *Intellectual Property* (IP) e uma estrutura de interconexão para a comunicação entre estes componentes.

Para desenvolver um sistema complexo usando SoC podemos usar a metodologia de *Co-Design*. Esta metodologia permite que o projetista projete um sistema com um equilíbrio de subsistemas de *hardware* e *software* de forma que, trabalhando juntos, consigam atingir um comportamento específico que atenda aos requisitos do projeto (NEDJAH; MOURELLE, 2007).

O desenvolvimento de um sistema utilizando *Co-Design* segundo Nedjah e Mourelle (2007), segue as etapas da Figura 4, onde primeiramente são definidas as especificações do sistema, as quais são implementadas em nível de *software*. Após isto, é feito o *profiling* do código, processo em que são identificadas as regiões críticas do sistema. Essas regiões críticas são regiões em que a solução implementada não atende aos requisitos de desempenho necessárias ou desejáveis. Após a identificação das regiões críticas é feito o particionamento do algoritmo transformando as regiões críticas em *hardware* e fazendo uma adaptação no *software*. Tendo como resultado um sistema embarcado.

Para identificar as regiões críticas é necessário executar o programa com dados representativos e analisar a execução. Uma análise para verificar as restrições de tempo pode

Figura 4 – Etapas de desenvolvimento de um sistema utilizando *Co-Design*



Fonte: Adaptado de Nedjah e Mourelle (2007)

ser feita colocando em diversas regiões do algoritmo "carimbos de tempo", variáveis que armazenarão valores lidos do tempo atual em que aquela região do código começou a ser executada e terminou a execução. Essa informação juntamente com o número de vezes que uma dada região foi executada permite verificar onde um programa gasta muito tempo. Outra forma de análise é selecionar regiões candidatas e estimar o tempo de execução apenas desses blocos. Esses blocos são identificados observando as ocorrências de laços, condicionais e fins de blocos. Os fins de blocos são identificados por retornos, comandos de quebra e comandos do tipo *goto* e *continue*.

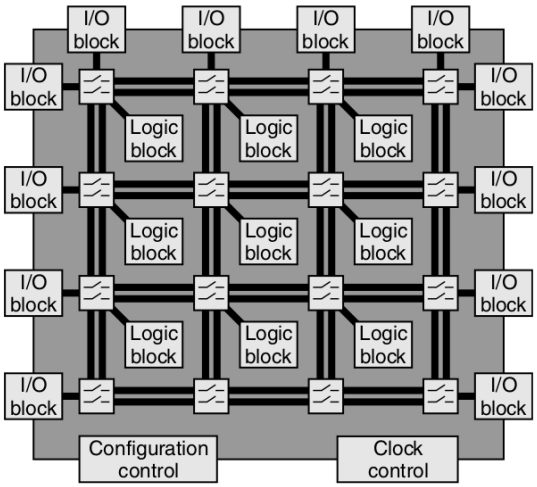
Na fase de desenvolvimento do subsistema de *hardware* é feita a tradução do código da região crítica em alguma *Hardware Description Language* (HDL), projetando os blocos lógicos. Em seguida, mapeia-se os blocos gerados e seleciona-se o roteamento de interconexão entre os blocos. Na fase de desenvolvimento do *software* é feita uma adaptação do protocolo de particionamento. Esse protocolo consiste em passar os parâmetros necessários para o subsistema de *hardware* e acessar os resultados retornados após a execução. Os subsistemas de *hardware* podem ser construídos em em Field Programmable Gate Arrays (FPGAs).

2.2.1 Field Programmable Gate Arrays (FGPAs)

Segundo Bailey (2011) FPGAs são dispositivos reconfiguráveis constituídos principalmente de blocos lógicos agrupados em uma estrutura de grade. Essa estrutura forma uma matriz de roteamento programável permitindo que esses blocos sejam conectados em configurações arbitrárias. Os blocos lógicos são baseados em uma arquitetura de tabela de consulta, permitindo a implementação de qualquer função arbitrária das entradas. Além dos blocos ló-

gicos existem blocos de entrada e saída, que permitem a conexão de sinais entre o FPGA e os dispositivos externos, e os blocos responsáveis por sinais de controle e de *clock*. A Figura 5 mostra a estrutura básica e os componentes essenciais de um FPGA genérico.

Figura 5 – Arquitetura básica de um FPGA



Fonte: Bailey (2011)

Segundo Cardoso e Diniz (2009), apesar do enorme potencial dos FPGAs, eles são extremamente difíceis de programar, pois exigem o uso de HDLs, como *Verilog* e *Very High Speed Integrated Circuits* (VHSIC) *Hardware Description Language* (conhecida como VHDL). Apesar da capacidade dessas linguagens de elevar o nível de abstração para construções estruturais ou mesmo comportamentais, as abstrações que elas expõem ao programador ainda são de nível bastante baixo. Por conta dessa dificuldade foi desenvolvido uma técnica chamada *High Level Synthesis* (HLS) para facilitar a programação em FPGAs.

2.2.2 High Level Synthesis

Nedjah e Mourelle (2007) afirmam a partir de vários trabalhos que HLS é o processo de derivar implementações de *hardware* para circuitos através de linguagens de programação de alto nível. Em vez de uma HDL, é possível usar uma linguagem de programação de alto nível, como C ou C++, para descrever o comportamento do sistema. Sendo feita a tradução para uma HDL em um estágio posterior do processo de *design*. Keating (2011) mostra que essa técnica tem como vantagens:

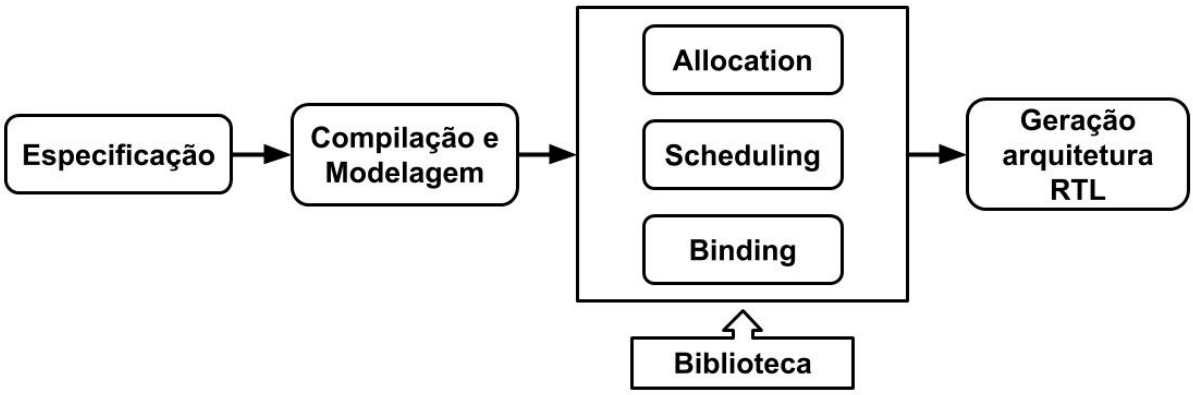
- O *design* pode ser avaliado em muitas arquiteturas distintas antes de se comprometer com o projeto detalhado;

- A quantidade de código necessária para um determinado *design* diminui, assim como o número de erros;
- É mais fácil de testar, depurar e raciocinar sobre o código.

Uma ferramenta de HLS típica executa as etapas mostradas na Figura 6, que são (COUSSY *et al.*, 2009):

1. Compilar e modelar a especificação do projeto;
2. Alocar recursos de *hardware* (*Allocation*);
3. Organizar as operações para ciclos de *clock* (*Scheduling*);
4. Fazer as vinculações necessárias (*Binding*);
5. Gerar uma arquitetura *Register Transfer Level* (RTL).

