UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

SAMMUEL RAMOS DA SILVA
Orientador: Prof. Me. Vinicius Antonio de Oliveira Martins

TINN
UMA ARQUITETURA PARA ACELERAÇÃO DE REDES NEURAIS EM FPGAS DE BAIXA DENSIDADE

Ouro Preto, MG
2022

UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

SAMMUEL RAMOS DA SILVA

TINN
UMA ARQUITETURA PARA ACELERAÇÃO DE REDES NEURAIS EM FPGAS DE BAIXA DENSIDADE

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Me. Vinicius Antonio de Oliveira Martins

Ouro Preto, MG
2022

SISBIN - SISTEMA DE BIBLIOTECAS E INFORMAÇÃO

S586t Silva, Sammuel Ramos da.
TINN [manuscrito]: uma arquitetura para aceleração de redes neurais em FPGAS de baixa densidade. / Sammuel Ramos da Silva. - 2022.
42 f.: il.: color., gráf., tab..

Orientador: Prof. Me. Vinicius Antonio de Oliveira Martins.
Monografia (Bacharelado). Universidade Federal de Ouro Preto. Instituto de Ciências Exatas e Biológicas. Graduação em Ciência da Computação .

1. Hardware. 2. Inteligência Artificial. 3. Redes neurais (Computação). I. Martins, Vinicius Antonio de Oliveira. II. Universidade Federal de Ouro Preto. III. Título.

CDU 004

Bibliotecário(a) Responsável: Luciana De Oliveira - SIAPE: 1.937.800

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E BIOLOGICAS
DEPARTAMENTO DE COMPUTACAO

FOLHA DE APROVAÇÃO

SAMMUEL RAMOS DA SILVA

TINN - UMA ARQUITETURA PARA ACELERAÇÃO DE REDES NEURAIS EM FPGAS DE BAIXA DENSIDADE

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal
de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Ciência da Computação

Aprovada em 21 de Junho de 2022.

Membros da banca

Vinicius Antonio de Oliveira Martins (Orientador) - Mestre - Universidade Federal de Ouro Preto
Fernando Cortez Sica (Examinador) - Doutor - Universidade Federal de Ouro Preto
Carlos Frederico M. da Cunha Cavalcanti (Examinador) - Doutor - Universidade Federal de Ouro Preto

Vinicius Antonio de Oliveira Martins, Orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 21/06/2022.

Documento assinado eletronicamente por **Vinícius Antônio de Oliveira Martins**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 21/06/2022, às 13:09, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **0346391** e o código CRC **A8F741B7**.

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35400-000
Telefone: 3135591692 - www.ufop.br

Dedico este trabalho aos meus pais que sempre me incentivaram a estudar e não mediram esforços para investir nos meus estudos.

Agradecimentos

O desenvolvimento deste trabalho só foi possível graças a participação de pessoas fundamentais em minha vida. Gostaria de agradecer:

Aos meus pais, Vagner Lúcio da Silva e Luiza Ramos da Silva por sempre me fornecer todo apoio, incentivo e críticas ao longo de toda a vida.

Agradeço ao meu orientador, Vinicius Antonio de Oliveira Martins, por toda a paciência, dedicação e conselhos dados por inúmeras horas dedicadas para a conclusão deste trabalho.

Agradeço ao professor Eduardo José da Silva Luz, pelo suporte e apoio durante a realização deste trabalho.

Agradeço ao meu amigo Gustavo Lucas e Stephany Mary por nunca me deixar desistir, por toda energia positiva, apoio, críticas e por estarem sempre ao meu lado.

Agradeço aos professores, em especial: Guilherme Tavares de Assis, Dayanne Gouveia Coelho e Amanda Sávio Nascimento e Silva, que através do método de ensino, paixão pela ciência e senso de humor, me moldaram em uma pessoa, e um cientista melhor.

Por fim, mas não menos importante, agradeço a todos os demais colegas e amigos que tive a oportunidade de compartilhar essa experiência, em especial aos remanescentes do 17.2, que transformaram momentos de fracassos e de aprendizagem em piadas. Sem vocês essa etapa seria de extremo tédio.

Resumo

Os avanços recentes em *hardware* fizeram com que o uso de Redes Neurais Convolucionais fossem adotadas para solução de problemas em diversos sistemas, como segurança pública ou privada, aviação, carros autônomos e *smartphones*. Muita dessas aplicações necessitam de acesso a unidades de processamento de alto nível, como *Graphic Processing Unit*, para realização dos cálculos matemáticos, o que acaba, em muitas vezes, sendo muito custoso em relação ao preço e ao consumo de energia. Avanços recentes na execução de redes neurais em dispositivos reconfiguráveis (FPGAs), obtiveram um bom desempenho, levando em consideração a quantidade de recursos disponíveis, consumo energético, e preço. Assim, este trabalho tem por objetivo implementar uma plataforma para aceleração de redes neurais convolucionais em FPGAs.

Abstract

Recent advances in hardware have made the use of Convolutional Neural Networks adopted to solve problems in various systems, such as public or private security, aviation, autonomous cars, and smartphones. Many of these applications require access to a high-level processing unit to carry out the mathematical calculations, which ends up being very costly in relation to the price and energy consumption. However, recent advances in the use of reconfigurable devices (FPGAs), aimed at running CNN, have achieved good performance with reduced costs in the area, energy consumption, and price. This work aims to implement, in reconfigurable platforms, experimental research regarding the use of these platforms for the execution of convolution neural networks.

Keywords: Hardware, Neural Networks, FPGA, Convolutional Neural Network, Artificial Inteligence.

Lista de Ilustrações

Figura 2.1 – Divisão de métodos para aprendizagem supervisionada e não-supervisionada 5
Figura 2.2 – Perceptron proposto por Rosenblatt na década de 1950 6
Figura 2.3 – Exemplo de um problema linearmente separável. Neste caso, o Perceptron têm de definir se uma cor é azul ou marrom. 7
Figura 2.4 – Neurônio e suas partes principais 8
Figura 2.5 – Exemplo de um produto de Hadamard entre duas matrizes 4×4. 8
Figura 2.6 – Exemplo de Convolução entre um IFM de dimensões $6 \times 6 \times 1$ e um filtro de dimensões $3 \times 3 \times 1$, com *stride* = 3 gerando uma imagem de saída com dimensões $2 \times 2 \times 1$. 9
Figura 2.7 – Exemplo de uma CNN, arquitetura da LeNet-5. A rede é composta por 7 camadas, sendo três de convoluções, duas de *pooling* e duas camadas *fully connected* 10
Figura 2.8 – Exemplo de *Pooling* entre um IFM de dimensões $4 \times 4 \times 1$ usando uma janela de dimensões 2×2. 11
Figura 2.9 – Gráficos da função *sigmoid*, $S(x)$, e ReLU, $r(x)$ 12
Figura 2.10–*Fully Connected layers* 12
Figura 2.11–Arquitetura genérica de uma FPGA 14
Figura 2.12–Arquitetura de um CLB 14
Figura 2.13–Exemplo de uma LUT com duas entradas sua arquitetura usada na implementação da função $f = \overline{a}b + a\overline{b} = a \oplus b$. 15
Figura 2.14–Estrutura de um Bloco DSP, com um multiplicador de complemento de dois 25×18, um acumulador de 48-bit e outras unidades. 16
Figura 2.15–Estrutura de um Bloco BRAM. 16
Figura 3.1 – Arquitetura proposta por (LIANG et al., 2020). 18
Figura 3.2 – Arquitetura do esquema *work-stealing* proposta por (SHEN et al., 2018). . . 18
Figura 3.3 – Arquitetura proposta por (VéSTIAS et al., 2019), nela podemos notar as divisões e blocos utilizados para a camada convolucional (Conv PE) e para a camada densa (FC PE). 19
Figura 3.4 – Arquitetura de reconfiguração proposta por (VENIERIS; BOUGANIS, 2019). 20
Figura 3.5 – Arquitetura da Lite-CNN proposta por (VéSTIAS et al., 2018). Nela podemos notar a comunicação entre os blocos e a memória externa, que recebe e envia pesos para o *Memory Buffer*. 20
Figura 3.6 – Arquitetura da TPU proposta por Jouppi et al. (2017) e usada como base em Fuhrmann (2018). 21
Figura 3.7 – Arquitetura proposta por Korol (2019). 22

Figura 3.8 – Exemplo do esquema "*ping-pong*"para comunicação entre camadas convolucionais e *Max-Pooling* Korol (2019). Enquanto um bloco de memoria recebe o resultado da camada de convolução, outro bloco envia os dados para a camada de *pooling*. 22
Figura 4.1 – Arquitetura Proposta 23
Figura 4.2 – Diagrama de bloco do *Kernel Memory* 24
Figura 4.3 – Diagrama de bloco do *Data Memory* 25
Figura 4.4 – Diagrama de bloco de uma unidade de multiplicação com maior filtro sendo $n-1 \times n-1$ 26
Figura 4.5 – Exemplo de *stride* de tamanho 2 e memória de tamanho 4. 27
Figura 4.6 – Exemplo de *stride* de tamanho 3 e memória de tamanho 4, sendo o bloco em amarelo o dado extra que será lido. 28
Figura 4.7 – Exemplo da operação de *shift* com *stride* igual a 2 e um filtro 4×4. O *current register* possuí os valores carregados na última operação realizada, como o *stride* é igual a 2, para completar 4 dados para realização da próxima operação é necessário o carregamento de somente outros 2 valores 28
Figura 4.8 – Tipo de instrução padrão 29
Figura 4.9 – Tipo de instrução de Peso 30
Figura 4.10–Tipo de instrução de *Stride* 30
Figura 5.1 – Modelo usado na avaliação da arquitetura 32
Figura 5.2 – Uso por bloco da Arquitetura 14×14 34
Figura 5.3 – Consumo de energia da arquitetura 14×14 proposta 34
Figura 5.4 – Velocidade teórica da arquitetura proposta em GOPS dependendo do tamanho da unidade de multiplicação 35
Figura 5.5 – Acurácia para a classificação de cada classe da mnist: TiNN x Pytorch . . . 36

Lista de Tabelas

Tabela 1.1 – Comparação entre dispositivos de borda e *Cloud computing* (VéSTIAS, 2019) 2
Tabela 4.1 – Lista de prioridade dos tipos de instruções e seus campos de bits 30
Tabela 4.2 – Sinais da interface AXI4-Lite (ARM, 2011) 31
Tabela 5.1 – Uso de recursos para a arquitetura 14x14 33
Tabela 5.2 – Uso de recursos de diferentes versões da Arquitetura proposta 35
Tabela 5.3 – Erro entre a implementação no Pytorch e no Hardware para os resultados da camada convolucional do modelo 36

Lista de Abreviaturas e Siglas

0SKP	*Zero Skipping*
BRAM	*Block Ram*
CCU	*Central Control Unit*
CLB	*Configurable Logic Block*
CNN	*Convolutional Neural Networks*
DL	*Deep Learning*
DNN	*Deep Neural Networks*
DSP	*Digital Signal Processor*
EWMM	*Element Wise Matrix Multiplication*
FF	*Flip-Flop*
FPGA	*Field Programmable Gate Array*
FSM	*Finite State Machine*
GFLOPs	*Giga Floating-point Operations per second*
GOPs	*Giga Operations per second*
GPU	*Graphics Processing Unit*
HDL	*Hardware Description Language*
IFM	*Input feature map*
IoT	*Internet of Things*
LUT	*Look-up Table*
MACC	*Multiply and Accumulate*
ML	*Machine Learning*
MLP	*Multi-Layer Perceptron*
MMCU	*Matrix Multiplication Control Unit*
MMU	*Matrix Multiplication Unit*

OFM	*Output feature map*
PE	*Processing Elements*
RAM	*Remote Access Memory*
RGB	*Red, Green and Blue*
RN	*Rede Neural*
RTL	*Register Transfer Level*
TOPs	*Tera Operations per second*
TPU	*Tensor Processing Unit*

Lista de Símbolos

$\odot$	Operador de multiplicação *element-wise* entre matrizes
$\oplus$	Operador Xor
$\sum$	Somatório
φ	Letra grega phi
ω	Letra grega minúscula omega
$\leftarrow$	Atribuição a esquerda
$\rightarrow$	Atribuição a direita
e	Constante de Euler
log	Operação logarítmica
$==$	Operação de igual

Sumário

1 Introdução **1**
1.1 Justificativa 2
1.2 Objetivos 3
1.3 Organização do Trabalho 3
2 Revisão Literatura **4**
2.1 Fundamentação Teórica 4
2.1.1 Machine Learning 4
2.1.1.1 *Deep learning* 6
2.1.1.2 Perceptron 6
2.1.1.3 *Element Wise Matrix Multiplication* (Produto de Hadamard) . 7
2.1.1.4 Convolução 8
2.1.1.5 *Convolution Neural Networks* (CNN) 9
2.1.2 Linguagem de Descrição de Hardware (HDL) 12
2.1.3 *Field Programmable Gate Array* (FPGA) 13
2.1.3.1 *Look-up Table* (LUT) 14
2.1.3.2 *Digital Signal Processor* (DSP) 15
2.1.3.3 *BlockRam* (BRAM) 15
2.1.4 Simulação e Síntese de Hardware 16
3 Trabalhos Relacionados **17**
4 Desenvolvimento do TiNN **23**
4.1 Arquitetura proposta do TiNN 23
4.2 Interface de Memórias 24
4.3 Unidade de Multiplicação de Matrizes 25
4.4 Acumuladores 26
4.5 Ativação 27
4.6 Unidades de Controle 27
4.7 Memória de Instruções - FIFO 29
4.8 Conjunto de Instruções 29
4.9 Interface AMBA AXI 30
5 Resultados **32**
5.1 Software Usados 32
5.2 Treinamento e Modelo 32
5.2.1 Treinamento 32
5.2.2 Modelo 33
5.3 Utilização dos Recursos e Consumo de Energia 33
5.4 Escalabilidade da Arquitetura 34

5.5 Velocidade Teórica 35
5.6 Comparação com o Pytorch 35
6 Considerações Finais 37
6.1 Conclusão 37
6.2 Trabalhos Futuros 37

Referências 39

1 Introdução

Com a ascensão das *Graphics Processing Units* (GPU) e da computação em nuvem, o uso de *deep neural networks* (DNN) tem se tornado o estado-da-arte para a solução de diversos problemas de Inteligência Artificial. Em especial, as *convolutional neural networks* (CNN) tem recebido grande atenção de pesquisadores e desenvolvedores devido a gama de problemas que pode vir a ser aplicada, como: classificação de imagens (RUSSAKOVSKY et al., 2015), aplicações na área médica (SALEHI et al., 2020) (MAHESH et al., 2021a), aplicações em veículos autônomos (MULLAPUDI et al., 2018), detecção de intrusos (PENG, 2020a), detecção facial (SUN; WANG; TANG, 2015), sistemas de segurança (XU, 2021) e diversas outras aplicações (LI et al., 2021).

Porém, segundo Véstias (2019), o uso de DNNs na nuvem é acompanhada de diversas limitações, tais como: alta latência, necessidade de uma rápida conexão com a internet e o alto custo de manutenção. Visando solucionar essas limitações, muitos modelos de arquitetura em dispositivos de borda começaram a ser desenvolvidos. Estes dispositivos são parte da computação de borda, e aproximam as aplicações (e as unidades responsáveis pela execução das mesmas) às fontes de dados. Com isso, a latência é reduzida e os dados não necessitam percorrer uma rede para ser enviado a uma plataforma para ser processado, deste modo obtendo-se tempos de resposta mais rápidos (VéSTIAS et al., 2020).

Como descrito em Ren et al. (2019), os paradigmas de *edge computing* e *cloud computing* se diferem em diversos aspectos, como: poder Computacional, latência, largura de banda, custo, segurança e energia (Tabela 1.1). Porém, elas não são mutuamente exclusivos, ou seja, eles devem colaborar entre si para prover melhor performance para uma determinada tarefa ou aplicação.

A execução de CNNs em dispositivos de borda é acompanhada da complexidade de obter tempo de resposta e precisão aceitáveis, visto que estes dispositivos possuem grandes limitações de recurso (LI et al., 2016). Por exemplo, ao desenvolver uma arquitetura voltada a CNNs, a quantidade de memória disponível no *hardware* pode vir a ser um problema. Isto ocorre pois, a rede neural é constituída por um conjunto de neurônios organizados em camadas, que são constituídas de multiplicações e somas. E a cada nova *neural network* (NN) desenvolvida, o número de camadas aumenta, o que, do ponto de vista do *hardware*, aumenta também a quantidade de dados que será armazenado na memória, transferido para as unidades de processamento e ser salvo na memória após o término do processamento (KOROL, 2019).

