Pós-Graduação em Ciência da Computação

JEFFERSON RAMOS LUCAS DOS ANJOS

InferenCNN: Uma biblioteca para Inferência de Redes Neurais Convolucionais Multiplataforma em OpenCL

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

Recife
2019

JEFFERSON RAMOS LUCAS DOS ANJOS

InferenCNN: Uma biblioteca para Inferência de Redes Neurais Convolucionais Multiplataforma em OpenCL

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção de título de Mestre em Ciência da Computação

Área de Concentração: Sistemas Embarcados

Orientador(a): Edna Natividade da Silva Barros

Recife

2019

Catalogação na fonte
Bibliotecária Mariana de Souza Alves CRB4-2106

A599i Anjos, Jefferson Ramos Lucas dos.
InferenCNN: Uma biblioteca para Inferência de Redes Neurais Convolucionais Multiplataforma em OpenCL/ Jefferson Ramos Lucas dos Anjos. – 2019.
86 f.: il., fig., tab.

Orientadora: Edna Natividade da Silva Barros.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da computação. Recife, 2019.
Inclui referências.

1. Sistemas Embarcados. 2. Inferência. 3. Redes Neurais Convolucionais. 4. Múltiplos Dispositivos. I. Barros, Edna Natividade da Silva (orientadora). II. Título.

005.256 CDD (22. ed.) UFPE-MEI 2019-160

Jefferson Ramos Lucas dos Anjos

“InferenCNN: Uma biblioteca para Inferência de Redes Neurais Convolucionais Multiplataforma em OpenCL”

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 14/03/2019.

BANCA EXAMINADORA

_____

Prof. Dr. Cleber Zanchettin
Centro de Informática/UFPE

_____

Prof. Dr. Victor Wanderley Costa de Medeiros
Departamento de Estatística e Informática/UFRPE

_____

Profa. Dra. Edna Natividade da Silva Barros
Centro de Informática/UFPE
(**Orientadora**)

“Dedico primeiramente a Deus, por me guardar durante toda a minha caminhada”.

“Dedico a todos os meus professores de Graduação e Mestrado, pois foram fundamentais para o meu crescimento”.

“Dedico aos meus pais, por todo amor, e força que me deram em todos os momentos da minha vida. AMO VOCÊS!”.

“Dedico este trabalho à minha família e amigos, que sempre estiveram presentes direta ou indiretamente em todos os momentos de minha formação, orando, apoiando, e com conselhos e conversas que ficaram guardados”.

AGRADECIMENTOS

Primeiramente gostaria de agradecer à Deus por todas as bençãos que Ele tem me concedido, e aos meus pais, pelo apoio, dedicação, e conselhos que me fizeram chegar até aqui. Sem eles não seria possível nenhuma das minhas conquistas.

Gostaria de agradecer à minha professora Edna Natividade da Silva Barros, por toda a orientação, aconselhamentos, força e confiança, pois tem me ajudado desde a minha Iniciação Científica e trabalho no LINCS - Laboratório Para Integração de Circuitos e Sistemas.

Agradeço à minha namorada Emylle Rayssa, por ter seguido junto comigo todos esses anos de crescimento, tornado meus dias melhores e mais tranquilos.

Gostaria de agradecer também à todos os meus amigos do LINCS, Antonyus Pyetro, João Gabriel, João Paulo, Severino José, Henrique Figueirôa, Bruno Pessoa, Josivan Reis, Hugo Albuquerque, Igor Silva, Érika Spencer, Vanessa Ogg, Djeefther Souza, Pedro Ishimaru e Lucas Camboin, por todo trabalho, brincadeiras, e momentos especiais que passamos juntos.

Um agradecimento especial para Antonyus Pyetro e João Gabriel, que me acompanharam desde a iniciação científica, trabalhando e crescendo juntos. Sem eles, eu não teria chegado até aqui. Foram grandes parceiros e amigos, durante todo o meu crescimento profissional e acadêmico.

Gostaria de agradecer também à todos os meus amigos do trabalho, Accenture IES, pois tem me proporcionado um grande crescimento profissional.