Figura 6 – Etapas do processo de HLS



Fonte: Adaptado de Coussy *et al.* (2009)

Coussy *et al.* (2009) afirmam que o processo de HLS sempre começa com a compilação da especificação. Esse primeiro passo transforma a descrição da entrada em uma representação formal, além de incluir otimizações como eliminação de código desnecessário, eliminação de falsas dependências de dados e transformações de *loops*. O modelo formal produzido pela compilação exibe os dados e os controles de dependências entre as operações. Eles explicam, ainda, que a etapa de *allocation* define o tipo e a quantidade de recursos de *hardware* (como unidades funcionais, armazenamento ou componentes de conectividade) necessários para satisfazer as restrições do *design*. A quantidade de recursos depende do dispositivo alvo utilizado. Dependendo da ferramenta de HLS, alguns componentes precisam ser adicionados durante a etapa de *scheduling* e *binding*, tais componentes são normalmente selecionados de uma biblioteca de componentes RTL.

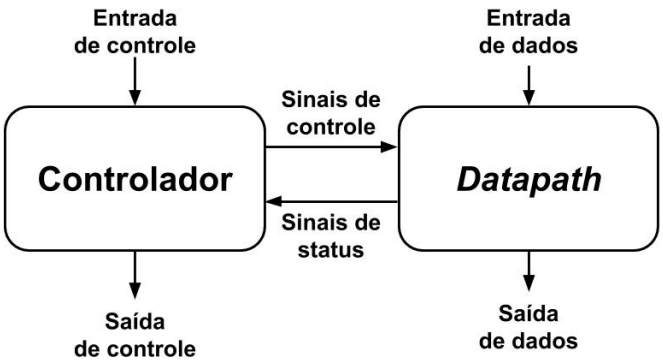
Na etapa de *scheduling*, são organizadas todas as operações requeridas na especifica-

ção em ciclos de execução. Dependendo do componente funcional pode ser executado com um ciclo de *clock* ou em mais ciclos. Operações podem ser executadas em paralelo se não possuir dependência de dados entre eles e se os recursos disponíveis forem suficientes.

Na etapa de *binding* é explicado que cada variável que carrega valores entre os ciclos de *clock* é vinculada a uma unidade de armazenamento. Cada operação no modelo de especificação é vinculada a uma das unidades funcionais capazes de executar a operação. Se houver várias unidades com essa capacidade, o algoritmo de ligação deve otimizar essa seleção. A vinculação de armazenamento e unidade funcional também depende da vinculação de conectividade. Essa vinculação requer que cada transferência de componente para componente seja vinculada a uma unidade de conexão, como um barramento ou um multiplexador. Na última etapa é aplicado todas as decisões feitas nas etapas anteriores e gerado um modelo RTL de *design* sintetizado.

A arquitetura RTL é constituída de um controlador e um *datapath* como visto na Figura 7. O controlador é uma máquina de estados finita que orquestra o fluxo de dados no *datapath*, definindo os valores dos sinais de controle, como as entradas selecionadas de unidades funcionais, registradores e multiplexadores. O *datapath* é um conjunto de elementos de armazenamento, como registradores e memórias, um conjunto de unidades funcionais, como ULA, e elementos interconectados, como multiplexadores (COUSSY *et al.*, 2009). O *datapath* fornece os dados, os resultados e executa as funções atribuídas. Cada *datapath* envia sinais para o controlador que são usados para determinar a próxima etapa na computação (GAJSKI *et al.*, 2009).

Figura 7 – Arquitetura RTL



Fonte: Adaptado de Gajski *et al.* (2009)

Para melhor entender os passos do HLS, Vahid (2008) mostra um exemplo da

conversão de forma manual de uma descrição de um sistema feito com uma linguagem de programação de alto nível em uma descrição RTL. O sistema de exemplo implementa a Soma de Diferenças Absolutas (SDA). O algoritmo implementado é representado na Figura 8.

Figura 8 – Algoritmo da Soma de Diferenças Absolutas (SDA)

```
1   int SDA (byte A [256], byte B[256]){
2       uint soma;
3       short uint i;
4       soma = 0;
5       i = 0;
6       while (i < 256) {
7           soma = soma + abs(A[i] - B[i]);
8           i = i + 1;
9       }
10      return (soma);
11  }
```

Fonte: Adaptado de Vahid (2008)

O primeiro passo é transformar o algoritmo SDA mostrado na Figura 8 em uma máquina de estados de alto nível, como pode ser visto na Figura 9.

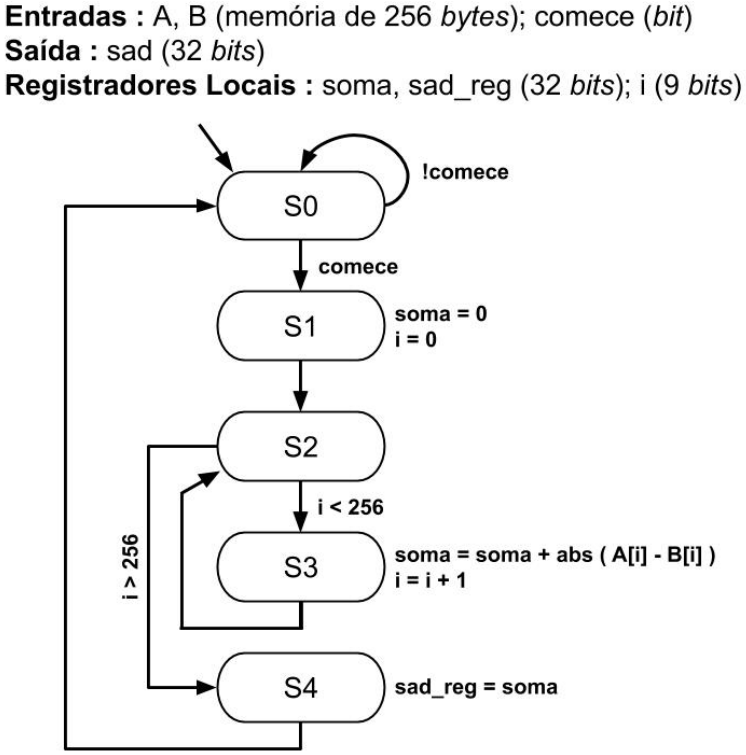
Na parte superior, estão as declarações da entrada, saída e os registradores que serão utilizados (*soma*, *i*, *sad_reg*). A máquina de estados de alto nível tem a seguinte execução:

- Inicia-se os valores de *soma* e *i* com 0;
- Em seguida, começa o laço: se *i* for menor que 256, a máquina de estados calculará o valor absoluto da diferença de A[i] e B[i]. Essa notação (A[i]) refere-se aos dados da palavra *i* na memória A. Em seguida atualiza o valor de *soma*, incrementa o valor de *i* e retorna ao início do laço;
- Caso *i* for maior que 256 é carregado no registrador *sad_reg* o valor de *soma*;
- Por fim é retornado ao estado inicial.

Nesse passo é feito algumas escolhas de recursos para a arquitetura. Foi definido que:

- Como o registrador *soma* conterá o valor parcial da soma de diferenças, então ele terá uma largura de 32 *bits*;
- Como o registrador *i* terá valores de 0 a 256, então ele terá uma largura de 9 *bits*;
- Como o registador *sad_reg* será conectado à saída *sad*, então os dois terão uma largura de

Figura 9 – Máquina de estados de alto nível do algoritmo SDA



Fonte: Adaptado de Vahid (2008)

32 *bits*.