Recentemente, diversos estudos propuseram a utilização de *field-programmable gate arrays* (FPGAs) como uma das plataformas para realizar aceleração de hardware para as CNNs, devido a sua capacidade de reconfiguração, baixo custo, grande quantidade de recursos lógicos, baixo consumo de energia, seu rápido avanço tecnológico e o fato de que, desenvolvimento

Dispositivos de Borda vs Computação na Nuvem	
Baixa latência	Alta latência
Baixo tempo de Resposta	Alto tempo de resposta
Baixa dependência em conectividade	Alta dependência em conectividade
Específico para tarefas	Específico para aplicações
Baixo Custo	Alto Custo
Baixo poder de computação	Alto poder de computação
Limitação de Energia	Não há limitação energética

Tabela 1.1 – Comparação entre dispositivos de borda e *Cloud computing* (VéSTIAS, 2019)

recentes de CNNs aumentaram a esparsidade e o uso de tipos de dados mais compactos, o que favorece FPGAs, pois essas são desenvolvidas para trabalhar com paralelismo irregulares e tipos de dados customizados.

Este capítulo encontra-se organizado como se segue. A Seção 1.1 apresenta a justificativa para a realização desse trabalho. A Seção 1.2 descreve os objetivos geral e específicos. Finalmente, a Seção 1.3 apresenta o delineamento do restante da monografia.

1.1 Justificativa

Com o crescimento do uso de modelos de inteligência artificial em dispositivos móveis e *Internet of Things* (IoT), é necessário que os *hardwares* responsáveis pela execução dos mesmos possuam a capacidade de realizar milhares de operações por segundo. Além disso, este crescimento implica em um aumento significativo no número de requisições e acesso a rede de internet que, por sua vez, causa o aumento da latência da rede.

Com a pandemia do COVID-19, foi possível notar como as redes de Internet são suscetíveis a oscilações advindas de múltiplos acessos simultâneos. Em 2020, com as restrições de mobilidades adotadas por governos em, praticamente, todo mundo, houve um aumento no uso da rede. Nos Estados Unidos, especificamente nas cidades de Nova York e Nova Jersey, segundo (BERGAMAN; IYENGAR, 2020), o tráfego teve um aumento de 44.6% e teve uma queda na velocidade de *download* de 5.5% no mês de março. Essas variações poderiam causar transtornos ainda maiores se um carro autônomo fizesse acesso a internet para realizar suas decisões pois, deste modo, teríamos chances maior de acidente durante este período de tempo.

Com o desenvolvimento de arquiteturas que viabilizem o uso de modelos de *machine learning* (ML) em dispositivos de borda, a execução destes modelos poderá ocorrer de forma mais rápida e eficiente e evitando ao máximo o acesso à rede de internet. Deste modo, este trabalho propõem o desenvolvimento de uma nova arquitetura voltada à execução de CNNs em FPGAs de baixa densidade [1].

[1] FPGAs de baixa densidade, são placas com uma quantidade menor de recursos e, devido a este motivo, mais baratas.

1.2 Objetivos

Este trabalho possui, como objetivo geral, o desenvolvimento e a validação do TiNN (*There is No Name*), uma arquitetura de *hardware* para aceleração de modelos de Redes Neurais Convolucionais. Para tanto, é realizada a implementação de mecanismos como convolução, *stride* e funções de ativações, presentes em uma grande quantidade de modelos de DNNs. A arquitetura desenvolvida focada no dispositivo Zynq7020, na qual os seus recursos devem ser usados de maneira econômica, porém sem prejudicar a execução do modelo de CNN.

1.3 Organização do Trabalho

Esta monografia encontra-se organizada como se segue. O Capítulo 1 realiza a contextualização do problema a ser trabalhado, apresentando os problemas centrais para o desenvolvimento de *hardware* voltado a modelos de CNNs. O capitulo 2 discute toda a fundamentação teórica necessária para a realização e entendimento deste trabalho, envolvendo conceitos sobre aprendizado de máquina,Redes Neurais Convolucionais, Linguagem de descrição de *Hardware*, *Field Programmable Gate Arrays* e Simulação e Síntese de *Hardware*, além dos Trabalhos relacionados. O Capítulo 4 descreve o desenvolvimento do TiNN, envolvendo tópicos como arquitetura de funcionamento e os módulos presentes nela. Por fim, o Capítulo 6 apresenta as considerações finais e a descrição das atividades pendentes para efetivação do trabalho.

2 Revisão Literatura

Este capítulo apresenta a revisão de literatura feita para a realização deste trabalho. Encontra-se organizado da seguinte forma: a Seção 2.1 apresenta a fundamentação teórica utilizada para o desenvolvimento deste trabalho e a Seção 3 apresenta os trabalhos que possuem temas diretamente relacionados ao objetivo geral deste trabalho.

2.1 Fundamentação Teórica

Nesta seção, é apresentado o suporte teórico necessário para o entendimento e o desenvolvimento deste trabalho. A Subseção 2.1.1 é uma breve introdução ao tema de *Machine Learning* e apresenta as noções básicas de aprendizagem profunda que são importantes ao trabalho realizado. As subseções 2.1.2 e 2.1.3 discorrem sobre Linguagem de Descrição de Hardware e sobre os componentes básicos que estão contidos em uma FPGA. Por fim, a Subseção 2.1.4 apresenta como é realizada a simulação e síntese de um algoritmo escrito em uma linguagem de descrição de hardware.

2.1.1 Machine Learning

Machine Learning (figura 2.1) é um ramo da inteligência artificial que é capaz realizar a automatização uma tarefa dado um modelo. Ou seja, ela permite que computadores aprendam, a partir de dados, a identificar padrões e realizarem tomadas de decisões com a finalidade de solucionar problemas complexos.

Em problemas de *Machine Learning*, o computador aprende a rotular um determinado alvo. Este aprendizado pode ser feito através de sistemas supervisionados, não supervisionados ou por reforço.

Na **aprendizagem supervisionada** os dados se encontram relacionados com as saídas, logo o que se busca é entender como os dados de entrada estão relacionados com os dados de saída, ou seja para cada conjunto de dados de entrada temos o registro de a qual classe (*label*) o conjunto pertence (GOODFELLOW; BENGIO; COURVILLE, 2016). Na **aprendizagem não-supervisionada** o modelo aprende com os dados que não foram previamente rotulados, ou seja não possuem classes. Deste modo, o modelo tenta identificar padrões nos dados para agrupa-los (GOODFELLOW; BENGIO; COURVILLE, 2016). Por fim, na **aprendizagem por reforço** o modelo irá passar por um treinamento para tomar uma sequência de decisões, a fim de atingir uma meta em ambiente incerto. Para isso, o modelo irá realizar uma sequência de decisões que podem ser corretas ou não, de modo a aprender com essas escolhas. Para cada decisão que aproxima o modelo do resultado correto, ele receberá uma recompensa, caso contrário uma penalidade.

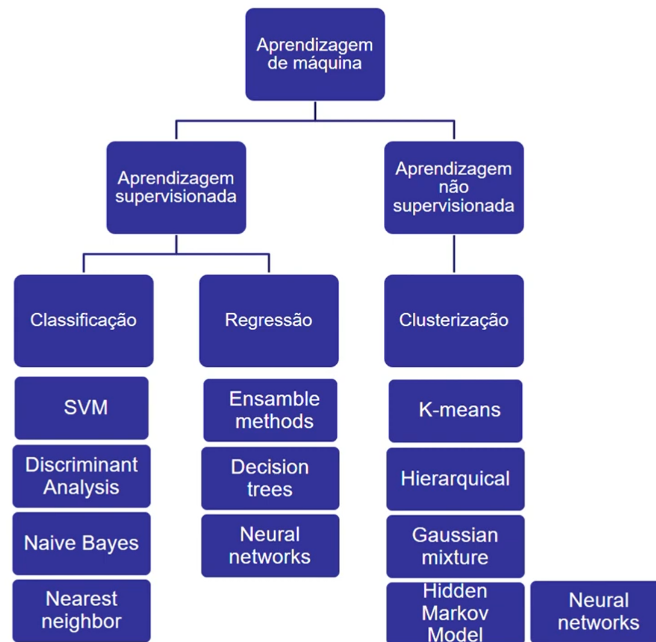

Figura 2.1 – Divisão de métodos para aprendizagem supervisionada e não-supervisionada

FONTE: Elaborado pelo autor

Um exemplo clássico é o problema de classificação de emails como *spam*, onde o modelo deve, a partir de um conjunto de dados, determinar se um email é ou não spam. Se formos usar um ser humano para realização dessa tarefa, ele poderia aprender de três formas:analisando os dados já coletados e suas relações em relação a ser *Spam* ou não (aprendizagem supervisionada); analisando os dados e encontrando padrões para rotular os dados (aprendizagem não-supervisionada); ou aprender através de tentativas e erros (aprendizagem por reforço).

Para o primeiro caso, onde já existe a definição das relações entre os dados coletados e as *labels*, o ser humano que está aprendendo a classificar os emails, iria observar os dados e suas *labels* e faria uma previsão de acordo com as informações que possuí. Para o caso onde as *labels* não são fornecidas, o ser humano poderia analisar os dados a fim de encontrar padrões, e agrupa-los em dois grupos distintos: *Spam* ou não-*Spam*. Por fim, no último caso, o ser humano iria estar em uma sala e a cada classificação correta ele ganharia uma bala, caso contrário uma pimenta. No inicio ele iria realizar as escolhas de forma randômica (de forma aleatória) a fim de começar a ter uma ideia de como os dados estão relacionados com o problema, e com o tempo ele será capaz de fazer predições melhores.

Após o término da fase de aprendizado, o modelo está preparado para realizar decisões em dados que são novos. Para isso um modelo de inferência [1] é gerado para avaliar os dados desconhecidos.

[1] Modelo de inferência realiza as decisões baseado no treino feito, ou seja, no conjunto de dados observado.

2.1.1.1 *Deep learning*

Deep learning (DL) é uma sub-área do *Machine Learning*. Os modelos atuais de DL são capazes de representar funções de grande complexidade, sendo capazes de resolver problemas que possuem um nível de complexidade grande, de modo que, se um ser humano fosse resolver levaria um tempo (GOODFELLOW; BENGIO; COURVILLE, 2016).

A ideia do *Deep learning* consiste em simular como o cérebro humano que recebe e processa as informações do ambiente. No caso do cérebro humano, a parte de transporte é realizada pelos neurônios, células que recebem informações por meio de sinais elétricos e transmitem a outros neurônios até que a informação chegue ao cérebro, onde é por fim, processada.

2.1.1.2 Perceptron

A ideia de criar um modelo que funcione de forma similar ao cérebro humano vem desde a década 1950, através do cientista Frank Rosenblatt. Rosenblatt propôs o Perceptron (Figura 2.2) um modelo matemático que tem como entrada diversos sinais e produz somente uma saída. para isso, ele propôs um conjunto de pesos, W_i, onde $W_i \in R$ e $i \in [0, n]$, que determinam o quão importante é cada uma das entradas X_i para a saída do perceptron. A saída é um valor binário, 0 ou 1, que é obtida através da análise do somatório da multiplicação entre a entrada X_i e o peso w_i, que ocorre no neurônio artificial do perceptron. Se, o somatório for maior que um dado limite, então a saída recebe 1, caso contrário 0, como pode ser visto na equação 2.1.

$$\begin{cases} 0 & if \quad \sum_i w_i x_i \leq limite \\ 1 & if \quad \sum_i w_i x_i > limite \end{cases} \tag{2.1}$$

Apesar da inspiração na biologia, os modelos de redes neurais artificiais usados atualmente não simulam o cérebro de forma completa, o cérebro humano possuí 10^{11} neurônios, conectados a, aproximadamente, 10^4 neurônios cada (MITCHELL, 1997). Isto resultaria em uma rede extremamente densa, impossível de ser executada em um computador.

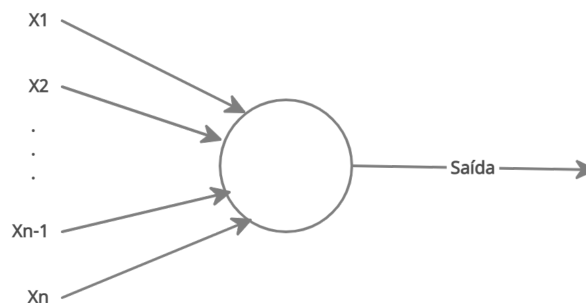

Figura 2.2 – Perceptron proposto por Rosenblatt na década de 1950

FONTE: Elaborado pelo autor

O *Multi-Layer Perceptron* (MLP) resolve a deficiência do perceptron comum para a resolução de problemas que não são linearmente separáveis. Para isto é usado mais de uma camada.

A parte principal de uma MLP é chamada de neurônio artificial (figura 2.4). A cada neurônio artificial k temos um conjunto de sinais x_j que são multiplicados por um peso w_{kj}, um somatório que realiza a soma dos valores resultantes da multiplicação por um valor chamado *bias* [2] b_k e uma função de ativação, responsável por manter em um limite o resultado do somatório (HAYKIN, 2009). De forma resumida, o MLP consiste em: uma camada de entrada, que possui conjunto de unidade sensoriais, chamadas de nós fonte; uma ou mais camadas ocultas de neurônios (nós computacionais); e uma camada de saída, constituída de um ou mais neurônios.

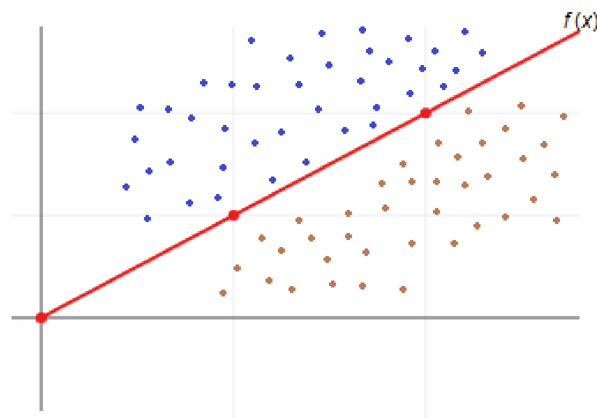

Figura 2.3 – Exemplo de um problema linearmente separável. Neste caso, o Perceptron têm de definir se uma cor é azul ou marrom.

FONTE: Elaborado pelo autor

O MLP é um tipo de modelo *feedfoward*. Nestes modelos a informação flui pela rede em uma direção somente, não possuindo ciclos (conexões que permitem o *feedback*). As redes Convolucionais, usadas em classificação de imagens, é um tipo de modelo *feedfoward* (GOODFELLOW; BENGIO; COURVILLE, 2016), e serão o foco neste trabalho.

2.1.1.3 *Element Wise Matrix Multiplication* (Produto de Hadamard)

O *Element Wise Matrix Multiplication*, ou Produto de Hadamard (figura 2.5) [3], é uma operação binaria entre duas matrizes de mesma dimensão e tem como resultado uma terceira matriz com a mesma dimensão dos operadores (HORN; JOHNSON, 2012).

Dada duas matrizes A e B, que possuam dimensão $l \times n$, o produto de Hadamard $A \odot B$ é uma matriz de dimensão $l \times n$, no qual os elementos são obtidos através da equação (MILLION,

[2] *Bias* é o valor que determina o quão fácil o MLP ativa, se ele possuir um valor elevado o modelo será ativado com mais facilidade, caso contrário com mais dificuldade (ZHANG et al., 2021)

[3] Jacques Hadamard (1865- 1963) foi um matemático francês que realizou diversas contribuições nos campos de Teoria dos Números, Analises Complexas, Geometria Diferencial e Equações parciais diferenciais.