Gostaria de agradecer ao Intel vLab Academic Cluster, por prover o ambiente para o desenvolvimento desta pesquisa em FPGA's

E, por último, mas não menos importante, agradeço à CAPES, pelo apoio financeiro.

RESUMO

As Redes Neurais Convolucionais (Convolutional Neural Networks - CNN) têm se mostrado eficazes para problemas de Visão Computacional, como: classificação, detecção e reconhecimento de padrões. Assim, muitas bibliotecas, frameworks e soluções têm sido desenvolvidas para treinar e realizar a inferência dessas redes em dispositivos de computação. A inferência é a fase de uso de uma Rede Neural, em que novos dados são propagados pela rede, já treinada, a fim de se obter uma classificação, detecção ou reconhecimento de algum padrão. Nos últimos anos muitas arquiteturas de CNN's foram desenvolvidas tendo como objetivo acelerar o treinamento e a inferência da rede neural, suprir a necessidade de problemas mais complexos e dar suporte à dispositivos de computação com limitação de memória e energia, como os dispositivos mobiles. Assim, com essa grande variedade de arquiteturas de CNN's e dispositivos de computação, os atuais frameworks não dão suporte à alguns tipos de dispositivos ou se limitam à dispositivos de um fabricante específico. Neste trabalho apresentamos a InferenCNN, uma biblioteca multiplataforma capaz de realizar a inferência dos modelos de CNN's, mapeando kernels OpenCL (Open Computing Language) em dispositivos de computação como CPU's (Central Processing Unit), GPU's (Graphics Processing Unit) e FPGA's (Field Programmable Gate Array). A utilização do framework OpenCL, permite que a biblioteca independa do dispositivo ou fabricante do dispositivo, dando flexibilidade ao executar a inferência das arquiteturas de CNN's. A InferenCNN apresenta um parser de pesos que permite a execução de arquiteturas de CNN's que foram treinadas em diferentes frameworks, como Tensorflow, Caffe e Pytorch. Apresenta também um gráfico de fluxo que executa as camadas da CNN de forma modular, deixando espaço para que modificações e otimizações possam ser feitas, e que novas camadas e arquiteturas possam ser implementadas. Os experimentos realizados analisaram a InferenCNN em dispositivos diferentes e compararam as implementações com o framework Tensorflow e aceleradores em FPGA's. Os resultados se mostraram promissores, apresentando melhoria de desempenho na inferência da arquitetura Inception quando comparada com o Tensorflow e desempenho próximo em outras arquiteturas. Quando comparada com aceleradores em FPGA a InferenCNN se mostrou com resultados inferiores em relação ao desempenho, mas com uma maior flexibilidade em realizar a inferência de várias arquiteturas de CNN.

Palavras-chave: Inferência. Redes Neurais Convolucionais. OpenCL. Múltiplos Dispositivos. Sistemas Embarcados.

ABSTRACT

Convolutional Neural Networks (CNN) have been shown to be effective for Computational Vision problems, such as: classification, detection and recognition of patterns. Thus, many libraries, frameworks, and solutions have been developed to train and infer these networks in computing devices. Inference is the phase of using a Neural Network, in which new data is propagated by the network, already trained, in order to obtain a classification, detection or recognition of some standard. In recent years many CNN architectures have been developed to accelerate neural network training and inference, address the need for more complex problems, and support memory and power-limiting computing devices such as mobile devices. Thus, with this wide variety of CNN architectures and computing devices, the current frameworks do not support some types of devices or are limited to the devices of a specific manufacturer. In this paper we present InferenCNN, a multiplatform library capable of inferring CNN models, mapping OpenCL (Open Computing Language) kernels to computing devices such as CPUs, GPUs and Field Programmable Gate Array). The use of the OpenCL framework allows the library to be independent of the device or device manufacturer, giving flexibility in performing the inference of CNN's architectures. InferenCNN presents a weight parser that allows the execution of CNN's architectures that have been trained in different frameworks, such as Tensorflow, Caffe and Pytorch. It also features a flow chart that runs the CNN layers in a modular way, leaving room for modifications and optimizations to be made, and for new layers and architectures to be implemented. The experiments carried out analyzed InferenCNN in different devices and compared the implementations with the Tensorflow framework and accelerators in FPGAs. The results were promising, presenting performance improvement in Inception architecture inference when compared to Tensorflow and near performance in other architectures. When compared with FPGA accelerators, InferenCNN was shown to be inferior in performance, but with greater flexibility in performing the inference of several CNN architectures.