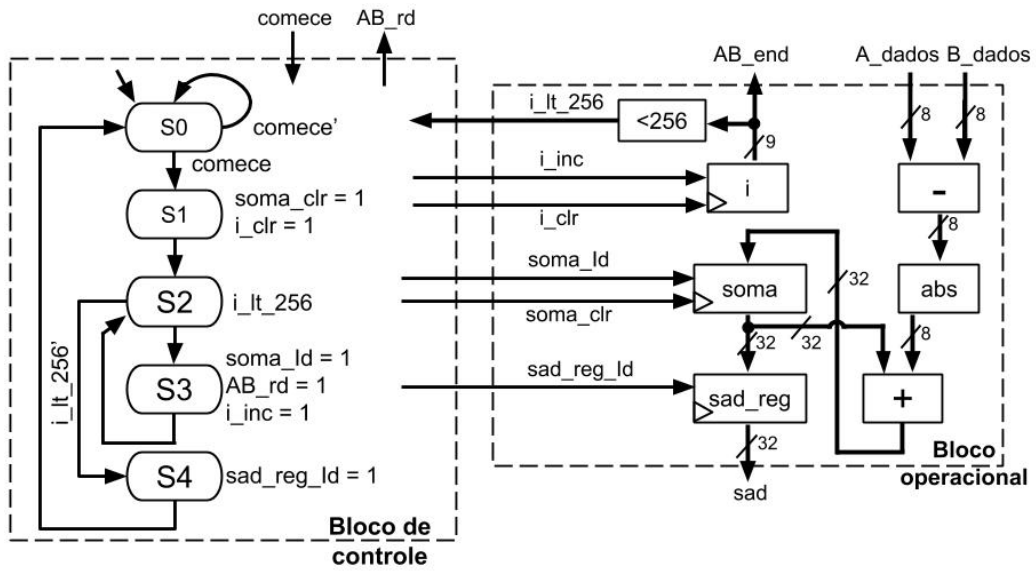
O segundo passo é criar um bloco operacional. A partir da máquina de estados de alto nível da Figura 9 é possível perceber que é necessário criar:

- Um subtrator;
- Um componente de valor absoluto;
- Um somador;
- Um comparador para *i* e 256.

Com isso podemos construir o bloco operacional mostrado no lado direito da Figura 10. Como o somador tem uma largura de 32 *bits*, é feito no componente com o símbolo de "+", uma adição de 24 *bits* à esquerda do valor da entrada vinda do componente que calcula *abs* que contém 8 *bits*.

O terceiro passo é conectar o bloco operacional a um bloco de controle, como é mostrado na Figura 10. Foi definido as interfaces para as memórias A e B como sendo constituídas pela linha de leitura (*AB_rd*), endereço (*AB_end*) e dados (*A_dados* e *B_dados*). Após a conexão é convertido a máquina de estados de alto nível da Figura 9 em uma Máquina de Estados Finita (MEF), como ser visto no lado esquerdo da Figura 10.

Figura 10 – Bloco operacional SAD e máquina de estados finita do bloco de controle



Fonte: Adaptado de Vahid (2008)

Para finalizar o projeto deve ser feito a conversão da MEF em uma implementação de blocos de controle. O bloco de controle pode ser desenvolvido em um processo de 5 passos:

Passo 1 : Crie uma MEF que descreva o comportamento desejado do bloco de controle;

Passo 2 : Crie a arquitetura cuja entradas são os *bits* do registrador de estado e as entradas da MEF. As saídas são os *bits* de próximo estado e as saídas da MEF;

Passo 3 : Atribua um número binário único a cada um dos estados;

Passo 4 : Crie uma tabela verdade para a lógica combinacional de modo tal que a lógica gere as saídas e os sinais de próximo estado de maneira correta para a MEF;

Passo 5 : Implemente a lógica combinacional usando qualquer método.

Os conceitos apresentados sobre a técnica de HLS são utilizados na implementação da solução para a construção de um bloco externo de IP para o uso no subsistema de *hardware*. O IP é um bloco funcional que pode conter funções implementadas em VHDL para o uso em uma aplicação específica e que pode ser reutilizado em outras aplicações (MAXFIELD, 2008).

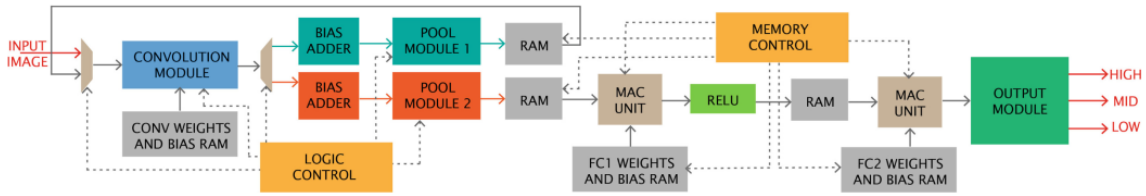
3 TRABALHOS RELACIONADOS

Dado o contexto deste trabalho, este Capítulo apresenta trabalhos com aplicações e técnicas de desenvolvimento que serviram como inspiração para a criação da solução proposta.

Ram *et al.* (2020) investigaram uma implementação FPGA de CNNs com processamento mínimo de *software*, em um esforço para acelerar o estágio de inferência de CNNs. Empregou-se quantização de *software* para facilitar a implementação em FPGAs e *pipelining* parcial para processar as várias camadas de uma CNN típica. O *design* do *hardware* foi codificado usando *Verilog* HDL. Como resultado foi desenvolvido um acelerador de *hardware* para classificação de dígitos manuscritos utilizando CNN cerca de 20x mais rápido comparado a execução em uma CPU Intel Core i5 de 8ª geração, e sendo capaz de alcançar um melhor equilíbrio entre consumo de energia, utilização de recursos e latência de *hardware*.

A Figura 11 mostra o esquemático do projeto. Ele aceita como entrada uma fonte de *streaming* de *pixels*, como uma câmera, e é fornecido três saídas, que são índices de pontuação de classe com três níveis de probabilidade: alta (*HIGH*), média (*MID*) e baixa (*LOW*).

Figura 11 – Esquemático de uma CNN implementada para a execução em FPGA

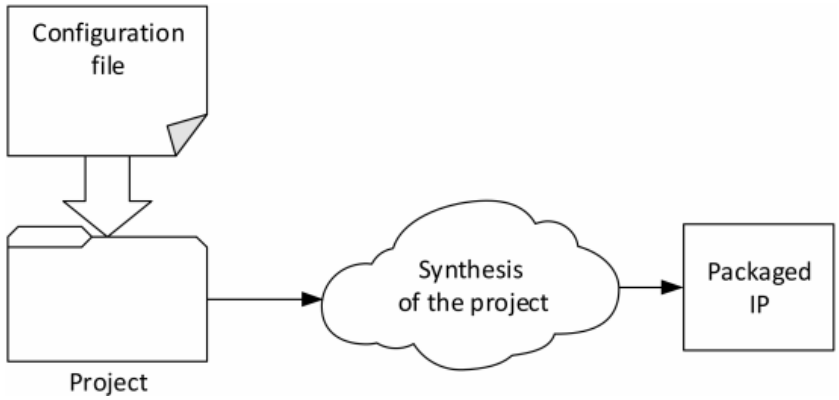


Fonte: Ram *et al.* (2020)

Baptista *et al.* (2019) fornecem uma plataforma para emular uma solução CNN genérica baseada em parâmetros em um FPGA. O usuário escreve um arquivo de configuração para definir a CNN desejada. Nesse arquivo é especificada a topologia, possibilitando a implementação de qualquer CNN que adote como camadas: convolução, *pooling* e *fully connected*. A plataforma reorganiza as informações do arquivo feito pelo projetista para obter a CNN descrita. É construída uma CNN usando arquitetura de fluxo, com as funções de ativação como função ReLU e *Softmax* implementadas no FPGA e as camadas sendo independentes uma das outras. Após a síntese do projeto o *design* é empacotado como um IP no *Vivado* HLS. O processo desde o preenchimento do arquivo de configuração pelo designer até o bloco IP gerado pela síntese do projeto pode ser visto no diagrama da Figura 12.