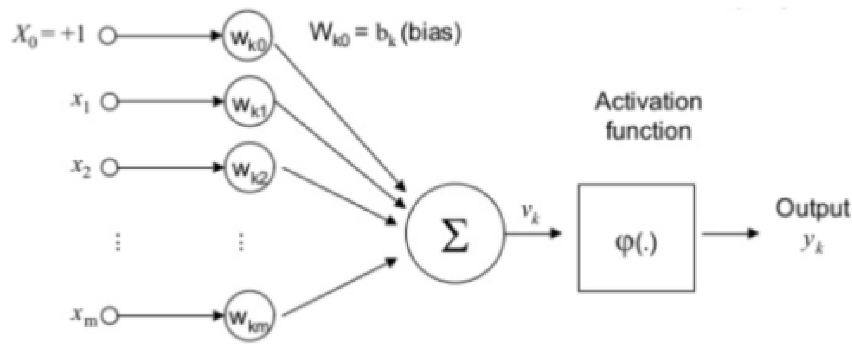

Figura 2.4 – Neurônio e suas partes principais

FONTE: (SINGH, n.d)

2007)

$$(A \odot B)_{ij} = (A)_{ij}(B)_{ij} \tag{2.2}$$

onde, $i \in [0, l]$ e $j \in [0, n]$. O exemplo abaixo demonstra o produto de Hadamard para as matrizes A e B de dimensão 4×4.

$$(A \odot B) = \begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix} \times \begin{bmatrix} B_{00} & B_{01} & B_{02} & B_{03} \\ B_{10} & B_{11} & B_{12} & B_{13} \\ B_{20} & B_{21} & B_{22} & B_{23} \\ B_{30} & B_{31} & B_{32} & B_{33} \end{bmatrix} = \begin{bmatrix} A_{00} \times B_{00} & A_{01} \times B_{01} & A_{02} \times B_{02} & A_{03} \times B_{03} \\ A_{10} \times B_{10} & A_{11} \times B_{11} & A_{12} \times B_{12} & A_{13} \times B_{13} \\ A_{20} \times B_{20} & A_{21} \times B_{21} & A_{22} \times B_{22} & A_{23} \times B_{23} \\ A_{30} \times B_{30} & A_{31} \times B_{31} & A_{32} \times B_{32} & A_{33} \times B_{33} \end{bmatrix}$$

Figura 2.5 – Exemplo de um produto de Hadamard entre duas matrizes 4×4.

FONTE: Elaborado pelo autor

O Produto de Hadamard é muito utilizado na operação de Convolução, que é de extrema importância para as CNN.

2.1.1.4 Convolução

A convolução é uma operação matemática que, a partir de duas funções gera uma terceira que é a soma do produto das mesmas em relação a uma região delimitada na qual ambas funções se sobrepõem em relação ao deslocamento existente entre elas. A operação de convolução é dada pela equação

$$(f * g)(n) = h(n) = \sum_{j=0}^{n} f(j) \cdot g(n - j) \tag{2.3}$$

onde f e g são sequências de tamanho k e a formula fornece o n-ésimo elemento do resultado.

Na Inteligência artificial e processamento de imagens são usados dois somatórios, visto que usamos duas dimensões (altura e largura), sendo a equação resultante dada por

$$g(x,y) = \omega * f(x,y) = \sum_{dx=-a}^{a} \sum_{dy=-b}^{b} \omega(dx,dy) f(x+dx, y+dy) \tag{2.4}$$

onde, ω é o filtro a ser aplicado sobre a imagem $f(x,y)$ resultando na imagem $g(x,y)$ e dx e dy são os valores do deslocamento a ser feito pelo filtro ω na função $f(x,y)$.

De uma forma mais explicita, a convolução é o produto entre uma das imagens por uma cópia deslocada e invertida da outra imagem, sendo aplicado o produto de Hadamard entre as duas imagens e o resultado de cada elemento do Produto de Hadamard é somado, gerando o resultado final. A imagem 2.6 exemplifica a operação de convolução entre uma imagem $6 \times 6 \times 1$ e um filtro (filtro) $3 \times 3 \times 1$, com o deslocamento (***stride***[4]) do filtro igual a 3.

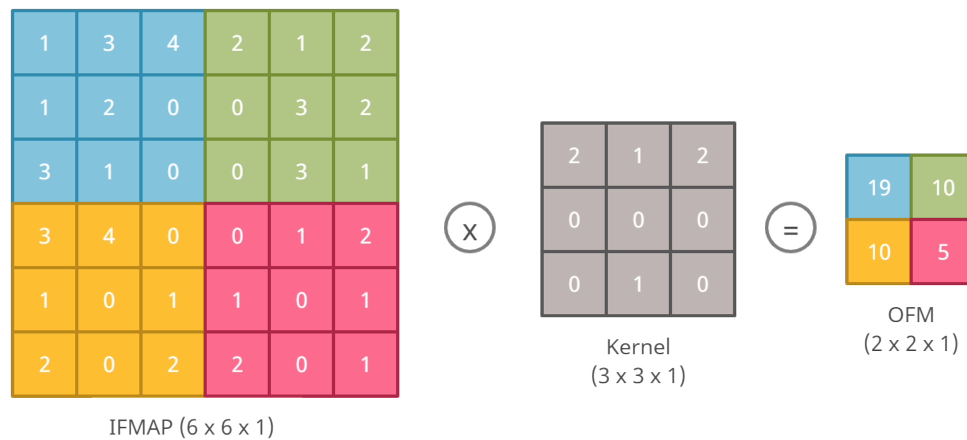

Figura 2.6 – Exemplo de Convolução entre um IFM de dimensões $6 \times 6 \times 1$ e um filtro de dimensões $3 \times 3 \times 1$, com *stride* = 3 gerando uma imagem de saída com dimensões $2 \times 2 \times 1$.

FONTE: Elaborado pelo autor

2.1.1.5 *Convolution Neural Networks* (CNN)

Convolution Neural Networks é um tipo de rede neural voltada a reconhecimento e classificação de imagens[5]. Uma CNN recebe uma imagem de entrada e atribui um determinado grau de importância, pesos ou *biases*, a aspectos da imagem, sendo assim capaz de realizar as

[4] Stride é uma operação que desloca o filtro sobre uma imagem. Por exemplo, se $Stride = 1$ o filtro irá se deslocar em 1 pixel por vez

[5] O problema de classificação de imagens é um dos principais na área de Visão Computacional, e possuí diversas aplicações em áreas como segurança (PENG, 2020b), saúde (MAHESH et al., 2021b) e carros autonômos (GRIGORESCU et al., 2020).

classificações necessárias. As imagens em uma CNN são tensores, ou seja,[6] matrizes com mais de duas dimensões. Para os computadores, uma imagem é representada por uma matriz de pixels (números), essa matriz pode ser de 1-Dimensão, cinza, ou em 3-Dimensões, RGB (vermelho, verde e azul).

Uma CNN consiste em diversas camadas de convolução (figura 2.7) , usadas na extração das características das imagens de entrada, nas quais são responsáveis pela maior parte das computações. Além da camada de convolução, uma CNN também possui camadas de *pooling*, para realização de *sub-sampling*, e camadas densas (*Fully-Connected*), que por sua vez, é a última camada da rede, que têm como finalidade a realização da classificação da imagem sendo processada. Em geral, as CNNs fazem uso de camadas conectadas em sequência para realizar extração e transformação de características. Devido as múltiplas camadas de representação sendo capazes de aprender características mais complexas e detectar padrões não lineares de forma sistemática. Em contraste com os outros modelos de redes neurais artificiais, CNNs possuem um grande número de vantagens: 1) Não necessitam de conexão total entre os neurônios da camada atual com a próxima, implicando em uma convergência mais rápida; 2) Os pesos são compartilhados, implicando na diminuição de parâmetros; 3) Redução das dimensões da imagem, diminuindo, também, os parâmetros (LI et al., 2020).

As convoluções ocorrem, primeiramente, entre a imagem de entrada e um filtro, essa convolução gera um *feature map*. Esse *feature map* resultante é chamado de *output feature map* (OFM) e ao ser alimentado a outra camada da rede ele recebe o nome de *input feature map* (IFM). Após passar pela primeira camada, o *feature map* é alimentado a próxima camada, que pode ser uma nova camada de convolução, *pooling*, ou a última camada da rede (camada densa).

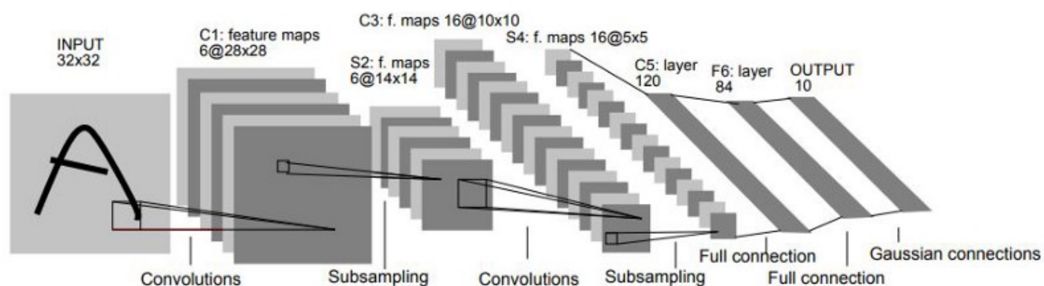

Figura 2.7 – Exemplo de uma CNN, arquitetura da LeNet-5. A rede é composta por 7 camadas, sendo três de convoluções, duas de *pooling* e duas camadas *fully connected*

FONTE: (LECUN et al., 1998)

As camadas de ***pooling*** são responsáveis por realizar o *subsampling* do IFM, a fim de diminuir o *overfitting*[7]. e conseguir evitar a *translation variance*[8]. Dada uma imagem de entrada

[6] Tensores são contêineres que podem armazenar dados em N dimensões. Uma matriz, por exemplo, é um Tensor de 2-Dimensões.

[7] *Overfitting* ocorre quando um modelo se ajusta ao conjunto de dados de treino, mas se é ineficaz para prever novos resultados

[8] Em computação gráfica, *translation invariance* se refere a habilidade do modelo reconhecer um objeto indepen-

e o tamanho da janela de *pooling*, a função de *pooling* é aplicada sobre os neurônios dentro da janela. Os dois principais métodos de *pooling* são o *Max-Pooling*, que seleciona o maior valor de neurônio dentro da janela de *pooling* e o *Average-Pooling*, que usa como resultado a média dos neurônios dentro da janela de *pooling* (figura 2.8).

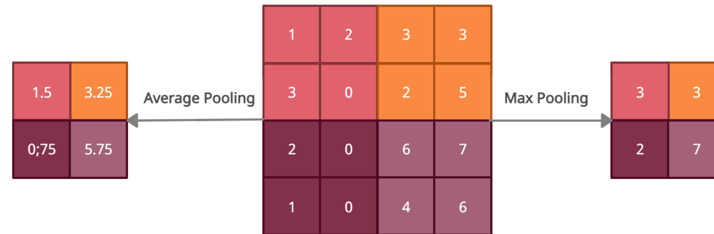

Figura 2.8 – Exemplo de *Pooling* entre um IFM de dimensões $4 \times 4 \times 1$ usando uma janela de dimensões 2×2.

FONTE: Elaborado pelo autor.

Cada camada de convolução de uma CNN é, muitas vezes, seguida por uma **camada de ativação** que aplica uma função não-linear para todos os valores do *feature map* (NWANKPA et al., 2018). As funções que são aplicadas, e usadas neste projeto, são a *sigmoid* e a ReLu (NAIR; HINTON, 2010).

$$sigmoid = \frac{1}{1 + e^{-z}} \tag{2.5}$$

$$ReLu = x^{+} = max(0, x) \tag{2.6}$$

A função *sigmoid*, equação 2.5, tem como sua principal vantagem o fato de não ser linear, ou seja, quando temos diversos neurônios usando-a a saída também não será linear. Os valores da função *sigmoid* variam entre 0 e 1, como pode ser visto na figura 2.9 (a). Porém, o fato de os resultados estarem entre 0 e 1 acaba fazendo com que a saída tenha somente valores positivos, e nem sempre desejamos que os valores enviados ao próximo neurônio tenham o mesmo sinal (ZHANG et al., 2021).

A função ReLU, equação 2.6, é outra função não linear, o que permite realizar o *backpropagation* e ter diversas camadas de neurônios artificiais ativados pela ReLU. Sua vantagem se encontra no fato de não ativar todos os neurônios ao mesmo tempo, pois para valores negativos a ReLU resulta 0 como valor de ativação, como pode ser visto na figura 2.9. Deste modo, é gerada uma rede esparsa e eficiente (ZHANG et al., 2021).

Outra operação muito aplicada na maioria das redes neurais é o ***padding***. Ele é usado para ajustar o tamanho do dado de entrada às configurações de uma camada. É muito utilizado

dente da posição que ele se encontra

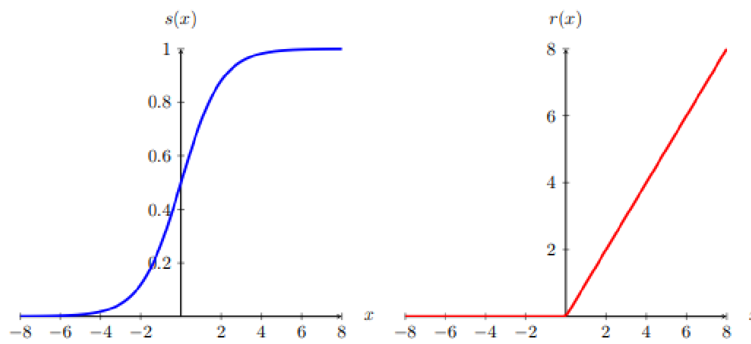

Figura 2.9 – Gráficos da função *sigmoid*, $S(x)$, e ReLU, $r(x)$

FONTE: (FUHRMANN, 2018).

em redes convolucionais quando a dimensão da janela do filtro e do *stride* não são compatíveis com a dimensão do IFM. Quando isso ocorre, é adicionado ao IFM colunas e linhas com valores 0, *zero padding*, fazendo com que a dimensão dele permita as operações da próxima camada.

As últimas camadas de um modelo de CNN são, geralmente, as **camadas densas**. Essas camadas possuem a mesma estrutura da rede neural tradicional, onde cada neurônio está conectado com todos os neurônios da camada anterior (ver figura 2.10). Elas são usadas para classificar os dados entre diversas classes, após a operação de extração das características (SINGH, n.d).

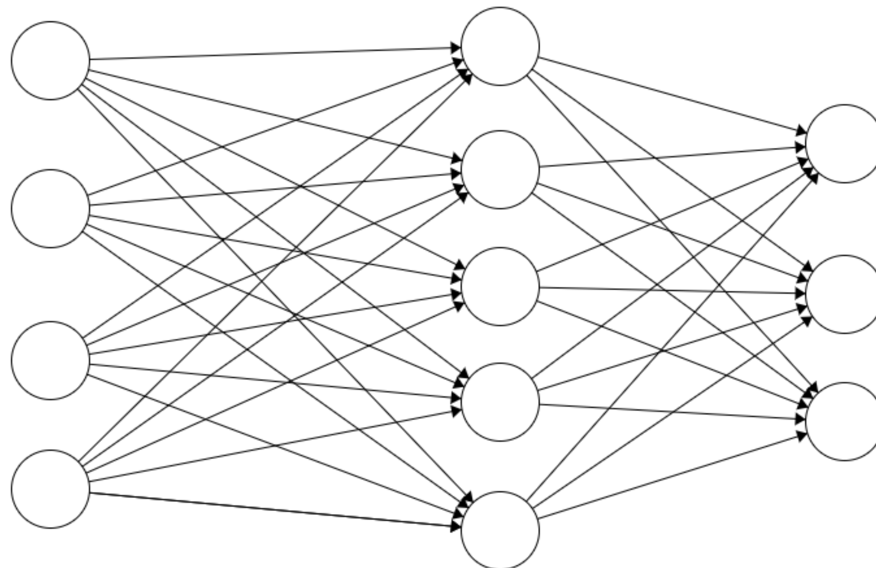

Figura 2.10 – *Fully Connected layers*

FONTE: (MONTENEGRO, 2020)

2.1.2 Linguagem de Descrição de Hardware (HDL)

Segundo Smith e Zamfirescu (1998), HDL é uma tipo de linguagem de programação usada para modelar operações desejadas em um hardware. É uma linguagem declarativa, facilitando

a descrição abstrata do comportamento do hardware e a modelagem da estrutura do hardware independente do comportamento do design.

Duas das principais HDL existentes são o VHDL e o Verilog. Neste projeto foi escolhido usar o VHDL pelas seguintes razões (SMITH; ZAMFIRESCU, 1998):

- Reusabilidade, *Procedures* e Funções podem ser colocadas em pacotes, permitido o reuso em todo *design*.
- Possibilidade de criação de tipos de dados pelo usuário, fazendo com que os modelos sejam menos complicados de serem criados.
- Melhor suporte para o controle de grandes design.
- Replicação estrutural, VHDL possui o comando *generate*, permitindo a replicação do mesmo design ou parte dele.