Keywords: Inference. Convolutional Neural Networks. OpenCL. Multiple Devices**.**

LISTA DE FIGURAS

Figura 1 – Neurônio 18
Figura 2 – Neurônio Artificial 19
Figura 3 – Multilayer Perceptron (MLP) 20
Figura 4 – Rede Neural para Classificação de Imagens 20
Figura 5 – LeNet 22
Figura 6 – AlexNet 23
Figura 7 – Modelo de uma Rede Neural Convolucional 24
Figura 8 – Camada de Convolução 25
Figura 9 – Camada de Convolução Depthwise 26
Figura 10 – Funções de Ativação 27
Figura 11 – Batch Normalization 28
Figura 12 – Camada de Padding 29
Figura 13 – Camada de Pooling 30
Figura 14 – Camada Densa 30
Figura 15 – Camada de Softmax 31
Figura 16 – VGG 32
Figura 17 – Pinha de Camadas VGG16 32
Figura 18 – Inception 33
Figura 19 – Bloco Inception 33
Figura 20 – ResNet 34
Figura 21 – MobileNet 35
Figura 22 – Código Tensorflow 36
Figura 23 – Soma e Multiplicação de Tensores 37
Figura 24 – AlexNet em Keras 39
Figura 25 – Modelo de Plataforma OpenCL 41
Figura 26 – Modelo de Execução OpenCL 42
Figura 27 – NDRange OpenCL 43
Figura 28 – Modelo de Memória OpenCL 44
Figura 29 – Análise de precição de Ponto Fixo 47
Figura 30 – Caffeinated FPGAs 48
Figura 31 – Acceleration of Deep Learning on FPGA 49
Figura 32 – PipeCNN 50
Figura 33 – Metodologia de Projeto da InferenCNN 55
Figura 34 – Parser Tensorflow 56
Figura 35 – Arquivo de Configuração VGG16 57
Figura 36 – Configuração do Arquivo de Pesos Convolucionais 58
Figura 37 – Exemplo do arquivo de pesos para a primeira camada da VGG16 . 59
Figura 38 – Configuração do Arquivo de Pesos Densos 60

Figura 39 – Carregamento da Arquitetura da CNN 61
Figura 40 – Grafo de Fluxo para uma VGG16 61
Figura 41 – Inicialização do Programa OpenCL 62
Figura 42 – Inicialização das Filas de Comando 63
Figura 43 – Execução 63
Figura 44 – Kernel de Convolução 65
Figura 45 – Kernel de Polling 66
Figura 46 – Kernel de Totalmente Conectada 67
Figura 47 – Kernel de Batch Normalization 67
Figura 48 – Kernel de AddResidual 68
Figura 49 – Kernels de Ativação 69
Figura 50 – Kernel de Padding 70
Figura 51 – Fluxo de Projeto do SDK do Intel® FPGA para OpenCL™ 73
Figura 52 – Canônico 75
Figura 53 – OpenCL CPU 75
Figura 54 – OpenCL GPU 76
Figura 55 – OpenCL FPGA 77

LISTA DE TABELAS

Tabela 1 – Resumo dos Trabalhos Relacionados 50
Tabela 2 – Especificação Nvidia Geforce GTX 1070 71
Tabela 3 – Especificação Arria 10 GX 1150 74
Tabela 4 – Análise de Potência 78
Tabela 5 – Utilização de Recursos FPGA 78
Tabela 6 – Comparativo de Desempenho com o Tensorflow 79
Tabela 7 – Comparativo com Outros Trabalhos de FPGA's 80
Tabela 8 – Comparação de Recursos 80

LISTA DE ABREVIATURAS E SIGLAS

CNN	Convolutional Neural Network - Redes Neurais Convolucionais
CPU	Central Processing Unit - Unidade Central de Processamento
DL	Deep Learning
DSP	Digital Signal Processor
FC	Fully Connect
FPGA	Field Programmable Gate Arrays
GPU	Graphics Processing Unit
MLP	Multi-layer percepton
SDK	Software Development Kit