Phu *et al.* (2019) projetaram um *script* gerador de código *Verilog* para criar um

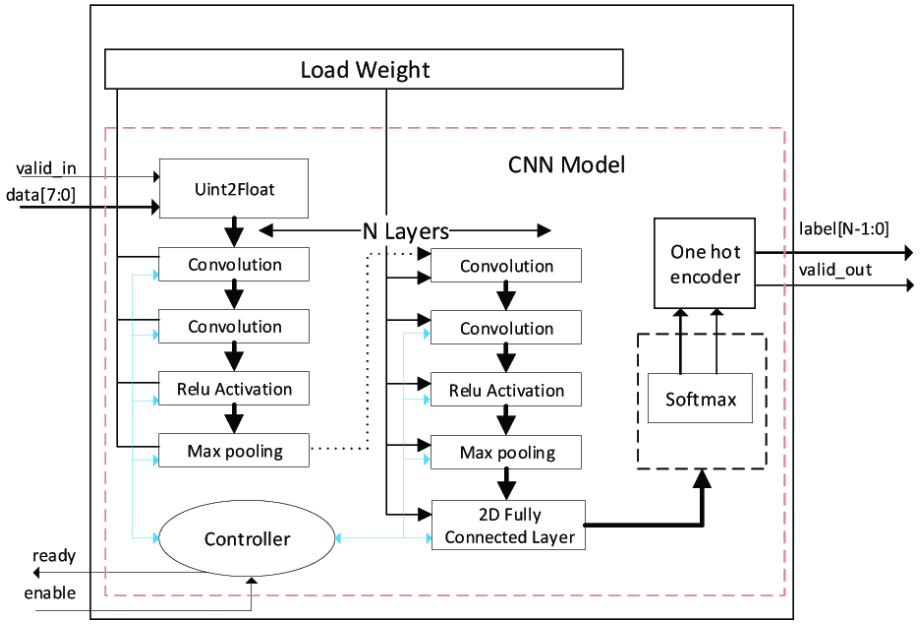
Figura 12 – Diagrama do processo de geração do IP



Fonte: Baptista *et al.* (2019)

CNN Core IP de *stream* totalmente *pipeline* em FPGA usando aritmética de ponto flutuante de precisão simples. Os autores foram capaz de minimizar as perdas de cálculo, reduzir o atraso de transmissão e a carga de processamento no servidor. O *CNN Core IP* contém várias camadas convolucionais, intercaladas por camadas de ativação e agrupamento, e uma camada totalmente conectada, podendo ser reconfigurado alterando o número de camadas, tamanho do *kernel*, tamanho de entrada e número de nós de saída. Para configurar o modelo CNN antes de gerar o código *Verilog* HDL usa-se um arquivo *json*. A Figura 13 mostra o diagrama do *CNN Core IP* apresentado pelos autores.

Figura 13 – Diagrama do *CNN Core IP*



Fonte: Phu *et al.* (2019)

Os trabalhos apresentados tem em comum a implementação de toda a rede CNN em um dispositivo FPGA, diferente deste apresentado que tem o objetivo de implementar

em FPGA apenas a área crítica identificada da rede. O trabalho Ram *et al.* (2020) mostrou uma implementação para acelerar o estágio de inferência da rede sem ser automatizado e utilizando *Verilog*, diferente deste apresentado que propõe um sistema automático utilizando HLS. O trabalho Baptista *et al.* (2019) emula uma rede CNN configurada pelo usuário utilizando HLS, e o trabalho Phu *et al.* (2019) projetou um *script* para gerar uma rede CNN em *Verilog* configurada por arquivo *json* feito pelo usuário. Esses dois últimos trabalhos são semelhantes a este apresentado em propor um sistema configurado pelo desenvolvedor da rede CNN, porém é proposto aqui que o desenvolvedor utilize o código da rede implementada em *python*.

O Quadro 1 mostra uma comparação entre os trabalhos apresentados. É destacado na segunda coluna como é o sistema implementado, se ele é um sistema manual ou automático. Caso ele seja automático é mostrado na terceira coluna qual a entrada do sistema, se for manual essa coluna não se aplica ao trabalho. E na quarta coluna é mostrado qual a técnica de implementação utilizada no desenvolvimento.

Quadro 1 – Comparativo dos trabalhos relacionados

Trabalho	**Geração**	**Entrada do sistema**	**Técnica de implementação**
(RAM *et al.*, 2020)	Manual	Não se aplica	*Verilog* HDL
(PHU *et al.*, 2019)	Automático	Arquivo *json*	*Verilog* HDL
(BAPTISTA *et al.*, 2019)	Automático	Arquivo com parâmetros	HLS
Trabalho apresentado	Automático	Rede CNN em *python*	HLS

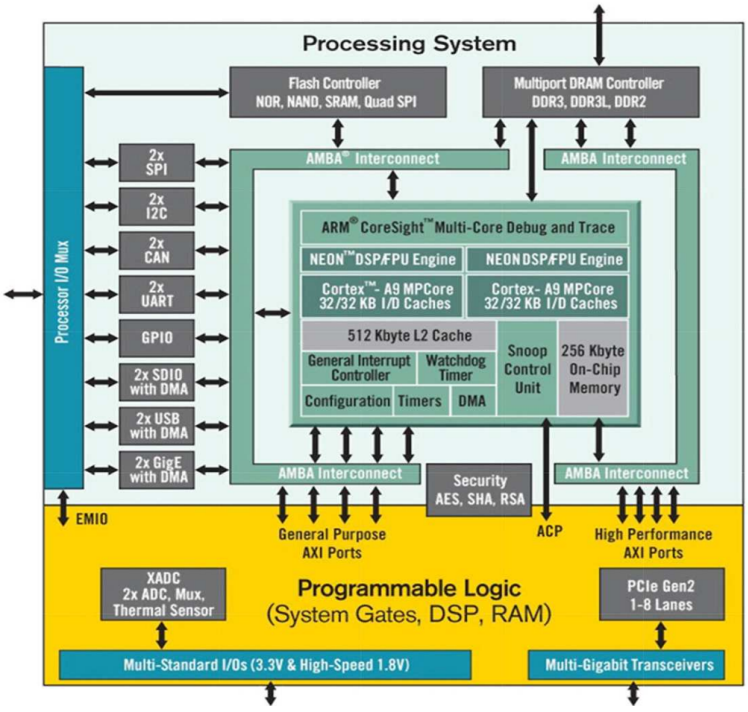
Fonte: Elaborado pelo autor

4 PROCEDIMENTOS METODOLÓGICOS

Este Capítulo apresenta as informações sobre o sistema proposto e os passos executados no seu desenvolvimento. O sistema recebe como entrada uma rede implementada e treinada em *python* e tem como saída uma rede CNN com componentes convertidos em código de descrição de *hardware* que podem ser executados em um FPGA[1].

A plataforma escolhida para a execução deste trabalho é a plataforma *Zynq APSoC*. A família *Zynq* é baseada na arquitetura *Xilinx All Programmable System-on-Chip (AP SoC)*, que integra um processador *dual-core ARM Cortex-A9* com a lógica *Xilinx 7-series Field Programmable Gate Array*. O *Zynq APSoC* é dividido em dois subsistemas distintos: *Processing System* (PS) e *Programmable Logic* (PL) (DIGILENT, 2018). A Figura 14 mostra uma visão geral da arquitetura do *Zynq APSoC*.

Figura 14 – Arquitetura *Zynq APSoC*



Fonte: DIGILENT (2018)

[1] Projeto disponível em https://github.com/VictoriaMaia/Arq-SoC-CNN-FPGA

4.1 Rede CNN Modelo

Dada uma rede CNN pré-treinada, o sistema apresentado aqui implementa automaticamente uma arquitetura SoC, específica para a plataforma *Zynq*, utilizando *co-design*. Portanto, com o objetivo de testar o sistema, foi desenvolvida uma pequena rede CNN utilizando uma *Application Programming Interface* (API) que permite a implementação de redes neurais chamada *Keras*[2].