2.1.3 *Field Programmable Gate Array* (FPGA)

Os *Field Programable Gate Array* (figura 2.11) funcionam com lógica programável, ou seja, a arquitetura que é compilada na placa é gerada através de uma linguagem de descrição de *hardware*. Eles são compostos por uma matriz de blocos lógicos independentes entre si que podem ser conectados através de interconexões programáveis, podendo ser reprogramada para execução de diferentes projetos eletrônicos.

Devido a sua capacidade de reprogramação, as FPGAs podem ser aplicadas a diversos campos e aplicações, como: área médica, comunicação sem-fio, indústria automotiva, processamento de imagem e video (XILINX, 2022b).

Uma FPGA é constituída por três blocos principais (ver figura 2.11: Blocos I/O (entrada e saída), que são módulos responsáveis pela interconexão entre os dados de entrada e dos dados resultantes dos blocos lógicos; os blocos Lógicos Configuráveis (CLBs), que possuem um conjunto de RAM para a criação de funções logicas combinatórias,*look-up tables* (LUTs), e *flip-flops* (FF) [9] para armazenamento de dados, podendo também possuir *shift registers*, multiplexadores e operadores aritméticos (ver figura 2.12); e o *Switch Matrix*, que são responsáveis por fazer a conexão dos blocos I/O com os CLBs e as conexões entre os CLBs da FPGA (MANSUR, 2016) (ZEIDMAN, 2006).

As próximas seções detalham de forma mais aprofundada as *Look-up table*, blocos de memória e blocos aritméticos de uma FPGA.

[9] Circuito digital que pode funcionar como memória de 1-bit (armazena 0 ou 1).

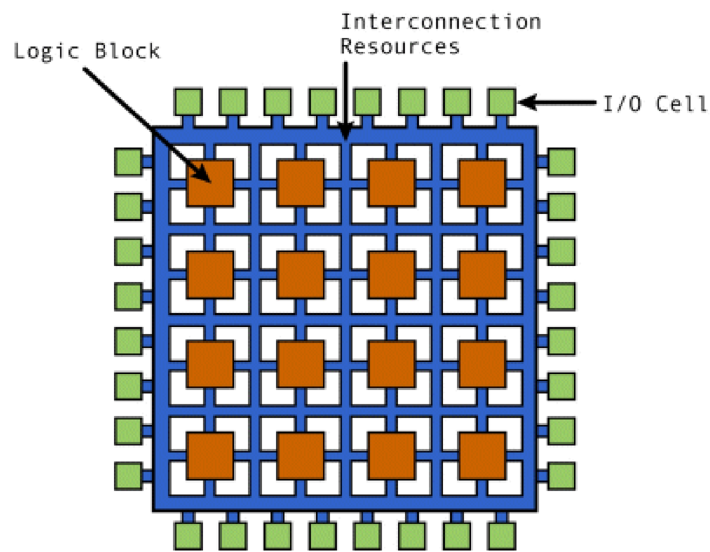

Figura 2.11 – Arquitetura genérica de uma FPGA

FONTE: (ZEIDMAN, 2006)

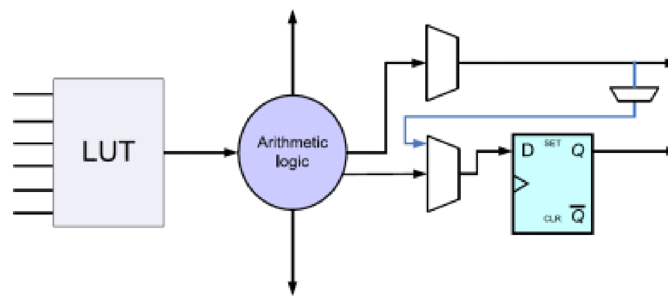

Figura 2.12 – Arquitetura de um CLB

FONTE: (BOBDA, 2007)

2.1.3.1 *Look-up Table* (LUT)

As LUTs (figura 2.13) são responsáveis pela implementação de funções nas FPGAs. Um conjunto de diversos LUTs formam um CLB, o que permite uma conexão mais rápida entre as LUTs do que as conexões CLB-*to*-CLB (KOROL, 2019).

Deste modo, uma função em uma FPGA é implementada a partir de uma lógica combinacional (tabela verdade) que define como a função irá se comportar. Ou seja, qualquer combinação de portas lógicas [10] que implementam um comportamento qualquer, pode ser realizada a partir do uso das LUTs.

[10] Portas Lógicas são dispositivos lógicos que, a partir de duas entradas (que podem ser 0 ou 1) que produzem uma saída válida.

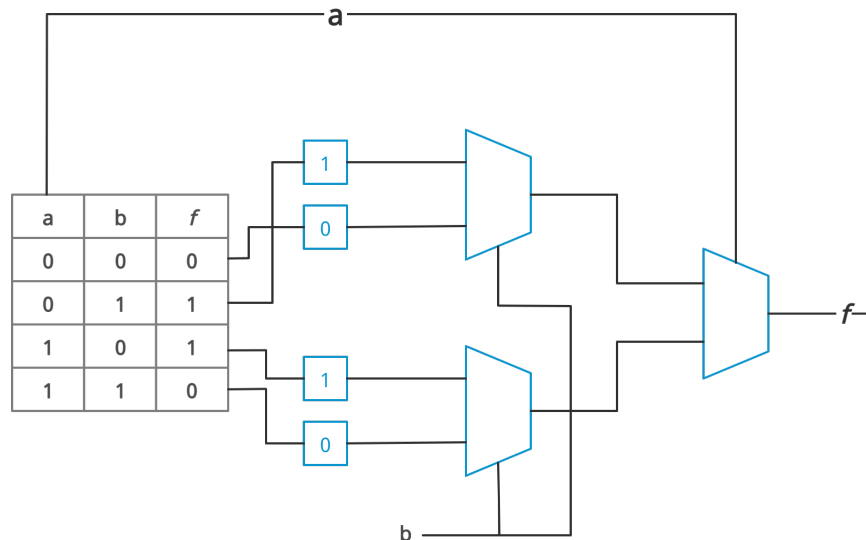

Figura 2.13 – Exemplo de uma LUT com duas entradas sua arquitetura usada na implementação da função $f = \overline{a}b + a\overline{b} = a \oplus b$.

FONTE: Elaborado pelo autor

2.1.3.2 *Digital Signal Processor* (DSP)

O uso de FPGAs para a execução de CNNs faz com que seja possível o cálculo convoluções, que por sua vez pode ser reduzido a operações de multiplicações e soma. Para a execução de operações que fazem uma grande quantidade de multiplicações binárias e acumulações sem usar as LUTs, as FPGAs possuem os blocos de sinais digitais (DSPs) (figura 2.14) (KOROL, 2019).

Os blocos DSPs são frequentemente usados em processamento de sinais. Deste modo, eles necessariamente possuem unidades de multiplicação e acumulação grandes, para realização de operações de *multiply-accumulate*, pro exemplo (XILINX, 2018c).

2.1.3.3 *BlockRam* (BRAM)

Os blocos de *Block RAM* (BRAM) (figura 2.15), são blocos lógicos que permitem um acesso rápido aos dados além de possuírem uma memória síncrona grande que permite o seu uso no mesmo ciclo de *clock* que a lógica RTL[11] (FUHRMANN, 2018). Ao combinar diversos BRAM pode ser obtida uma memória de qualquer largura ou profundidade, o que substitui a necessidade de implementar a memória com LUTs ou FF, o que não é adequado para memórias muito grandes (XILINX, 2016).

[11] *Register Tranfer Level* descreve as operações de um circuito síncrono que por sua vez é descrito em termo do fluxo dos sinais.

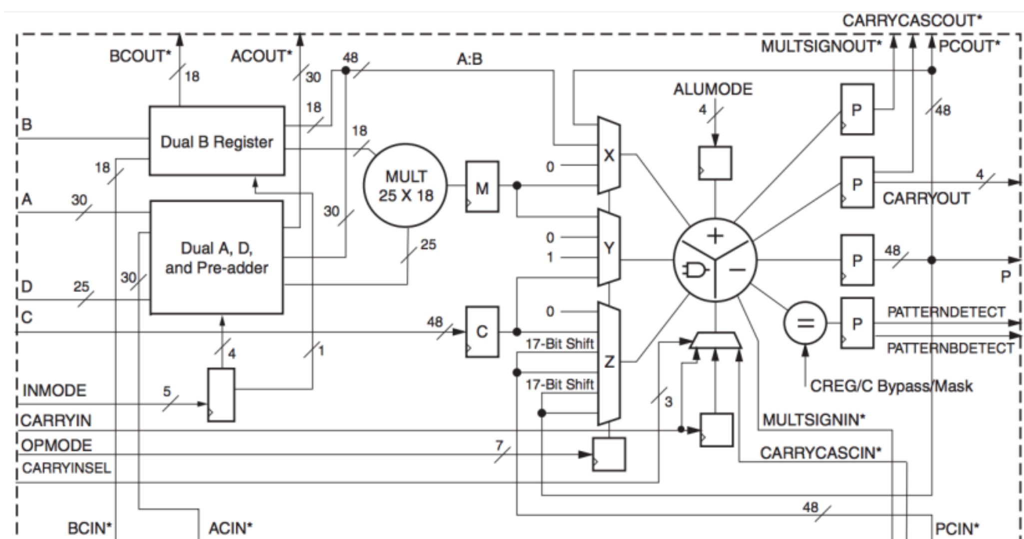

Figura 2.14 – Estrutura de um Bloco DSP, com um multiplicador de complemento de dois 25×18, um acumulador de 48-bit e outras unidades.

FONTE: (XILINX, 2018c)

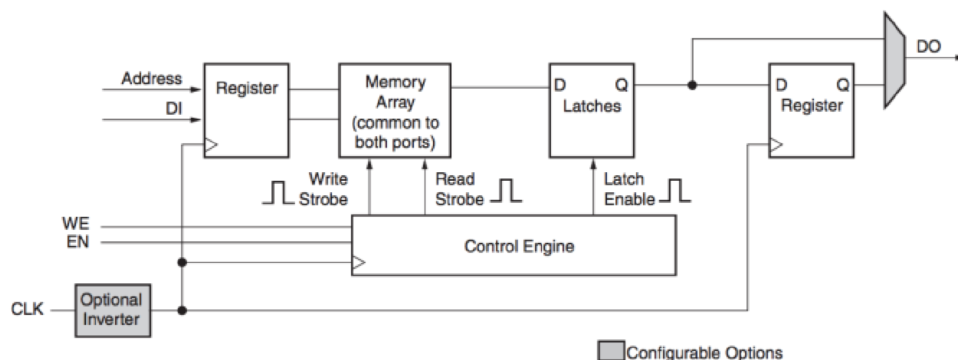

Figura 2.15 – Estrutura de um Bloco BRAM.

FONTE: (XILINX, 2016)

2.1.4 Simulação e Síntese de Hardware

Para garantir o funcionamento do design das unidades implementadas, é necessário a realização de testes e simulações, chamadas de *testbench*. Na simulação de hardware, os resultados são avaliados usando formas de onda, que mostram o estado do hardware no decorrer do tempo. Caso o *design* final não apresente nenhum erro em nenhuma de suas unidades durante as simulações, ele está apto a ser submetido a fase de síntese (XILINX, 2018a).

Na fase de síntese realiza otimizações sobre a lógica RTL implementada e as converte me componentes de hardware (XILINX, 2018b).

3 Trabalhos Relacionados

Nesta seção, é apresentado alguns trabalhos que visaram a solução do problema tratado neste trabalho.

Dispositivos com foco em computação de borda devem possuir baixo consumo de energia e serem baratos. Porém, para se tornarem viáveis para execução de CNNs, devem ser capazes de atingirem milhares de GFLOPs (*Giga Floating-point Operations per second*), respeitando estas limitações. Plataformas como GPUs podem atingir milhares de GFLOPs e *Tensor Processing Units* (TPUs) [1] são capazes de atingir marcas de TOPs (*Tera Operations per second*), porém a um custo de uma enorme quantidade de energia, o que inviabiliza o uso dessas tecnologias para dispositivos de bordas (GOOGLE, n.d). Soluções especificas são mais eficientes, porém sofrem com a limitação em relação a possibilidade de re-configuração, o que acaba por torná-los, de forma rápida, obsoletos em relação aos avanços das redes neurais (VéSTIAS, 2019).

Devido a estes fatores, implementações de arquiteturas para execução de CNNs em FPGAs ganharam foco devido possibilidade de reconfigurabilidade de acordo com a rede neural sendo executada (VéSTIAS, 2019). Para isso, nos últimos anos, foram desenvolvidas pesquisas focando na redução do custo de memória e na complexidade computacional para executar inferências na rede.

Pesquisas que fizeram uso de FPGAs de alto custo, demonstraram uma boa performance na execução de redes neurais. Em Liang et al. (2020), usando uma placa ZCU102, os autores propuseram uma arquitetura que faz uso do algoritmo *winograd*, diminuindo a complexidade aritmética e com um ganho de performance. Além disso, foi desenvolvida uma estrutura de *line buffers*, a fim de reusar os dados do *feature map* entre diversos blocos. O reuso dos dados, exemplificado na figura 3.2, é feito armazenando parte dos dados de entrada na memória interna e esses dados irão ser reutilizados na próxima convolução. Nesta figura, podemos notar que m blocos do dado de entrada são armazenados na memória interna e depois reutilizados na próxima convolução, onde serão carregados n elementos somente. Essa modificações permitiram a arquitetura atingir, em media, 1006.4 GOP/s e 854.6 GOP/s para as camadas convolucionais e as demais camadas da AlexNet, respectivamente.

Em Kala et al. (2019), os autores também aplicaram o algoritmo de *winograd*. Os autores propuseram a arquitetura UniWiG, onde tanto o *winograd* quando o produto de Hadamard podem ser acelarados usando o mesmo conjunto de *processing elements* [2]. A arquitetura proposta pelos autores, usando uma Virtex-7 VX690T, atingiram 407.23 GOPs executando a VGG-16 e 433.53

[1] *Tensor Processing Units*, são circuitos integrados de ações específicas, usada para realizar a aceleração de modelos de aprendizagem de máquina (GOOGLE, n.d).

[2] *Processing Elements* é equivalente aos neurônios da rede neural, são basicamente dispositivos que recebem uma quantidade de sinais de entrada e, baseado neles, gera uma saída ou não

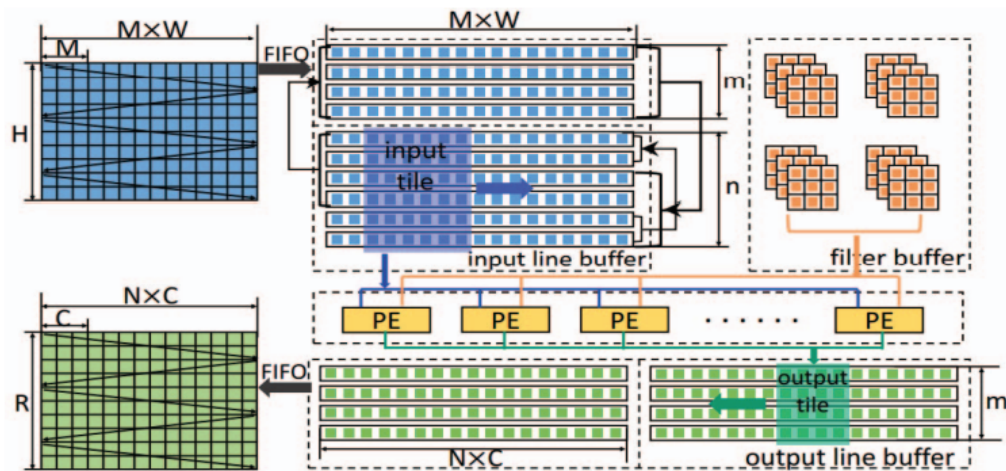

Figura 3.1 – Arquitetura proposta por (LIANG et al., 2020).

GOPs executando a AlexNet.

Em Shen et al. (2018), os autores desenvolveram uma arquitetura baseada em um esquema *work-stealing*, a fim de garantir que todos os *linear arrays* ficassem equilibrados, evitando ociosidade na arquitetura. A figura 3.2 exemplifica este processo, quando uma nova "carga"é adicionada na fila de execução de um dos *workloads* é realizada uma verificação se há *workloads* ociosas, ou com menos "cargas"a serem processadas, antes de designar o responsável pelo processamento. A arquitetura proposta pelos autores obteve 100.9 GFLOPs na camada totalmente conectada da AlexNet.