SUMÁRIO

1 INTRODUÇÃO . . . 14
1.1 Contexto Geral . . . 14
1.2 Contexto Específicos . . . 15
1.3 Contribuições . . . 15
1.4 Organização da dissertação . . . 16

2 FUNDAMENTAÇÃO TEÓRICA . . . 18
2.1 Redes Neurais . . . 18
2.2 Treinamento e Inferência de uma Rede Neural . . . 20
2.3 Redes Neurais Profundas . . . 21
2.3.1 Aprendizagem Profunda . . . **21**
2.3.2 Aplicação Prática das Redes Neurais Profundas . . . **22**
2.4 Redes Neurais Convolucionais . . . 23
2.4.1 Tipos de Camadas . . . **24**
2.4.1.1 *Camada de Convolução* . . . **24**
2.4.1.2 *Camada de Convolução Depthwise* . . . **25**
2.4.1.3 *Camada de Ativação* . . . **26**
2.4.1.4 *Camada de Batch Normalization* . . . **27**
2.4.1.5 *Camada de Preenchimento (Padding)* . . . **28**
2.4.1.6 *Camada de Subamostragem (Pooling)* . . . **29**
2.4.1.7 *Camada Densa* . . . **30**
2.4.1.8 *Camada Softmax* . . . **31**
2.4.1.9 *Arquiteturas de Redes Neurais Convolucionais* . . . **31**
2.4.1.10 *Vgg* . . . **31**
2.4.1.11 *Inception* . . . **33**
2.4.1.12 *ResNet* . . . **34**
2.4.1.13 *MobileNet* . . . **34**
2.5 Bibliotecas e Frameworks . . . 35
2.5.1 Tensorflow . . . **35**
2.5.1.1 *Fluxograma Tensorflow* . . . **36**
2.5.2 Pytorch . . . **39**
2.5.3 Caffe . . . **40**
2.6 OpenCL . . . 40
2.6.1 Modelo de Plataforma . . . **41**
2.6.2 Modelo de Execução . . . **41**
2.6.3 Modelo de Memória . . . **43**
2.6.4 Modelo de Programação . . . **44**

3 TRABALHOS RELACIONADOS 45
3.1 Trabalhos que envolvem o uso de GPU 45
3.2 Trabalhos que envolvem o uso de FPGA 46

4 IMPLEMENTAÇÃO PROPOSTA 52
4.1 Análise da Complexidade Computacional 52
4.1.1 Camada de Convolução **52**
4.1.2 Camada de Pooling **53**
4.1.3 Camada Totalmente Conectada **53**
4.2 Metodologia de Projeto 54
4.2.1 Fase de Treino **55**
4.2.2 Parser Tensorflow **55**
4.2.2.1 *Arquivo de Configuração* **56**
4.2.2.2 *Arquivos de Pesos* **57**
4.2.3 Fase de Mapeamento **60**
4.3 Implementação C/C++ Canônica 64
4.4 Implementação OpenCL 64
4.4.1 Construção dos Kernels: **64**
4.4.1.1 *Kernel Convolução* **65**
4.4.1.2 *Kernel de Convolução DepthWise* **65**
4.4.1.3 *Kernel de Polling* **66**
4.4.1.4 *Kernel de FC* **66**
4.4.1.5 *Kernel de BatchNormalization* **67**
4.4.1.6 *Kernel addResidual* **68**
4.4.1.7 *Kernels Ativação* **68**
4.4.1.8 *Kernel de Padding* **69**

5 EXPERIMENTOS E RESULTADOS 71
5.1 Configuração dos Experimentos 71
5.2 Análise e Comparações da Biblioteca Proposta 74
5.2.1 Análise de Tempo **74**
5.2.2 Análise de Potência **77**
5.2.3 Utilização de Recursos **78**
5.2.4 Comparação com outros Frameworks e Bibliotecas **79**