Como o processo desenvolvido é genérico e não depende do problema a ser resolvido, só são necessárias as informações da arquitetura e os pesos obtidos ao fim do treinamento da rede. Por isso, foi escolhido um problema bem conhecido na área de Aprendizagem de Máquina: o reconhecimento de dígitos manuscritos. Esse problema é um problema de classificação contendo 10 classes representando os dígitos de 0 a 9.

As imagens de dígitos utilizadas foram retiradas do banco de dados MNIST[3]. Esse banco de dados é amplamente conhecido e utilizado nas implementações de resoluções desse problema. Ele contém 60000 imagens de treino e 10000 imagens de teste, sendo todas em escala de cinza e de tamanho 28x28 *pixels*. Alguns exemplos de imagens retiradas do banco de dados pode ser visto na Figura 15.

Figura 15 – Exemplo de imagens retiradas do banco de dados MNIST *digits*

Fonte: MNIST

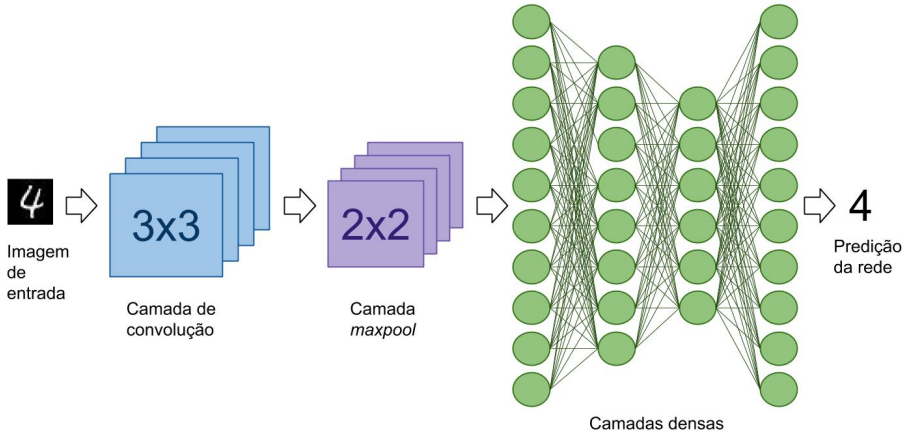
A rede implementada contém as seguintes configurações e é representada visualmente na Figura 16.

- 1 camada de convolução com 4 filtros de tamanho 3x3;
- 1 camada de *maxpool* de tamanho 2x2;
- 3 camadas densas utilizando a função de ativação *relu* com 10, 8 e 6 neurônios;
- 1 camada densa utilizando a função de ativação *softmax* com 10 neurônios;

Foram utilizadas 10000 imagens para treino e 8000 imagens para teste para ser possível vários treinos em tempo útil. Na fase de treino foi utilizado 10 épocas com 1000 imagens para o treino de cada época. Como trata-se de uma rede para testes, não foi feito nenhum refinamento na rede para melhorar seu desempenho.

[2] https://keras.io/about/
[3] http://yann.lecun.com/exdb/mnist/

Figura 16 – Rede CNN implementada

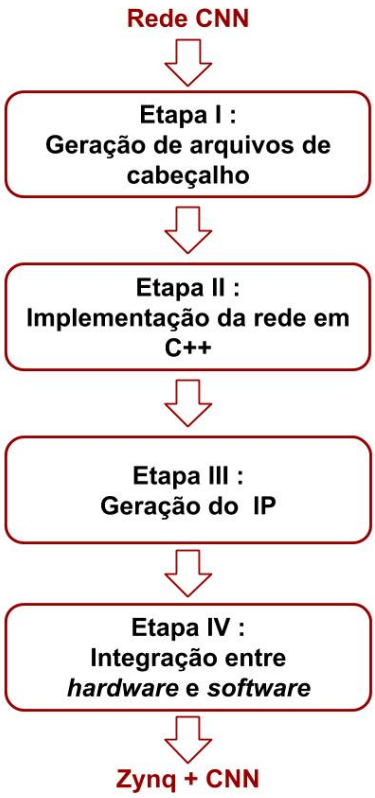


Fonte: Elaborado pelo autor.

4.2 Geração da Arquitetura SoC

O processo de geração da arquitetura SoC inclui 4 etapas principais: I) Geração de arquivos de cabeçalho, II) Implementação da rede em C++, III) Geração do IP e, por último, IV) Integração entre *hardware* e *software*. Essas etapas são mostradas na Figura 17.

Figura 17 – Diagrama representando as etapas de desenvolvimento do sistema implementado

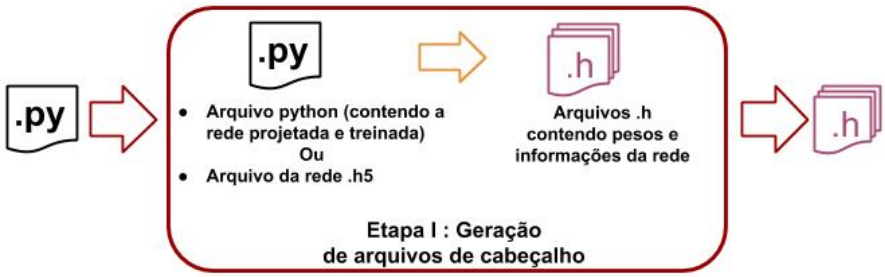


Fonte: Elaborado pelo autor.

4.2.1 Etapa I - Geração de de arquivos de cabeçalho

A etapa I do desenvolvimento é responsável por gerar os arquivos de informação da rede, sendo representada na Figura 18. Esta etapa recebe como entrada o código *python* com uma rede CNN implementada e treinada, e retorna as informações mais importantes em arquivos de cabeçalho.

Figura 18 – Etapa I - Geração de de arquivos de cabeçalho



Fonte: Elaborado pelo autor.

Nesta etapa foi desenvolvido um *script* que armazena os pesos, os valores de *bias* das camadas e as informações sobre a arquitetura, produzindo um arquivo de cabeçalho (.h) para o código C++ equivalente. Ele pode ser executado de duas formas, chamado como uma função do *python* durante a implementação da rede, ou recebendo como parâmetro um arquivo .h5. Esse arquivo (.h5) é utilizado pelo *Keras* para o armazenamento da arquitetura e parâmetros da rede[4].

Durante a implementação da rede utilizando o *Keras*, é gerado um modelo com todas as informações da arquitetura. Esse modelo é um objeto da classe *Sequential*[5]. O *script* acessa o modelo, lê as informações da rede e as salva nos arquivos de cabeçalho. Foi armazenado além dos valores de peso, as seguintes informações sobre a arquitetura:

- Quantidade de linhas e colunas da imagem;
- Quantidade de filtros convolucionais;
- Quantidade de linhas e colunas dos filtros convolucionais;
- Quantidade de linhas e colunas do filtro *pooling*;
- Quantidade de camadas densas;
- O nome das funções de ativação de cada camada densa;
- Quantidade de neurônios de cada camada densa;

[4] https://www.tensorflow.org/guide/keras/save_and_serialize
[5] https://keras.io/api/models/sequential/

4.2.2 Etapa II - Implementação da rede em C++

Para gerar código em *hardware* e executar a rede CNN na plataforma *Zynq APSoC* é necessário que a rede esteja implementada em C++. A etapa II é responsável pela conversão da rede em *python* para uma rede em C++, sendo representada na Figura 19.

Figura 19 – Etapa II - Implementação da rede em C++

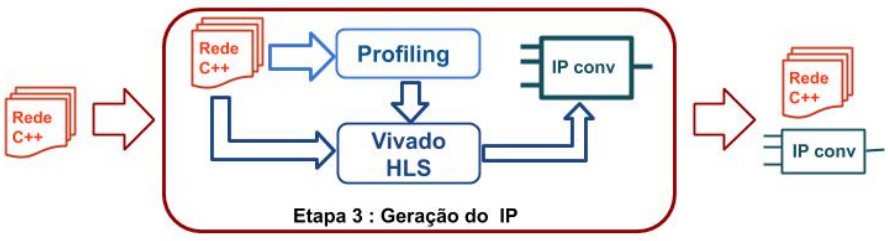
Fonte: Elaborado pelo autor.