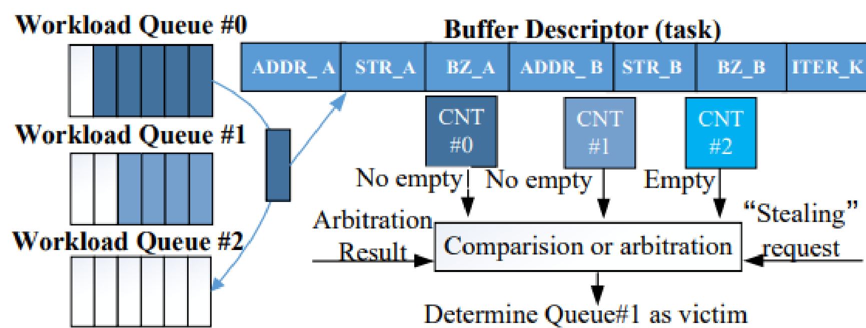

Figura 3.2 – Arquitetura do esquema *work-stealing* proposta por (SHEN et al., 2018).

Em Wang, Lin e Wang (2018), os autores introduziram as *fast convolution units*, voltadas ao cálculo de convoluções em modelos de CNN e novos modelos para o armazenamento e reuso dos pixels. Os pixels intermediarios são armazenados na memória interna, o que diminui o requerimento de banda. A implementação usou o modelo VGG-16 na placa de médio custo, ZYNQ7045, obteve uma performance de 316 GOPs e na placa Virtex VC707 atingindo 1250.21 GOPs.

Também fazendo uso da placa ZYNQ7045, em Véstias et al. (2019) é proposta uma arquitetura que faz uso de aritmética de ponto fixos, *zero-skipping*, evitando a multiplicação por zeros, e *weight-prunning*. A imagem de entrada e os *features map* gerados são armazenados na memoria interna para ser processado, caso a memória não suporte a imagem ou o *feature map* é dividido e processado em partes. Os autores também dividiram as camadas convolucionais das camadas densas, permitindo otimizações especificas para cada uma (ver figura 3.3). Essa arquitetura é capaz de realizar a inferência de uma imagem na AlexNet em 1.0ms com menos de 1% de perda na acurácia e atinge 1401 GOPs.

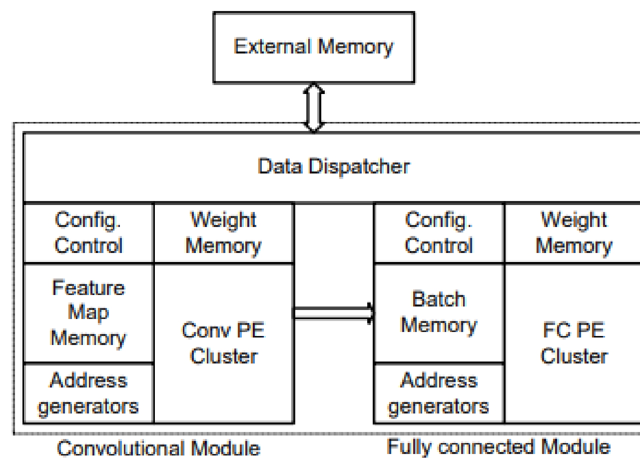

Figura 3.3 – Arquitetura proposta por (VéSTIAS et al., 2019), nela podemos notar as divisões e blocos utilizados para a camada convolucional (Conv PE) e para a camada densa (FC PE).

Alguns autores focaram em realizar inferências de redes neurais em placas de baixo custo. Em Véstias et al. (2019), a inferência de uma imagem na rede AlexNet foi realizada em 2.9ms em uma ZYNQ7020. Em Venieris e Bouganis (2019), os autores propuseram uma arquitetura que é capaz de executar redes neurais regulares e irregulares. A arquitetura é capaz de ser reconfigurada sempre que o dado passa de um subgrafo (camada) para outra, porém essa reconfiguração prejudica o tempo de execução da redes neurais, o que pode ser diminuído ao ser empregado o processamento em *batchs*. Para modelos regulares, a arquitetura atinge 38.30 GOPs para AlexNet e 48.53 GOPs para a VGG16. Já em modelos irregulares, a arquitetura atinge, usando a placa Zynq-7045, para o modelo GoogLeNet 165 GOPs e 155.57 GOPs para a DenseNet-161. Na figura 3.4 podemos notar esta divisão de reconfigurações realizadas, onde cada quadrado vermelho pontilhado é um subgrafo, e o bloco a esquerda mostra quais estruturas que foram gerada pela arquitetura para um subgrafo, enquanto o bloco a direita mostra a arquitetura gerada completa para um determinado subgrafo.

Em, Guo et al. (2018), os autores propuseram uma estraégia de *data-quantization*, que reduz o *bit-width* para 8-bit sem perda significativa de acurácia e um acelerador de CNNs flexível.

Em Véstias et al. (2018), os autores propuseram a Lite-CNN (ver figura 3.5), uma arqui-

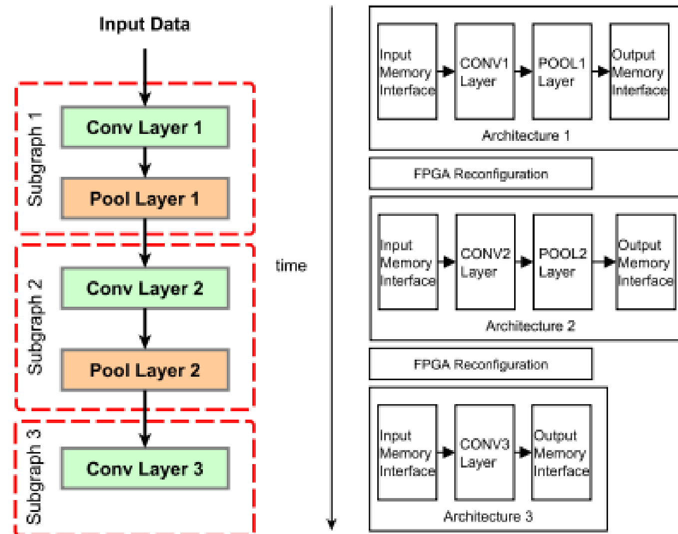

Figura 3.4 – Arquitetura de reconfiguração proposta por (VENIERIS; BOUGANIS, 2019).

tetura que faz uso de representação dos dados em 8-bit e realiza uma reorganização dos produtos escalares, diminuindo na metade as multiplicações realizadas, porém com o problema que, metade do tempo é gasto carregando os pesos da camada densa. A arquitetura teve performance de 410 GOPs na AlexNet em uma ZYNQ7020. Em Gonçalves, Peres e Véstias (2019), foi realizada algumas otimizações na arquitetura proposta por Véstias et al. (2018), a fim de suportar *hybrid quantization*. Na arquitetura usando camadas 4×4 é obtido 819 GOPs . Essa arquitetura é capaz de inferir uma imagem na AlexNet em 7.4ms com a placa ZYNQ7020.

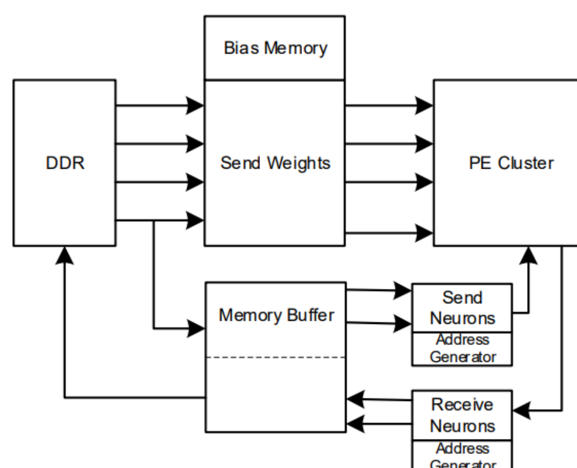

Figura 3.5 – Arquitetura da Lite-CNN proposta por (VéSTIAS et al., 2018). Nela podemos notar a comunicação entre os blocos e a memória externa, que recebe e envia pesos para o *Memory Buffer*.

Em Fuhrmann (2018), é proposta uma arquitetura similar ao TPU (JOUPPI et al., 2017)

(ver figura 3.6). Deste modo, a arquitetura é um coprocessador, que não permite a possibilidade de execução de um programa de forma independente. Deste modo, as instruções são armazenadas na memoria FIFO. A arquitetura é constituída de: Um *buffer* unificado, responsável por armazenar os *feature maps*, um *weight buffer*, responsável por armazenar os pesos das camadas, uma unidade de multiplicação sistólica, um acumulador e uma unidade de ativação (ReLU e Sigmoid). Essa arquitetura pode atingir, usando matrizes 14×14 aproximadamente 72 GOPs usando a placa ZYNQ7020.

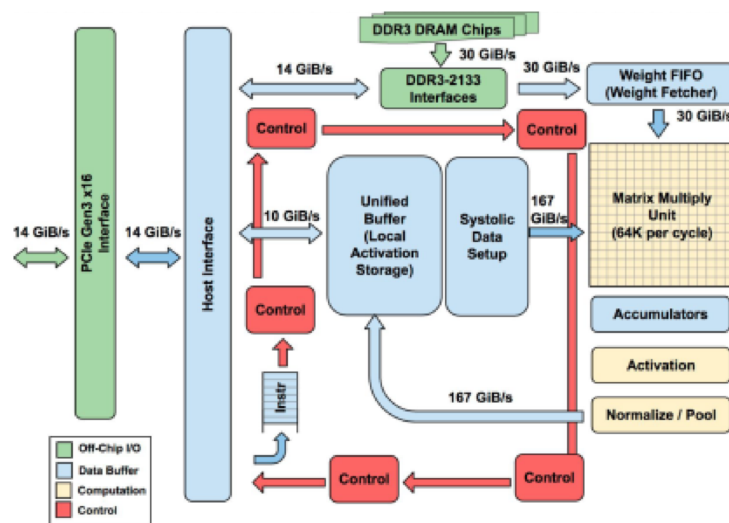

Figura 3.6 – Arquitetura da TPU proposta por Jouppi et al. (2017) e usada como base em Fuhrmann (2018).

Em Korol (2019) é proposta uma arquitetura para execução da AlexNet na placa Virtex 7 XC7VX690T. Como pode ser observado na figura 3.7 a arquitetura possui três camadas, sendo a primeira a camada de entrada de dados, a segunda executa as funções relacionada a ela e a terceira é a camada *multilayer* que executa diversas camadas da AlexNet. As camadas estão conectadas usando a ideia "*ping-pong*"(ver figura 3.8), no qual enquanto uma memória recebe os resultados da convolução, a outra memória fica livre podendo ser lida para realização de outras operações (i.e *max-pooling*). Além disso, o autor escolhe para unidade de multiplicação usar uma *multiply-add tree*, a fim de economizar recursos.

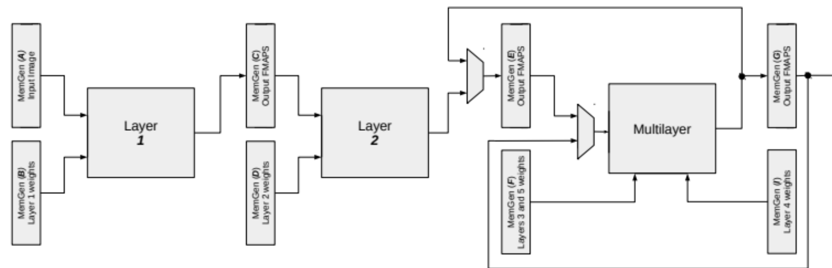

Figura 3.7 – Arquitetura proposta por Korol (2019).

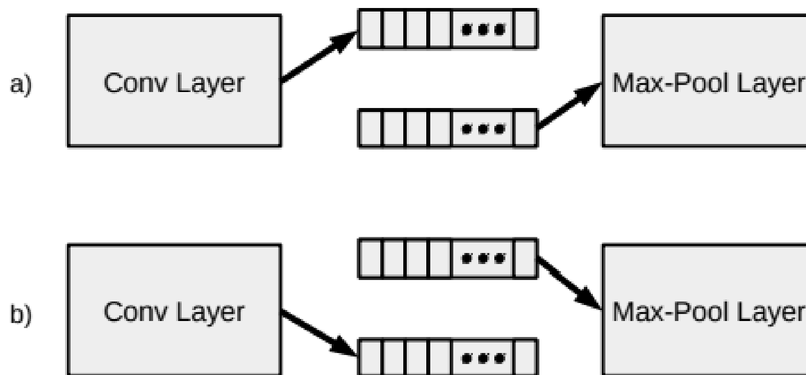

Figura 3.8 – Exemplo do esquema "*ping-pong*"para comunicação entre camadas convolucionais e *Max-Pooling* Korol (2019). Enquanto um bloco de memoria recebe o resultado da camada de convolução, outro bloco envia os dados para a camada de *pooling*.

4 Desenvolvimento do TiNN

Este trabalho de monografia possui, como objetivo geral, o desenvolvimento e a validação de uma arquitetura de *hardware* para execução de CNNs. Para tal, a arquitetura proposta neste trabalho busca englobar as vantagens já obtidas em trabalhos anteriores, porém usando uma placa de baixa densidade. O modelo de acesso a memória externa é similar ao proposto por Véstias et al. (2018) e Gonçalves, Peres e Véstias (2019), enquanto a unidade de multipliação faz uso de *mutiply-add tree*, como usada em Korol (2019) e o *zero-skipping* usado em Véstias et al. (2019), evitando assim multiplicações por zero e consequentemente o chaveamento desnecessário de registros. E as unidades de memória serão divididas em duas principais: *Data Buffer* e *Weight Buffer*, similar ao Fuhrmann (2018).

Desta forma, este capítulo apresenta uma proposta de versão inicial e funcional do TiNN, onde a Seção 4.1 detalha cada unidade necessária para execução.

4.1 Arquitetura proposta do TiNN

A arquitetura TiNN consiste em um *cluster* de *processing elements* (PE) para realização do cálculo da multiplicação de matrizes *element-wise*, que são projetados para realizar uma operação MAC (Multiplicação e soma) a cada ciclo de *clock*, um *buffer* de memória para armazenar na memória interna os valores de entrada iniciais e o OFM, uma memória para armazenar o conjunto de instruções a serem executadas e o módulo para realizar a operação de e ativação (ver figura 4.1).

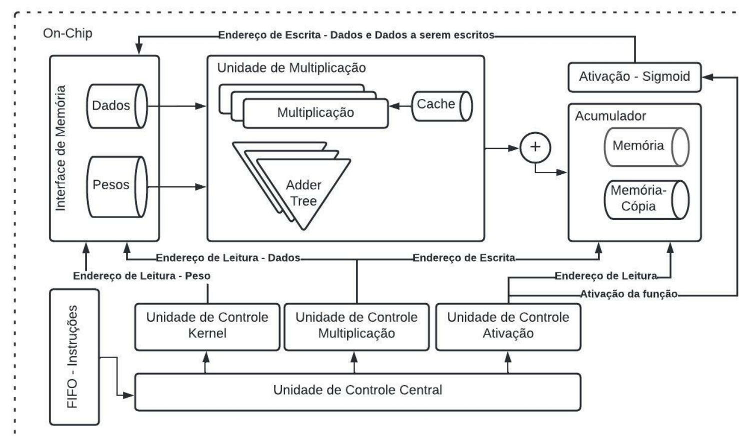

Figura 4.1 – Arquitetura Proposta

FONTE: Elaborado pelo autor

4.2 Interface de Memórias

A interface de memória é composta pela *Kernel Memory* e *Data Memory*. Os dados que são armazenados em ambas memórias são representados em 8 bits e fazem uso do *Fixed-Point Format*. Os dados são agrupados em k *clusters* de tamanho s, permitindo assim a leitura em *batch* dos mesmos. O preenchimento de cada *cluster* é feito considerando as linhas do IFM de forma sequencial e o tamanho do filtro maior. Por exemplo, na AlexNet, o maior tamanho de um filtro é 11×11, neste caso, em cada *cluster* da memória terá 11×8 bits e cada IFM na primeira camada de convolução tem 224 valores em cada linha. Logo, se imaginarmos a memória como uma matriz de *clusters*, teríamos para a AlexNet, na primeira camada de convolução, e considerando somente a primeira linha do IFM, 21 *clusters*.