Esta etapa recebe como entrada os arquivos de cabeçalho gerados na etapa I e retorna o projeto da rede implementada em C++. Como a rede em C++ necessita ser equivalente à rede em *python* foram implementados manualmente arquivos genéricos .cpp contendo instruções que executam as mesmas funcionalidades das camadas da rede. A partir das informações nos arquivos .h os arquivos .cpp conseguem gerar o mesmo resultado de predição da rede executada em *python*.

4.2.3 Etapa III - Geração do IP

A etapa III é responsável pela conversão de partes do código C++ em código de descrição de *hardware*, sendo representada na Figura 20.

Figura 20 – Etapa III - Geração do IP



Fonte: Elaborado pelo autor.

Esta etapa recebe a rede implementada em C++ e retorna o um componente IP, da

área crítica do código. Nesta etapa é feito um *profiling* do código em C++ para verificar qual é a área crítica, a camada que demora mais tempo de execução. Tendo essa informação é feito no programa Xilinx Vivado HLS a conversão do código C++ da área crítica em VHDL.

A versão do *Vivado* HLS que foi utilizada é a 2016.4. Essa ferramenta é responsável por transformar uma especificação C, C++ ou *SystemC* em arquivos de implementação RTL nas linguagens *Verilog* ou VHDL que são sintetizáveis em um *Xilinx* FPGA (XILINX, 2020). Para os testes de tempo utilizou-se elementos da biblioteca *chrono*[6], inserindo o elemento *high resolution clock* no começo e no fim de cada camada a ser analisada, o qual é responsável por fornecer o valor de tempo atual com maior precisão.

Os resultados dos testes de *profiling* estão na Tabela 2 na seção 5. A partir das informações mostradas nessa tabela pode-se perceber que a camada de convolução é a responsável pela maior parte do tempo de execução, portanto ela é a área crítica do código. Assim, o IP gerado irá executar as operações da camada de convolução.

4.2.4 Etapa IV - Integração entre hardware e software

A etapa IV é responsável por gerar a saída do sistema, sendo representada na Figura 21.

Figura 21 – Integração entre *hardware* e *software*

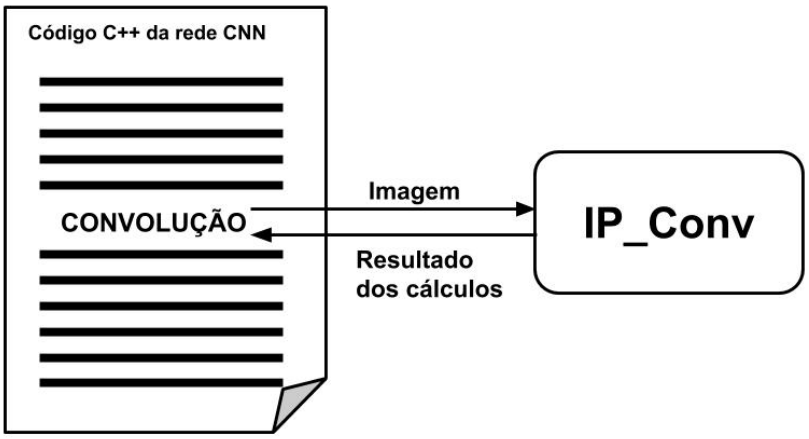
Fonte: Elaborado pelo autor.

Esta etapa recebe como entrada a rede em C++ e o IP da convolução e retorna a arquitetura com *co-design* que executa a rede CNN implementada na plataforma *Zynq APSoC*. Nesta etapa é adicionada ao código C++ da rede a chamada de ativação do IP gerado. O sistema irá executar o código C++ até chegar o momento da camada de convolução. Nessa parte do código o IP será ativado através de uma chamada de função passando uma imagem como parâmetro. O IP irá efetuar as instruções programadas. Ao fim do processo, o código C++ irá

[6] http://www.cplusplus.com/reference/chrono/

receber os resultados dos cálculos e continuará a sua execução. A Figura 22 mostra visualmente o processo da ativação do IP durante a execução do código da rede.

Figura 22 – Processo de ativação do IP durante a execução do código C++ da rede



Fonte: Elaborado pelo autor.

A arquitetura final está representada na Figura 23, onde temos a entrada do sistema, responsável por receber as imagens, a saída do sistema, responsável por mostrar a predição da rede, o processador *Zynq* responsável pela execução do *software* e o armazenamento das informações, e o IP_Conv, responsável por executar os cálculos da convolução.

Figura 23 – Arquitetura do sistema final

Fonte: Elaborado pelo autor.

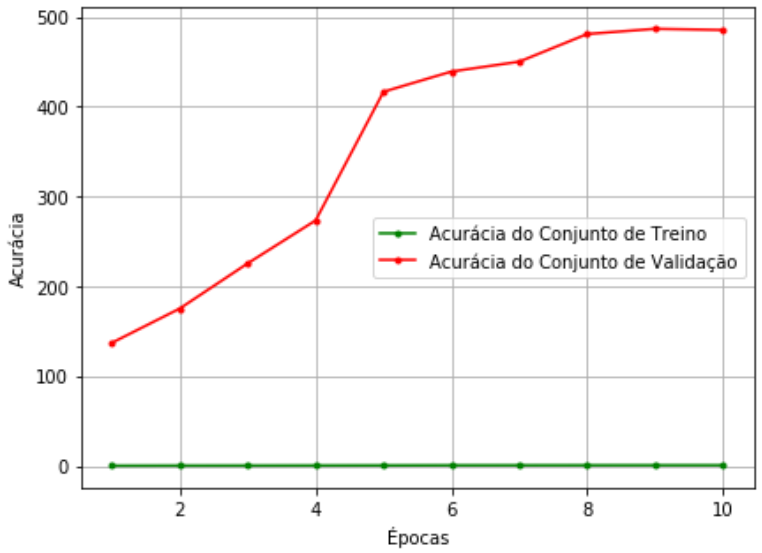
5 RESULTADOS

Este Capítulo apresenta os resultados da avaliação da rede em *python* implementada na etapa I, o resultado da implementação da rede em C++ na etapa II, o resultado do *profiling* feito da rede em C++ na etapa III e o resultado do teste de tempo feito para medir a eficiência do IP gerado na etapa III.

5.1 Avaliação da rede em *python*

Para avaliar a rede implementada em *python* na etapa I, foi utilizada a função de custo *categorical cross entropy* e a métrica avaliativa acurácia. Podemos ver nas Figuras 24 e 25, os gráficos com os valores de acurácia e *cross entropy loss* respectivamente, do conjunto de dados de treino e validação do modelo calculados durante o treinamento da rede.

Figura 24 – Gráfico com valores de acurácia do treino e validação durante o treinamento da rede

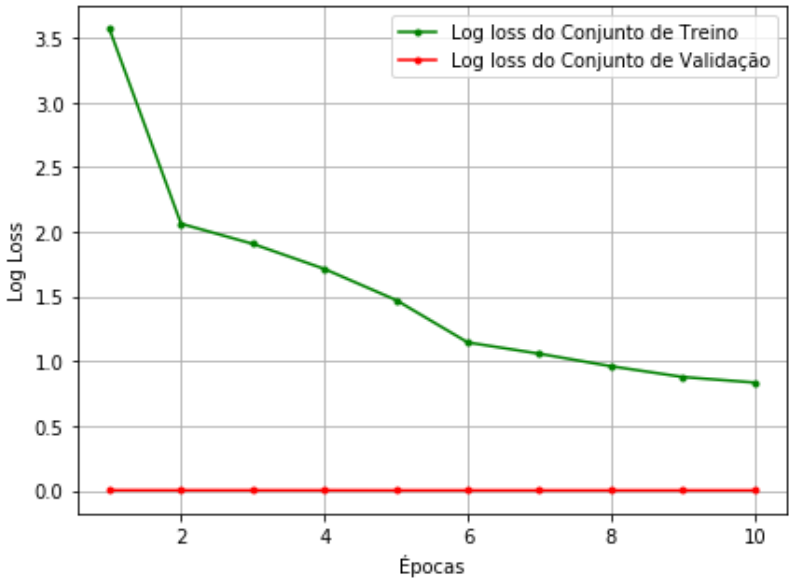


Fonte: Elaborado pelo autor.

No gráfico da Figura 24 podemos ver que a acurácia do conjunto de validação tem um aumento crescente a medida que o número de épocas aumenta. O que significa que a rede tem bons valores de predição ao fim do treinamento, ou seja, ela está conseguindo prever corretamente os valores de dígitos. Enquanto o valor no conjunto de treino permanece tendo valores pequenos aproximados a zero.

No gráfico da Figura 25 podemos perceber que ao longo do treinamento o conjunto de

Figura 25 – Gráfico com valores de *cross entropy loss* do treino e validação durante o treinamento da rede



Fonte: Elaborado pelo autor.

treino vai tendo seu valor de *cross entropy loss* menor, o que significa que a rede esta aprendendo e melhorando a predição a medida que o número de épocas aumenta. Enquanto o valor no conjunto de validação tem valores pequenos, próximos de zero.

Na Tabela 1 podemos ver a avaliação da rede utilizando as mesmas funções avaliativas usadas no treino.

Tabela 1 – Resultados da avaliação da rede

Métrica de avaliação	Valor do resultado
Acurácia	0.88840
Cross entropy loss	0.94145

Fonte: Elaborado pelo autor.

Podemos perceber que a rede implementada tem um desempenho razoável. Ela se mostra com um bom valor de acurácia, mas o valor do *cross entropy loss* está alto. Como a rede CNN foi implementada com o objetivo de testar o sistema, não se teve o compromisso com uma maior eficiência de predição e não foi feito um refinamento da rede.

5.2 Resultado da rede implementada em C++

A Figura 26 mostra a comparação das saídas da rede em *python* e da rede em C++. Sendo perceptível que a conversão do código *python* para o código em C++ foi feito corretamente, já que elas geram os mesmos valores da última camada densa, onde tem os valores de probabilidade para cada neurônio de saída, e o mesmo valor de predição para as imagens de teste.

Figura 26 – Comparação das predições feitas em *python* e em C++

```
[[9.9758196e-01 1.0275129e-11 2.8200890e-13 1.4491323e-16 5.3583797e-17 3.7580075e-10 1.7803187e-09 9.1398615e-05
2.3233672e-03 3.1898667e-06]]
predição : [0]
[[1.4914139e-19 9.9948490e-01 2.5758642e-09 1.1131055e-09 7.7069071e-09 4.6252483e-04 4.5645553e-05 6.7899018e-13
3.0101955e-07 6.6518442e-06]]
predição : [1]
[[1.8970161e-08 1.6856330e-02 9.6181238e-01 3.7527999e-03 4.6998710e-05 1.6803419e-02 2.5299931e-04 6.7623238e-05
3.9195913e-04 1.5573069e-05]]
predição : [2]
[[8.0609119e-09 2.1877674e-19 3.2256722e-34 2.8514452e-22 9.4275855e-26 3.3844394e-10 6.5858481e-22 9.9974149e-01
1.8092825e-08 2.5844760e-04]]
predição : [7]
```

(a) Predição feita em *python*

```
vi@hakunamatata:/media/vi/D2029DF5029DDEB3/UFC/TCC/cnn/implementaçãoEmC$ ./app imagens/imageTest0.jpg
0.997582 1.02752e-11 2.82011e-13 1.44915e-16 5.35844e-17 3.75803e-10 1.78033e-09 9.1399e-05 0.00232337 3.18988e-06
predição : 0
vi@hakunamatata:/media/vi/D2029DF5029DDEB3/UFC/TCC/cnn/implementaçãoEmC$ ./app imagens/imageTest1.jpg
1.49141e-19 0.999485 2.57587e-09 1.1131e-09 7.70692e-09 0.000462523 4.56457e-05 6.78986e-13 3.01019e-07 6.65183e-06
predição : 1
vi@hakunamatata:/media/vi/D2029DF5029DDEB3/UFC/TCC/cnn/implementaçãoEmC$ ./app imagens/imageTest2.jpg
1.89701e-08 0.0168563 0.961812 0.0037528 4.69986e-05 0.0168034 0.000252999 6.76232e-05 0.000391959 1.5573e-05
predição : 2
vi@hakunamatata:/media/vi/D2029DF5029DDEB3/UFC/TCC/cnn/implementaçãoEmC$ ./app imagens/imageTest7.jpg
8.06087e-09 2.18778e-19 3.2257e-34 2.85148e-22 9.4277e-26 3.38446e-10 6.58587e-22 0.999742 1.80927e-08 0.000258448
predição : 7
vi@hakunamatata:/media/vi/D2029DF5029DDEB3/UFC/TCC/cnn/implementaçãoEmC$
```

(b) Predição feita em C++

Fonte: Elaborado pelo autor.

5.3 Resultado do *profiling* feito da rede em C++

Os testes de *profiling* foram executados em um computador com as seguintes especificações:

- Processador Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz
- Memória de 8GB DDR4
- Sistema Operacional Linux Ubuntu 18.04

Foram feitos três testes. Cada teste foi executado 10 vezes. Foi calculado a média do tempo de execução e calculado a porcentagem de tempo gasto de cada camada. O primeiro teste calculou o tempo de execução para 1 imagem como entrada. O segundo teste calculou o tempo

de execução de 1000 imagens. O terceiro teste calculou o tempo de execução de 10000 imagens. O resultado dos testes pode ser visto na Tabela 2, que mostra o percentual do tempo total gasto em cada camada.

Tabela 2 – Resultados do *profiling* de cada camada da rede nos três testes

Teste	Convolucional	*Maxpool*	*Dense* 0	*Dense* 1	*Dense* 2	*Dense* 3
1 imagem	61%	14,76%	17,46%	1,72%	1,10%	3,93%
1000 imagens	64,30%	14%	16,43%	1,43%	1,10%	2,71%
10000 imagens	64,41%	13,84%	16,48%	1,43%	1,11%	2,71%

Fonte: Elaborado pelo autor.

Podemos perceber que, mesmo com uma grande diferença no número de predições necessárias, as camadas apresentam uma pequena variação de valores nos três testes. Os percentuais de tempo ficaram muito próximos, variando entre 61% e 64,41% na camada de convolução, que é a que demanda a maior parte do tempo.

5.4 Resultado do teste de tempo feito sobre o IP gerado

O teste de tempo foi feito no Vivado HLS, executando a rede 100 vezes. Foi computado o tempo de execução da camada convolucional feita em *software* e da camada convolucional convertida em *hardware* através da criação do IP. Após as medidas de tempo foi calculada a média. O resultado dos testes pode ser visto na Tabela 3.

Tabela 3 – Comparação entre o tempo médio de computação da rede, com a convolução executada por *hardware* e por *software*

Teste	Convolucional	Demais Camadas	Tempo Total	*Speedup*
software	312 μs	199 μs	511 μs	1,00
hardware	20.5 μs	199 μs	219.5 μs	2,33

Fonte: Elaborado pelo autor.

Podemos perceber que o IP gerado consegue executar em um tempo bem menor do que o código em C++, executando a convolução 15,2 vezes mais rápido do que por *software*. Isto indica que a solução proposta apresenta bons resultados e mostra a viabilidade de execução da rede CNN em um FPGA de forma eficiente.

Por fim, ressalta-se, ainda, que os valores apresentados correspondem à rede usada como modelo, podendo variar para outas redes. Acredita-se, porém, que os resultados tenham um comportamento similar em outros cenários.