Para leitura e escrita, ambas memória possuem vetores booleanos de *enable*. Esses vetores são responsáveis por ativar uma posição da memória para escrita ou leitura. Usando o exemplo anterior para visualização, temos que um *cluster* da memória irá possuir 11 valores de 8 bits cada. Deste modo, os vetores de *enable* possuirão tamanho 11 e, por exemplo, se a posição 0 no *cluster* for verdadeira e as demais posições serem falsas, para o vetor de escrita, a posição 0 irá receber um novo valor, sem alterar os demais valores do *cluster*.

O *Kernel Memory* é responsável por armazenar os pesos que serão utilizados pela *Multiplication Matrix Unit*. Nela é possível ler ou escrever um bloco de pesos por vez (ver figura 4.2).

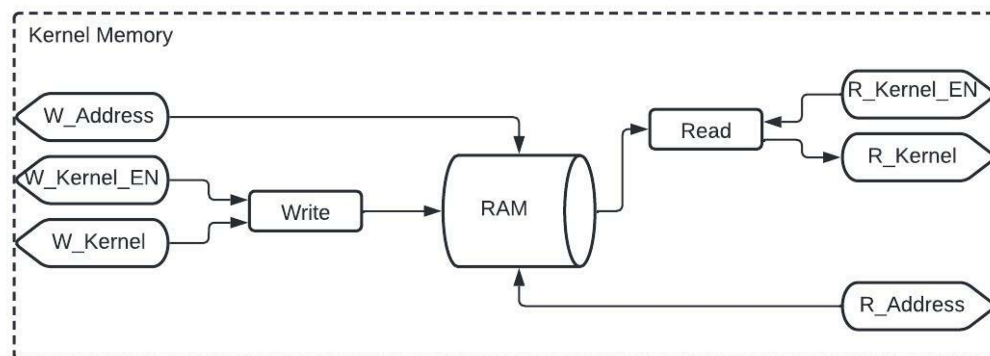

Figura 4.2 – Diagrama de bloco do *Kernel Memory*

FONTE: Elaborado pelo autor

O *Data Memory* (ver figura 4.3) é responsável por armazenar a entrada da imagem e o OFM. A mesma abordagem usada no projeto do *Kernel Memory* foi empregada aqui, porém como o *Data Memory* precisa armazenar os resultados das instruções realizadas na FPGA, é necessário ter uma porta com maior prioridade. Esta porta com maior prioridade irá ser usada para realizar a escrita de novos dados, vindos da memória externa, na memória interna. Assim como na *Kernel Memory*, a *Data Memory* só pode realizar uma operação de *Read* ou de *Write*.

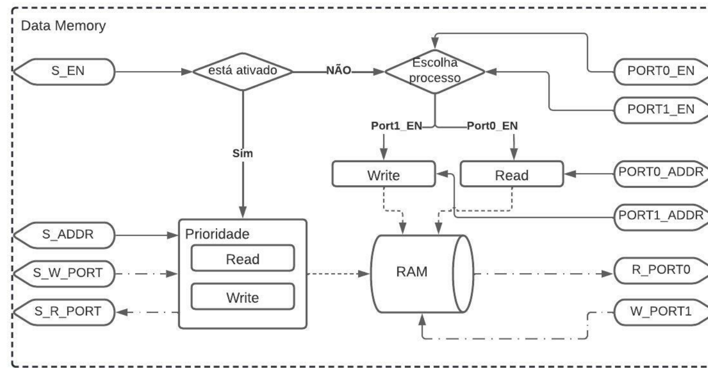

Figura 4.3 – Diagrama de bloco do *Data Memory*

FONTE: Elaborado pelo autor

4.3 Unidade de Multiplicação de Matrizes

Nas CNNs, temos um conjunto de operações de Multiplicação e soma (MACCs). Esta operação é usada na operação de convolução. Geralmente, as operações de multiplicação e acumulação são construídas adotando matrizes sistólicas e usando blocos MACCs em uma estrutura de grade. Porém, neste projeto, a implementação das operações de Multiplicação e soma foi feita usando árvore, chamada *Multiply Adder Tree*. Nesta abordagem, usamos uma linha de multiplicadores e uma árvore somadora. Os resultados de dois blocos de multiplicação, nas folhas, são adicionados ao próximo nível. Assim, os resultados viajam na árvore, até a raiz, que produz o resultado final (KOROL, 2019).

Para um CNN com o maior filtro usado de tamanho $s \times s$, será gerado s *Adder-Trees* com s blocos de multiplicação nas folhas, e $floor(log_2 s)$ blocos de profundidade de acumulação. Se s não for par, então é gerada outra camada com $floor(log_2 s) - 1$ registradores, para poder realizar todas as operações de acumulação (ver figura 4.4).

Na primeira linha (nós folhas) é realizada a multiplicação entre os pesos (vindos do *Kernel Memory*) e dos IFM (vindos do Data Memory) e nas linhas seguintes é realizada a soma entre os valores resultantes das multiplicações e das somas realizadas até atingir o nó raiz. Os resultados obtidos após as multiplicações e somas são enviados para a unidade de acumulação, onde é realizada a acumulação com os valores já presentes na memória ou a sobrescrita de uma posição da memória.

Uma memória *cache*, que é responsável por auxiliar no processo de *Stride*, foi adicionada para permitir o acesso rápido a dados anteriores. Dado um kernel $k \times k$, um IFM $n \times n$ e um stride de s, temos que a janela do kernel irá se deslocar sobre o IFM, no pior caso, com $s = 1$, onde a memória cache irá armazenar $n \times k \times (n - 1)$ dados.

Na Unidade de Multiplicação também é implementada a função *Zero-skipping* (0SKP), responsável pela verificação do dado, vindo do *Data Memory*, ou peso, vindo do *Kernel Memory*,

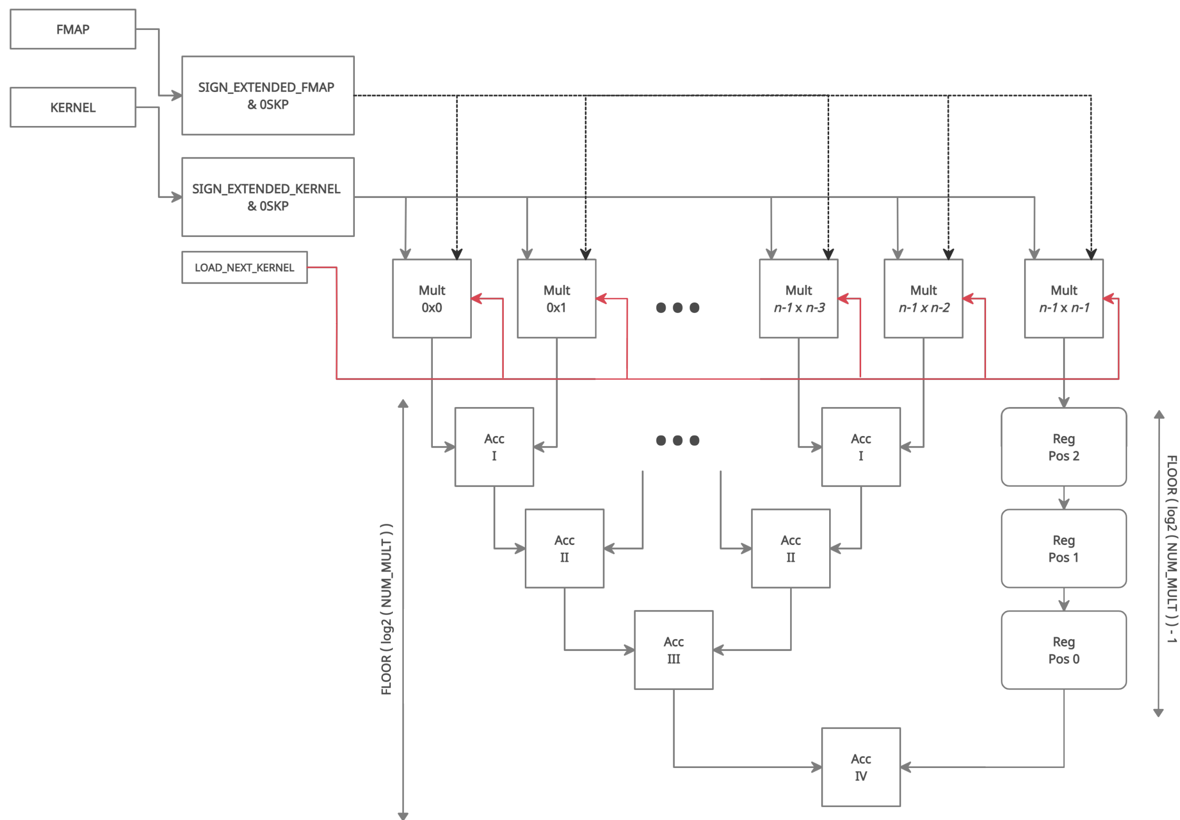

Figura 4.4 – Diagrama de bloco de uma unidade de multiplicação com maior filtro sendo $n - 1 \times n - 1$

FONTE: Elaborado pelo autor

ser igual a zero, evitando assim multiplicações com esse valor, que causaria o chaveamento dos registradores.

4.4 Acumuladores

Para realizar a acumulação dos resultados da multiplicação realizada em uma camada anterior, foram utilizados acumuladores. Os acumuladores são usados para somar os OFM gerados em uma camada, após a multiplicação do IFM com as matrizes de peso, resultando em um OFM final. Para realizar essa acumulação são usadas duas memórias, pois o resultado da acumulação é salvo no endereço transferido. Sendo assim, como duas portas (gravação e leitura simultânea) são necessárias, a memória para a saída dos acumuladores deve ser redundante, para que um dado do acumulador que será utilizado em sequência, não seja sobrescrito.

4.5 Ativação

A função de ativação implementada foi a *Sigmoid*, para tanto foi usada uma *look-up-table* e o valor da função pode ser lida com a entrada. Deste modo, o resultado da função não precisa ser calculado.

4.6 Unidades de Controle

Para controlar todas as operações, a arquitetura possui, quatro blocos de unidades de controle: *Kernel Memory Control* (KMC), *Matrix Multiplication Control Unit* (MMCU), *Activation Control Unit* (ACU) e *Central Control Unit* (CCU).

O *Kernel Memory Control* (KMC) é responsável por gerar os endereços que serão lidos do *Kernel Memory* e enviados para a *Matrix Multiplication Unit*.

A *Matrix Multiplication Control Unit* (MMCU) controla de qual endereço, quais dados, em um *cluster* lido do *Data Memory*, será enviado para a *Matrix Multiplication Unit*. Controla também quais blocos da unidade de multiplicação serão ativados. Para este controle ocorrer a MMCU possuí duas FSM: Uma responsável por realizar a operação de *stride*, e definir quais dados de um *cluster* será lido; e a outra responsável por ativar um bloco da unidade de multiplicação.

A arquitetura aceita *strides* de tamanho entre 1 e o tamanho do kernel k e a definição de quais posições em um *cluster* será lida irá depender do tamanho do *stride* e do filtro sendo usado. Por exemplo, considerando um *cluster* de tamanho 4, um filtro de tamanho 2×2 e uma operação utilizando *stride* de tamanho 2, temos que as ativações de leitura, será sempre do tamanho do *stride*, pois tanto o filtro quanto o *stride* são pares (ver figura 4.5). Caso o o filtro ou o *stride* ser ímpar, temos que alguns *enable* de leitura deverá ler todo o resto dos dados do *cluster* se o próximo *stride* passar do tamanho do *cluster* (ver figura 4.6).

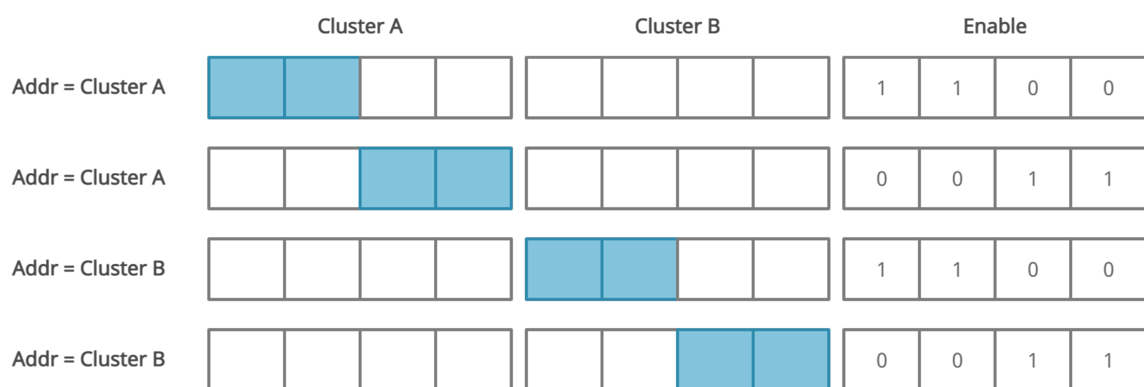

Figura 4.5 – Exemplo de *stride* de tamanho 2 e memória de tamanho 4.

FONTE: Elaborado pelo autor

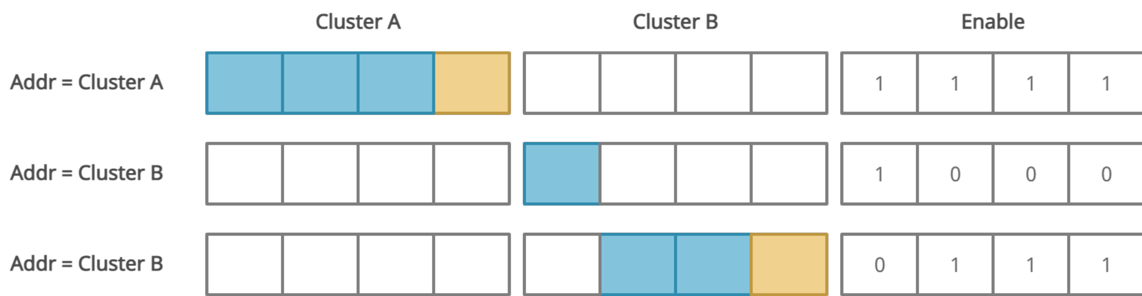

Figura 4.6 – Exemplo de *stride* de tamanho 3 e memória de tamanho 4, sendo o bloco em amarelo o dado extra que será lido.

FONTE: Elaborado pelo autor

Com a adoção da operação de *stride* como o ativador de leitura no *Data Memory* fazemos com que ocorra menos leituras na memória, visto que, na maioria dos modelos de CNNs o tamanho do *stride* é menor que o tamanho da janela do filtro, tendo assim, leituras em posições que já foram acessadas anteriormente e o dado já se encontra na memória. Deste modo, é necessário realizar somente uma operação de *shift* com os dados carregados anteriormente e o *load* com os dados lidos da memória. A figura 4.7 exemplifica a operação de *shift*, nela temos o filtro de tamanho igual a 4 e o *stride* igual a 2. Considerando que no registrador tenha os dados carregados anteriormente, temos que dois destes dados irão ser utilizados na próxima operação, sendo então, necessário carregar somente os outros dois dados restantes para completar o tamanho do filtro.

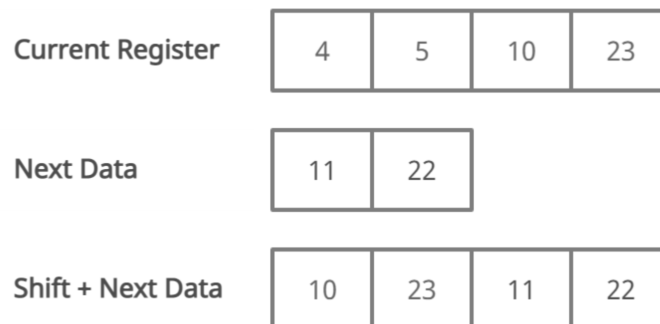

Figura 4.7 – Exemplo da operação de *shift* com *stride* igual a 2 e um filtro 4×4. O *current register* possuí os valores carregados na última operação realizada, como o *stride* é igual a 2, para completar 4 dados para realização da próxima operação é necessário o carregamento de somente outros 2 valores

FONTE: Elaborado pelo autor

A segunda FSM é responsável por realizar o controle de quais blocos de multiplicação serão ativados. Como foi dito anteriormente, na Unidade de Multiplicação temos s blocos com uma *Multiply Adder Tree* implementada. Esses blocos podem ser vistos como a linha da matriz

do filtro, ou seja, o primeiro bloco teria os s valores da primeira linha, o segundo bloco s valores da segunda e assim suscetivamente.