6 CONCLUSÃO E TRABALHOS FUTUROS

Nas diversas aplicações envolvendo o uso de CNNs há o aumento de interesse dessa rede no desenvolvimento de dispositivos inteligentes. Dentre as soluções para a execução de Redes Neurais Convolucionais em sistemas embarcados existe as alterações na estrutura da rede ou a utilização de algoritmos que auxiliem a execução dessa rede nos FPGAs.

Neste trabalho foi apresentado um sistema que gera uma arquitetura *co-design* para a execução de redes CNNs implementadas em *python*. Possibilitando a execução dessa rede em sistemas embarcados que contenham em seu *hardware* um FPGA.

Durante o desenvolvimento do sistema foram feitos testes de tempo no código da rede CNN para produzir um IP da área crítica, com a intenção de utilizar as vantagens de desempenho do FPGA. Partindo do pressuposto que um código feito em *hardware* executa mais rápido que um código executado apenas em *software*, o trabalho mostrou como resultado uma execução da rede CNN de forma mais eficiente na arquitetura gerada.

Nos testes de tempo foi descoberto que a camada convolucional gasta em média 63% do total de tempo de execução na rede usada como modelo. Foi desenvolvido, então, um IP com as instruções da camada convolucional. Foi feito um teste de tempo entre a convolução sendo executada apenas em *software* e a convolução executada com o IP. Nesse teste foi notado que o IP executou aproximadamente 15,2 vezes mais rápido que a convolução em *software*.

Assim, no final do desenvolvimento obteve-se uma arquitetura SoC com a rede implementada em C++ com componente de convolução implementado em *hardware*. Como trabalhos futuros temos a realização de testes em um sistema final integrando um FPGA a um sistema de captura de imagens. Além disso, outros modelos de CNN podem ser testados, bem como podemos criar *script* para automatizar as etapas que fazem uso direto do Vivado[1] e Vivado HLS.

[1] https://www.xilinx.com/products/design-tools/vivado.html

REFERÊNCIAS

BAILEY, D. G. **Design for embedded image processing on FPGAs**. Singapore: John Wiley & Sons, 2011. ISBN 978-0-470-82850-2.

BAPTISTA, D.; MORGADO-DIAS, F.; SOUSA, L. A platform based on hls to implement a generic cnn on an fpga. In: 2019 INTERNATIONAL CONFERENCE IN ENGINEERING APPLICATIONS (ICEA). Sao Miguel, Portugal: IEEE, 2019. p. 1–7. ISSN 978-1-7281-2962-4.

BONNARD, J.; ABDELOUAHAB, K.; PELCAT, M.; BERRY, F. On building a cnn-based multi-view smart camera for real-time object detection. **Microprocessors and Microsystems**, v. 77, p. 103177, 2020.

CAO, J.; ZHANG, Q.; SHI, W. **Edge computing:** a primer. Cham: Springer, 2018. ISBN 978-3-030-02083-5.

CARDOSO, J. M. P.; DINIZ, P. C. **Compilation techniques for reconfigurable architectures**. New York: Springer, 2009. ISBN 978-0-387-09670-4.

COUSSY, P.; GAJSKI, D.; MEREDITH, M.; TAKACH, A. An introduction to high-level synthesis. **Design & Test of Computers, IEEE**, v. 26, p. 8 – 17, 09 2009.

DIGILENT. **Zybo Z7 board reference manual**. 2018. Disponível em: https://reference.digilentinc.com/_media/reference/programmable-logic/zybo-z7/zybo-z7_rm.pdf. Acesso em: 03 Set. 2020.

FAN, S.; LI, J.; ZHANG, Y.; TIAN, X.; WANG, Q.; HE, X.; ZHANG, C.; HUANG, W. On line detection of defective apples using computer vision system combined with deep learning methods. **Journal of Food Engineering**, v. 286, 2020.

GAJSKI, D. D.; ABDI, S.; GERSTLAUER, A.; SCHIRNER, G. **Embedded system design:** modeling, synthesis and verification. US: Springer, 2009. ISBN 978-1-4419-0504-8.

HAYKIN, S. **Redes neurais:** princípios e prática. 2. ed. Porto Alegre: Bookman, 2007. ISBN 978-8573077186.

HEINRICH, A.; GüTTLER, F.; SCHENKL, S.; WAGNER, R.; TEICHGRäBER, U.-M. Automatic human identification based on dental x-ray radiographs using computer vision. **Scientific Reports**, v. 10, n. 1, 2020.

JIMÉNEZ, M.; COUVERTIER, I.; PALOMERA, R. **Introduction to embedded systems:** using microcontrollers and the MSP430. New York: Springer, 2014. ISBN 1461431425.

KEATING, M. **The simple art of SoC design:** closing the gap between RTL and ESL. New York: Springer, 2011. ISBN 978-1-4419-8585-9.

KHAN, S.; RAHMANI, H.; SHAH, S. A. A.; BENNAMOUN, M. **A guide to convolutional Neural Networks for Computer Vision**. [S.l.]: Morgan & Claypool, 2018. ISBN 9781681730226.

MAHADEVASWAMY, U.; RAO, M. S.; VRUSHAB, S.; ANAGHA, C.; SANGAMESHWAR, V. Visual speech processing and recognition. **Advances in Intelligent Systems and Computing**, v. 1141, p. 481–491, 2021.

MAXFIELD, C. **FPGAs:** instant access. Oxford, Reino Unido: Elsevier, 2008. v. 1.

MITTAL, S. A survey of fpga-based accelerators for convolutional neural networks. **Neural computing and applications**, Springer, v. 32, n. 4, p. 1109–1139, fev. 2020. ISSN 0941-0643.

NEDJAH, N.; MOURELLE, L. de M. **Co-design for system acceleration:** a quantitative approach. US: Springer, 2007. ISBN 978-1-4020-5546-1.

PASRICHA, S.; DUTT, N. **On-chip communication architectures:** system on chip interconnect. USA: Elsevier, 2008. ISBN 978-0-12-373892-9.

PHU, H. V.; TAN, T. M.; MEN, P. V.; HIEU, N. V.; CUONG, T. V. Design and implementation of configurable convolutional neural network on fpga. In: 2019 6th NAFOSTED CONFERENCE ON INFORMATION AND COMPUTER SCIENCE (NICS). Hanoi, Vietnam: IEEE, 2019. p. 298–302. ISSN 978-1-7281-5163-2.

RAM, R. G. S.; CHATURVEDI, N.; SAURAV, S.; SINGH, S. An fpga based hardware accelerator for classification of handwritten digits. In: 18th INTERNATIONAL CONFERENCE ON INTELLIGENT SYSTEMS DESIGN AND APPLICATIONS. Suíça, Cham: Springer International Publishing, 2020. v. 940, p. 945–954. ISBN 978-3-030-16657-1.

RASCHKA, S.; MIRJALILI, V. **Python machine learning:** Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow. 2. ed. Birmingham: Packt Publishing Ltd, 2017.

SZELISKI, R. **Computer Vision:** algorithms and applications. London: Springer, 2011.

VAHID, F. **Sistemas digitais:** projeto, otimização e HDLs. Porto Alegre: Bookman, 2008. ISBN 978-85-7780-190-9.

XILINX. **Vivado design suite user guide:** High-Level Synthesis. 2020. Disponível em: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf. Acesso em: 07 Ago. 2020.

YU, W.; KIM, S.; CHEN, F.; CHOI, J. Pedestrian detection based on improved mask r-cnn algorithm. **Advances in Intelligent Systems and Computing**, v. 1197 AISC, p. 1515–1522, 2021.

ZHANG, L.; BU, X.; LI, B. Xnorconv: Cnns accelerator implemented on fpga using a hybrid cnns structure and an inter-layer pipeline method. **IET Image Processing**, v. 14, n. 1, p. 105–113, 2020.