Além disto, a MMCU é responsável por gerar os endereços do acumulador no qual irá ser armazenada o resultado da operação de convolução.

A *Activation Control Unit* (ACU) é responsável por decodificar a instrução de ativação e definir qual função de ativação será usada, o endereço de leitura do dado da memória do acumulador e definir em qual endereço o dado ativado será armazenado no *Data Memory*.

A *Central Control Unit* (CCU) traduz as instruções do FIFO, decodifica e determina qual unidade de controle deve ser ativada. A leitura de novas instruções da FIFO só pode ocorrer se os sinais de *Busy* e *Resource Busy*, ambos vindos das demais unidades de controle, estão desativados, caso contrário a FIFO se torna bloqueada até o fim da operação que está sendo realizada.

4.7 Memória de Instruções - FIFO

Como o tempo de execução varia e o sistema *host* trabalha em paralelo com o TiNN, uma memória FIFO é introduzida para as instruções. Aqui, novas instruções são armazenadas em ordem e, em seguida, acessadas pelo coordenador. Como a BRAM é usada em outros componentes, a memória da FIFO é implementada usando LUTRAM.

4.8 Conjunto de Instruções

O conjunto de instruções adotado é similar ao usado em Fuhrmann (2018).

Existem três tipos de instruções. Eles possuem campos de bits com formato diferente, porém possuem o mesmo comprimento. O primeiro tipo de instrução, a instrução padrão (ver figura 4.8), possuí os campos: OP-Code (especificação da operação), comprimento dos dados, endereço do Acumulador e endereço da *Data Memory*.

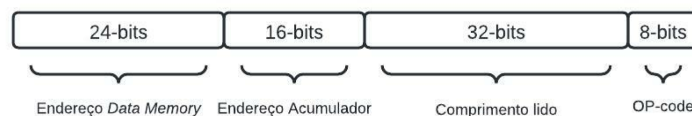

Figura 4.8 – Tipo de instrução padrão

FONTE: Elaborado pelo autor

O segundo tipo de instrução é a de peso (ver figura 4.9). Nesta instrução os campos de endereço da instrução padrão são combinadas em um campo de endereço maior.

Figura 4.9 – Tipo de instrução de Peso

FONTE: Elaborado pelo autor

O último tipo de instrução é o *Stride* (ver figura 4.10). Nele, teremos apenas o Op-code e a quantidade de *strides* que irá ocorrer em uma linha do IFM, o que é chamado de *jump*. Neste caso, como o stride, geralmente, não é um valor maior que 31, o *jump* terá apenas 5-bits.

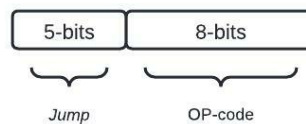

Figura 4.10 – Tipo de instrução de *Stride*

FONTE: Elaborado pelo autor

O Op-code é lido de acordo com a prioridade, logo o mesmo é dividido em 7 versões possíveis. O conjunto de bits alto marca o tipo de instrução que será realizada, com exceção dos comandos: desativado, no qual representa a arquitetura sem nenhuma instrução sendo realizada e sincronizado, no qual representa a execução de todas instruções presentes na FIFO. A lista de instruções disponíveis pode ser visto na tabela 4.1.

Op-Code	Significado	Endereço Data Memory	Acumulador
00000000	Desativado	-	-
0000001x	*Halt*	-	-
0000011x	*Stride*	-	-
00001xxx	*KMCU*	Usa 40-bits	Usa 40-bits
001xxxxx	*MMCU*	Usado	Usado
1xxxxxxx	*ACU*	Usado	Usado
11111111	Sincronizado	-	-

Tabela 4.1 – Lista de prioridade dos tipos de instruções e seus campos de bits

4.9 Interface AMBA AXI

O padrão *Advanced eXtensible Interface* (AXI) da AMBA (*Advanced Microcontroller Bus Architecture*) é usado para comunicação no chip, pois a mesma é suportada na maioria das FPGAs/SoCs (ARM, 2011). Na arquitetura proposta, é usado o AXI4-Lite, que permite

Global	Canal de Endereço de Escrita	Canal de Escrita de dados	Canal de Escrita de resposta	Canal de Leitura de endereço	Canal de Leitura de dados
ACLK	AWVALID	WVALID	BVALD	ARVALID	RVALID
ARESETn	AWREADY	WREADY	BREADY	ARREADY	RREADY
-	AWADDR	WDATA	BRESP	ARADDR	RDATA
-	AWPROT	WSTRB	-	ARPROT	RRESP

Tabela 4.2 – Sinais da interface AXI4-Lite (ARM, 2011)

uma interface AXI totalmente funcional com a lógica mais simples possível. Os sinais mais importantes desta interface são explicados brevemente abaixo.

A interface do AXI4-Lite suporta 5 canais - canal de endereço de gravação, canal de dados de gravação, canal de resposta de gravação, canal de endereço de leitura e canal de dados de leitura. Cada canal possui seus próprios sinais para o fluxo de controle (ARM, 2011).

Os sinais VALID sinalizam quando um valor é estável no barramento de transmissão, enquanto READY sinaliza que um dado foi escolhido assim que o VALID for aplicado. Caso os sinais de controle forem usados incorretamente, pode ocorrer um *deadlock* (uma situação em que dois ou mais processos ficam impedidos de continuar suas execuções).

5 Resultados

Este Capítulo apresenta os resultados relacionados à arquitetura proposta. Primeiro, é fornecida a utilização dos recursos da FPGA. Em seguida, é detalhado o processo de validação. Por fim, são expostas algumas limitações encontradas durante este trabalho, bem como possíveis soluções.

5.1 Software Usados

A arquitetura proposta foi implementada usando a linguagem de programação VHDL, o *software* Active-HDL (ALDEC, 2022) para a realização dos testes e para o processo de síntese e implementação foi usado o *software* Vivado 2020.2 (XILINX, 2022a).

5.2 Treinamento e Modelo

O modelo criado para o treinamento da rede usada na avaliação da arquitetura foi implementado usando o Pytorch (PASZKE et al., 2019). Para o treinamento e avaliação foi usado o conjunto de dados MNIST (DENG, 2012).

5.2.1 Treinamento

O conjunto de dados MNIST é frequentemente usado em *benchmarks* para avaliação da precisão de um modelo de *machine learning*. Ele consiste em 70000 amostras de imagens de dígitos manuscritos no formato 28×28 em escala de cinza. Destas 70000 amostras, 60000 são usadas para treinamento e 10000 para avaliação e teste.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1             [-1, 14, 8, 8]           2,758
            Linear-2                   [-1, 10]           8,970
================================================================
Total params: 11,728
Trainable params: 11,728
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.01
Params size (MB): 0.04
Estimated Total Size (MB): 0.05
----------------------------------------------------------------
```

Figura 5.1 – Modelo usado na avaliação da arquitetura

FONTE: Elaborado pelo autor

5.2.2 Modelo

O modelo usado para avaliação da rede possuí duas camadas (ver figura 5.1). A primeira é uma camada convolucional com *kernel* = 14, *stride* = 2 e uma ativação *softmax*, conectada a uma camada *fully-connected*, por sua vez, tem como saída uma das 10 classes do problema MNIST.

Para ser possível utilizar os pesos treinados na arquitetura proposta é necessário realizar a quantização dos pesos, pois a rede é treinada em ponto flutuante e a arquitetura faz uso dos pesos em inteiros de 8-bit. A quantização é feita usando a seguinte fórmula:

$$W_{int_8} = \{\frac{x}{128} | x \in N, -128 \leq x \leq 127\} \quad (5.1)$$

5.3 Utilização dos Recursos e Consumo de Energia

A tabela 5.1 apresenta a quantidade de recursos usadas pela arquitetura, obtida após o processo de síntese, usando como base o tamanho do bloco de multiplicação de 14×14. A arquitetura, que não possuí a implementação da camada de *pooling* nem da camada *fully-connected*, faz uso de 95.71% e 36.74% dos blocos de BRAM e LUT disponíveis, respectivamente. A implementação das unidades de multiplicação foi sintetizada em LUTs, sendo responsável por 72.74% do seu uso.

Recurso	Usado	Disponivel	Utilizado (%)
LUT	19545	53200	36.74
LUTRAM	171	17400	0.98
FF	19538	106400	18.91
BRAM	134	140	95.71
DSP	20	220	9.09

Tabela 5.1 – Uso de recursos para a arquitetura 14x14

Em relação ao uso dos demais recursos, temos que a arquitetura fez uso de apenas 20 blocos DSP. Já a para a lógica sintetizada em *Flip-Flops*, foi obtido 66.82% de uso na MMU, 13.97% na unidade de Acumulação, 5.14% nas unidades de controle, 4.34% usados na unidade de Ativação, 1.72% usados nos blocos de memória e 8.01% usados para os demais blocos da arquitetura. O uso de recursos por cada bloco da arquitetura 14×14 pode ser visto na figura 5.2.

O consumo de energia pode ser visto na figura 5.4 com maior detalhe. Para a arquitetura 14×14 fazendo uso de uma frequência de *clock* de 177.77 MHz, foi reportado, pelo processo de síntese, um consumo dinâmico em *Watts* (W) de 0.801 W e um consumo estático de 0.185 W.

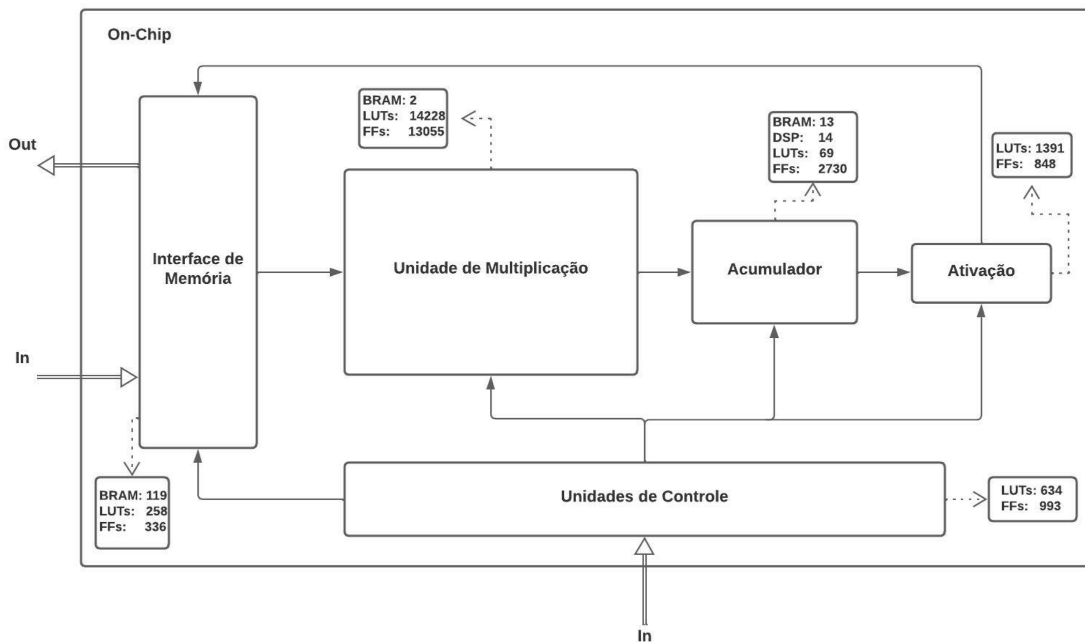

Figura 5.2 – Uso por bloco da Arquitetura 14×14

FONTE: Elaborado pelo autor

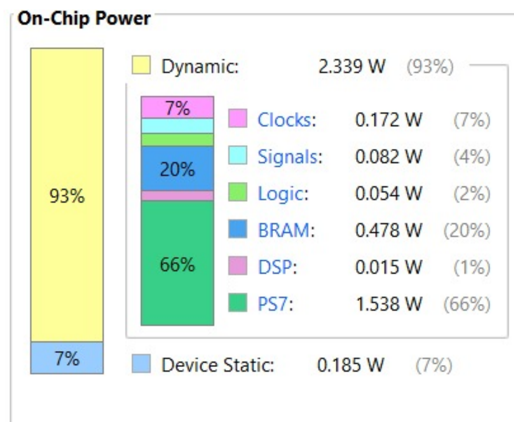

Figura 5.3 – Consumo de energia da arquitetura 14×14 proposta

FONTE: Elaborado pelo autor

5.4 Escalabilidade da Arquitetura

A arquitetura proposta pode ser sintetizada em tamanhos menores com a mesma frequência usada na arquitetura 14×14 (177.77 MHz).

A quantidade de recurso usada pela arquitetura é exibida na tabela 5.2. Nela podemos notar que, ao diminuir o tamanho do bloco de multiplicação temos a liberação de mais recursos.

Tamanho/ Recurso	**6**	**8**	**10**	**12**	**14**
LUT	6893	6987	7354	7544	19545
LUTRAM	67	76	87	103	171
FF	8233	8767	9103	9624	19538
BRAM	44	67	82	102	134
DSP	7	9	12	16	20

Tabela 5.2 – Uso de recursos de diferentes versões da Arquitetura proposta

5.5 Velocidade Teórica

A velocidade teórica é dada pelas operações realizadas pela arquitetura. Para essa medida, foi considerada as operações de multiplicação, soma e *stride* (operação de shift). Deste modo, a velocidade teórica depende de N e pode ser calculada como:

$$OPS = (N + (N - 1) + (N - S))) \times 177.77MHz \tag{5.2}$$

onde, N é o tamanho da unidade de multiplicação e S o tamanho do *stride*. Veja que, caso $S = N$, ou seja, não há compartilhamento de dados nos IFMAP, não irá ocorrer operações de *shift*. A figura 5.4 exibe a velocidade teórica calculada para a arquitetura proposta com $S = 1$.

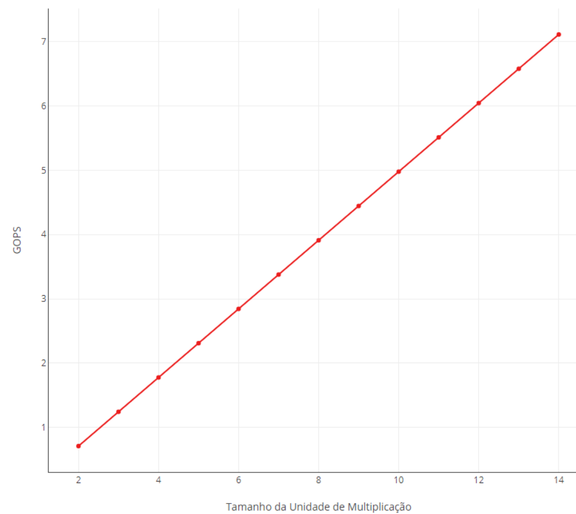

Figura 5.4 – Velocidade teórica da arquitetura proposta em GOPS dependendo do tamanho da unidade de multiplicação

FONTE: Elaborado pelo autor

5.6 Comparação com o Pytorch

Para avaliar os resultados obtidos, a acurácia obtida pela arquitetura é comparada com o Pytorch (PASZKE et al., 2019). No Pytorch, o modelo implementado obteve a acurácia de 94.47%, enquanto, a arquitetura proposta obteve a acurácia de 93.77% (ver figura 5.5). A tabela

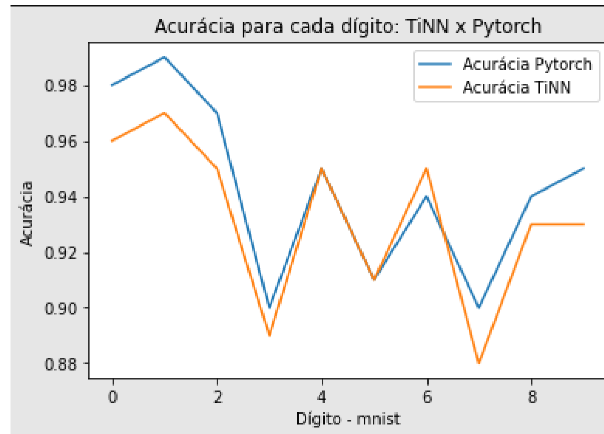

Figura 5.5 – Acurácia para a classificação de cada classe da mnist: TiNN x Pytorch

FONTE: Elaborado pelo autor

5.3 exibe a comparação entre os resultados obtidos na primeira camada de convolução do modelo executado no Pytorch e os obtidos pela arquitetura proposta.

Pytorch	**TiNN**	**Error (%)**
82.0224	81	1.2465
104.3328	104	0.3189
104.1664	104	0.1597
114.0096	113	0.8855
124.4416	125	0.4487
121.9584	122	0.0341
103.1808	102	1.1444
81.6896	82	0.3799

Tabela 5.3 – Erro entre a implementação no Pytorch e no Hardware para os resultados da camada convolucional do modelo

6 Considerações Finais

6.1 Conclusão

Como apresentado, este trabalho se propôs desenvolver uma arquitetura versátil para a execução de CNN em FPGAs de baixa densidade. Apesar do estudo da arquitetura ter sido focado na versão com a unidade de multiplicação de tamanho 14 e *stride* = 1, é possível modificar esses parâmetros e configurar outros tipos de redes convolucionais.

Para a avaliação da arquitetura proposta, o uso de recursos e o consumo de energia foram observados. Além disto, foi implementada uma rede convolucional, em *pytorch*, para realização de testes. Os testes mostraram que a arquitetura proposta consegue obter uma acurácia próxima a modelos executados em GPUs.

A arquitetura proposta, ainda embrionária, pode ser muito útil para sistemas de IoT, graças ao seu baixo consumo de energia e a pequena perda de acurácia em relação a modelos em GPUs.

6.2 Trabalhos Futuros

A principal contribuição deste trabalho é a criação de um ambiente que pode vir a ser usado para maiores explorações. Modificações podem ser realizadas na unidade de multiplicação a fim de obter melhor desempenho da arquitetura, como:

- Modificação da estrutura de árvore para fazer uso dos blocos DSP;
- Substituição da estrutura de árvore por uma estrutura sistólica, a fim de extrair o máximo das unidades de multiplicação, usando o paralelismo intrínseco fornecido por esta estrutura;
- Realizar testes com unidades de multiplicações de tamanhos menores, como 5×5 e 3×3, o que, como mostrado, liberaria mais recursos e possibilitaria o uso de múltiplas unidades de multiplicação concorrentemente;

O alto uso de LUTs pela unidade de multiplicação mostra que há espaço para melhora da arquitetura, como o uso de mais de uma árvore de multiplicação (para mais de um *kernel* carregado) na unidade de multiplicação ao otimizar a unidade de multiplicação para fazer uso dos blocos DSP. Deste modo liberando mais blocos de LUTs para poder ser usado em outras implementações.

Além dos pontos listados acima, outras extensões a serem estudas são a implementação das camadas de *pooling* e *fully-connected* e a flexibilização do bloco de *stride*. As duas primeiras camadas foram discutidas neste trabalho, mas não foram implementadas. E o *stride* adotado pela

arquitetura atual é definido pelo usuário no processo de síntese e deve ser fixo para todas as camadas da arquitetura.

Por fim, uma análise de como contornar o alto uso dos blocos de BRAM deve ser realizada, a fim de ser possível o uso da arquitetura para redes mais complexas.

Referências

ALDEC. *Active HDL*. 2022. <https://www.aldec.com/en/products/fpga_simulation/active_hdl_student/>. Accessed: 2022-05-29.

ARM. *AMBA AXI and ACE protocol Specification*. 2011.

BERGAMAN, A.; IYENGAR, J. *How COVID-19 is affecting internet performance*. 2020.

BOBDA, C. *Introduction to Reconfigurable Computing: Architectures, Algorithms, and Applications*. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2007. ISBN 1402060882.

DENG, L. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, IEEE, v. 29, n. 6, p. 141–142, 2012.

FUHRMANN, J. Implementierung einer tensor processing unit mitdem fokus auf embedded systems und das internet of things. 2018. Disponível em: <https://reposit.haw-hamburg.de/handle/20.500.12738/8527>.

GONçALVES, A.; PERES, T.; VéSTIAS, M. P. Exploring data size to run convolutional neural networks in low density fpgas. In: HOCHBERGER, C.; NELSON, B.; KOCH, A.; WOODS, R. F.; DINIZ, P. C. (Ed.). *ARC*. Springer, 2019. (Lecture Notes in Computer Science, v. 11444), p. 387–401. ISBN 978-3-030-17227-5. Disponível em: <http://dblp.uni-trier.de/db/conf/arc/arc2019.html#GoncalvesPV19>.

GOODFELLOW, I. J.; BENGIO, Y.; COURVILLE, A. *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016. <http://www.deeplearningbook.org>.

GOOGLE. *Cloud Tensor Processing Units (Cloud TPUs)*. n.d. <https://cloud.google.com/tpu/docs/tpus>. Accessed: 2021-10-10.

GRIGORESCU, S. M.; TRASNEA, B.; COCIAS, T. T.; MACESANU, G. A survey of deep learning techniques for autonomous driving. *ArXiv*, abs/1910.07738, 2020.

GUO, K.; SUI, L.; QIU, J.; YU, J.; WANG, J.; YAO, S.; HAN, S.; WANG, Y.; YANG, H. Angel-eye: A complete design flow for mapping cnn onto embedded fpga. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, v. 37, n. 1, p. 35–47, 2018. Disponível em: <http://dblp.uni-trier.de/db/journals/tcad/tcad37.html#GuoSQYWYHWY18>.

HAYKIN, S. S. *Neural networks and learning machines*. Third. Upper Saddle River, NJ: Pearson Education, 2009.

HORN, R. A.; JOHNSON, C. R. *Matrix analysis*. [S.l.]: Cambridge University Press, 2012.

JOUPPI, N. P.; YOUNG, C.; PATIL, N.; PATTERSON, D. A.; AGRAWAL, G.; BAJWA, R. S.; BATES, S.; BHATIA, S.; BODEN, N. J.; BORCHERS, A.; BOYLE, R.; CANTIN, P. luc; CHAO, C.; CLARK, C.; CORIELL, J.; DALEY, M.; DAU, M.; DEAN, J.; GELB, B.; GHAEMMAGHAMI, T. V.; GOTTIPATI, R.; GULLAND, W.; HAGMANN, R. B.; HO, C. R.; HOGBERG, D.; HU, J.; HUNDT, R.; HURT, D.; IBARZ, J.; JAFFEY, A.; JAWORSKI, A.; KAPLAN, A.; KHAITAN, H.; KILLEBREW, D.; KOCH, A.; KUMAR, N.; LACY, S.;

LAUDON, J.; LAW, J.; LE, D.; LEARY, C.; LIU, Z.; LUCKE, K. A.; LUNDIN, A.; MACKEAN, G.; MAGGIORE, A.; MAHONY, M.; MILLER, K.; NAGARAJAN, R.; NARAYANASWAMI, R.; NI, R.; NIX, K.; NORRIE, T.; OMERNICK, M.; PENUKONDA, N.; PHELPS, A.; ROSS, J.; ROSS, M.; SALEK, A.; SAMADIANI, E.; SEVERN, C.; SIZIKOV, G.; SNELHAM, M.; SOUTER, J. W.; STEINBERG, D.; SWING, A.; TAN, M.; THORSON, G.; TIAN, B.; TOMA, H.; TUTTLE, E.; VASUDEVAN, V.; WALTER, R.; WANG, W.; WILCOX, E.; YOON, D. H. In-datacenter performance analysis of a tensor processing unit. *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, p. 1–12, 2017.

KALA, S.; JOSE, B. R.; MATHEW, J.; SIVANANDAN, N. High-performance cnn accelerator on fpga using unified winograd-gemm architecture. *IEEE Trans. Very Large Scale Integr. Syst.*, v. 27, n. 12, p. 2816–2828, 2019. Disponível em: <http://dblp.uni-trier.de/db/journals/tvlsi/tvlsi27.html#KalaJMS19>.

KOROL, G. S. *An FPGA implementation for convolutional neural network*. 2019.

LECUN, Y.; BOTTOU, L.; BENGIO, Y.; HAFFNER, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, v. 86, n. 11, p. 2278–2324, 1998.

LI, H.; FAN, X.; JIAO, L.; CAO, W.; ZHOU, X.; WANG, L. A high performance fpga-based accelerator for large-scale convolutional neural networks. In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. [S.l.: s.n.], 2016. p. 1–9.

LI, Z.; LIU, F.; YANG, W.; PENG, S.; ZHOU, J. A survey of convolutional neural networks: analysis, applications, and prospects. *IEEE Transactions on Neural Networks and Learning Systems*, IEEE, 2021.

LI, Z.; YANG, W.; PENG, S.; LIU, F. A survey of convolutional neural networks: Analysis, applications, and prospects. *CoRR*, abs/2004.02806, 2020. Disponível em: <https://arxiv.org/abs/2004.02806>.

LIANG, Y.; LU, L.; XIAO, Q.; YAN, S. Evaluating fast algorithms for convolutional neural networks on fpgas. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, v. 39, n. 4, p. 857–870, 2020. Disponível em: <http://dblp.uni-trier.de/db/journals/tcad/tcad39.html#LiangLXY20>.

MAHESH, P.; PRATHYUSHA, Y. G.; SAHITHI, B.; NAGENDRAM, S. Covid-19 detection from chest x-ray using convolution neural networks. In: IOP PUBLISHING. *Journal of Physics: Conference Series*. [S.l.], 2021. v. 1804, n. 1, p. 012197.

MAHESH, P.; PRATHYUSHA, Y. G.; SAHITHI, B.; NAGENDRAM, S. Covid-19 detection from chest x-ray using convolution neural networks. *Journal of Physics: Conference Series*, IOP Publishing, v. 1804, n. 1, p. 012197, feb 2021. Disponível em: <https://doi.org/10.1088/1742-6596/1804/1/012197>.

MANSUR, R. L. *FPGA – A Flexibilidade No Projeto De Hardware. Parte 1 De 3*. 2016. <http://www2.decom.ufop.br/imobilis/fpga-o-prodigio-de-flexibilidade/>. Accessed: 2021-10-07.

MILLION, E. *The Hadamard Product*. 2007. <http://buzzard.ups.edu/courses/2007spring/projects/million-paper.pdf>. Accessed: 2021-09-18.

MITCHELL, T. M. *Machine Learning*. New York: McGraw-Hill, 1997. ISBN 978-0-07-042807-2.

MONTENEGRO, H. *Fully connected layer*. 2020. <https://www.fastaireference.com/tabular-data/fully-connected-layer>. Accessed: 2021-10-07.

MULLAPUDI, R. T.; MARK, W. R.; SHAZEER, N.; FATAHALIAN, K. Hydranets: Specialized dynamic architectures for efficient inference. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. [S.l.: s.n.], 2018.

NAIR, V.; HINTON, G. E. Rectified linear units improve restricted boltzmann machines. In: *ICML*. [S.l.: s.n.], 2010.

NWANKPA, C.; IJOMAH, W.; GACHAGAN, A.; MARSHALL, S. *Activation Functions: Comparison of trends in Practice and Research for Deep Learning*. 2018.

PASZKE, A.; GROSS, S.; MASSA, F.; LERER, A.; BRADBURY, J.; CHANAN, G.; KILLEEN, T.; LIN, Z.; GIMELSHEIN, N.; ANTIGA, L.; DESMAISON, A.; KOPF, A.; YANG, E.; DEVITO, Z.; RAISON, M.; TEJANI, A.; CHILAMKURTHY, S.; STEINER, B.; FANG, L.; BAI, J.; CHINTALA, S. Pytorch: An imperative style, high-performance deep learning library. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019. p. 8024–8035. Disponível em: <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>.

PENG, Y. Application of convolutional neural network in intrusion detection. In: IEEE. *2020 International Conference on Advance in Ambient Computing and Intelligence (ICAACI)*. [S.l.], 2020. p. 169–172.

PENG, Y. Application of convolutional neural network in intrusion detection. In: *2020 International Conference on Advance in Ambient Computing and Intelligence (ICAACI)*. [S.l.: s.n.], 2020. p. 169–172.

REN, J.; ZHANG, D.; HE, S.; ZHANG, Y.; LI, T. Edge-cloud orchestrated network computing paradigms: Transparent computing, mobile edge computing, fog computing, and cloudlet. *ACM Comput. Surv.*, v. 52, 2019. Disponível em: <https://dl.acm.org/doi/10.1145/3362031>.

RUSSAKOVSKY, O.; DENG, J.; SU, H.; KRAUSE, J.; SATHEESH, S.; MA, S.; HUANG, Z.; KARPATHY, A.; KHOSLA, A.; BERNSTEIN, M. et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, Springer, v. 115, n. 3, p. 211–252, 2015.

SALEHI, A. W.; BAGLAT, P.; SHARMA, B. B.; GUPTA, G.; UPADHYA, A. A cnn model: Earlier diagnosis and classification of alzheimer disease using mri. In: IEEE. *2020 International Conference on Smart Electronics and Communication (ICOSEC)*. [S.l.], 2020. p. 156–161.

SHEN, J.; QIAO, Y.; HUANG, Y.; WEN, M.; ZHANG, C. Towards a multi-array architecture for accelerating large-scale matrix multiplication on fpgas. In: *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. [S.l.: s.n.], 2018. p. 1–5.

SINGH, S. *Fully Connected Layer: The brute force layer of a Machine Learning model*. n.d. <https://iq.opengenus.org/fully-connected-layer/>. Accessed: 2021-08-14.

SMITH, D. J.; ZAMFIRESCU, A. *HDL Chip Design: A Practical Guide for Designing, Synthesizing and Simulating ASICs and FPGAs Using VHDL or Verilog*. [S.l.]: Doone Publications, 1998. ISBN 0965193438.

SUN, Y.; WANG, X.; TANG, X. Deeply learned face representations are sparse, selective, and robust. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. [S.l.: s.n.], 2015. p. 2892–2900.

VENIERIS, S. I.; BOUGANIS, C.-S. fpgaconvnet: Mapping regular and irregular convolutional neural networks on fpgas. *IEEE Trans. Neural Networks Learn. Syst.*, v. 30, n. 2, p. 326–342, 2019. Disponível em: <http://dblp.uni-trier.de/db/journals/tnn/tnn30.html#VenierisB19>.

VéSTIAS, M.; DUARTE, R. P.; SOUSA, J. T. de; NETO, H. Lite-cnn: A high-performance architecture to execute cnns in low density fpgas. In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. [S.l.: s.n.], 2018. p. 399–3993.

VéSTIAS, M. P. A survey of convolutional neural networks on edge with reconfigurable computing. *Algorithms*, v. 12, n. 8, 2019. ISSN 1999-4893. Disponível em: <https://www.mdpi.com/1999-4893/12/8/154>.

VéSTIAS, M. P.; DUARTE, R. P.; SOUSA, J. T. de; NETO, H. C. Fast convolutional neural networks in low density fpgas using zero-skipping and weight pruning. *Electronics*, v. 8, n. 11, 2019. ISSN 2079-9292. Disponível em: <https://www.mdpi.com/2079-9292/8/11/1321>.

VéSTIAS, M. P.; DUARTE, R. P.; SOUSA, J. T. de; NETO, H. C. Moving deep learning to the edge. *Algorithms*, v. 13, n. 5, 2020. ISSN 1999-4893. Disponível em: <https://www.mdpi.com/1999-4893/13/5/125>.

WANG, J.; LIN, J.; WANG, Z. Efficient hardware architectures for deep convolutional neural network. *IEEE Trans. Circuits Syst. I Regul. Pap.*, v. 65-I, n. 6, p. 1941–1953, 2018. Disponível em: <http://dblp.uni-trier.de/db/journals/tcas/tcasI65.html#WangLW18>.

XILINX. *Xilinx, Inc.: 7 Series FPGAs Memory Resources*. 2016.

XILINX. *Vivado Design Suite User Guide Simulation*. 2018.

XILINX. *Vivado Design Suite User Guide Synthesis*. 2018.

XILINX. *Xilinx, Inc: 7 Series DSP48E1 Slice*. 2018.

XILINX. *Vivado*. 2022. <https://www.xilinx.com/support/download.html/>. Accessed: 2022-05-29.

XILINX. *What is an FPGA*. 2022. <https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html>. Accessed: 2021-10-08.

XU, J. A deep learning approach to building an intelligent video surveillance system. *Multimedia Tools and Applications*, Springer, v. 80, n. 4, p. 5495–5515, 2021.

ZEIDMAN, B. *All about FPGAs*. 2006. <https://www.eetimes.com/all-about-fpgas/>. Accessed: 2021-10-07.

ZHANG, A.; LIPTON, Z. C.; LI, M.; SMOLA, A. J. Dive into deep learning. *arXiv preprint arXiv:2106.11342*, 2021.