6 CONCLUSÃO 82
6.1 Principais Contribuições 82
6.2 Trabalhos Futuros 82

REFERÊNCIAS 84

1 INTRODUÇÃO

1.1 Contexto Geral

Atualmente as chamadas Redes Neurais Profundas (Deep Neural Networks) têm se mostrado eficazes para problemas com imagens: classificação, reconhecimento de padrões (RONNEBERGER et al., 2015), extração de caracteres (OCR's) (MICROSOFT, 2018) e detecção de objetos (GOOGLE, 2017). Dentre elas estão as CNN's (Convolutional Neural Networks), redes profundas que se caracterizam por sua camada mais importante, a camada de convolução. Em problemas de Visão Computacional, as CNN's se tornaram o estado da arte desde a AlexNet (ALEX KRIZHEVSKY et al., 2012), se popularizando ao vencer o ImageNet Challenge (IMAGENET, 2009): ILSVRC 2012 que tem como objetivo obter a maior taxa de acerto no problema de classificação de imagens com múltiplas classes. Melhorias foram alcançadas em detrimento do aumento da complexidade desses modelos, como o aumento da profundidade das CNN's (KAREN SIMONYAN; ANDREW G. HOWARD, 2014; CHRISTIAN SZEGEDY et al., 2015). Essas redes apresentam alto consumo de memória e processamento gerado pelas sucessivas convoluções ao longo da rede.

A construção de uma rede neural artificial é dividida em 2 fases: fase de treinamento e fase de inferência. A fase de treinamento é normalmente realizada em alguma *cloud* com suporte à GPU's, com um grande conjunto de dados e objetivo de gerar um modelo especialista em aprender padrões desse conjunto de dados. A inferência é a fase de uso de um modelo na qual os exemplos, entrada em que se deseja realizar a inferência, são apresentados ao modelo que responde com as probabilidades das classes para as quais foram treinados. Os exemplos precisam passar por todas as camadas do modelo e é por isso que em arquiteturas profundas a inferência pode se tornar lenta. De acordo com Vivienne (SZE et al., 2017), a fase de inferência é normalmente executada em dispositivos onde os recursos são limitados pelo consumo de energia, pela computação, e limitações de memória pelo seu custo.

Em muitas aplicações reais de áreas estratégicas tais como: robótica e sistemas embarcados, a inferência é executada por frameworks que apresentam limitações de memória, processamento e não dão suporte à vários dispositivos de execução. O Tensorflow (GOOGLE, 2015), Caffe (YANGQING JIA, 2017), e Pytorch (ADAM PASZKE et al., 2016) são exemplos de frameworks e bibliotecas que dão suporte ao treinamento e inferência de modelos de CNN's limitados à CPU's e GPU's Nvidia. Segundo DiCecco (DICECCO et al., 2016), o Caffe apresenta um modo de operação chamado de Brew, que determina o dispositivo de destino na qual a inferência da CNN ou o treinamento são executados. A configuração padrão do Caffe apresenta Brews para CPU's e GPU's Nvidia, assim a dificuldade de permitir que o Caffe realize a inferência em GPU's

que não sejam da empresa Nvidia, ou a inferência em FPGA's, é a de realizar a implementação de Brews para esses dispositivos.

A motivação para desenvolver este trabalho originou-se da necessidade de uma biblioteca que realiza a inferência das arquiteturas de CNN's sem se limitar à um conjunto específico de dispositivos de computação, apresentando flexibilidade com CPU's, GPU's e FPGA's de qualquer fabricante.

1.2 Contexto Específicos

Neste trabalho apresentamos uma biblioteca capaz de realizar a inferência de modelos de CNN's em CPU's, GPU's e FPGA's, independendo da configuração e fabricante do dispositivo, apresentando um parser para mapear arquiteturas de CNN's treinadas em outros frameworks. Sendo assim, a biblioteca independente da especificação que a arquitetura foi treinada.

A fim de solucionar o problema da dependência do dispositivo, é utilizado o framework OpenCL (Open Computing Language) (THE OPENCL, 2009), padrão proposto pela Khronos (THE KHRONOS GROUP, 2000) e aceito por vários fabricantes de hardware (INTEL. . . , 1968; NVIDIA. . . , 1993; AMD. . . , 1969), que tem a característica de ser executado em ambientes computacionais heterogêneos compostos por CPU's, GPU's, FPGA's e outros dispositivos de computação.

Em uma especificação em OpenCL a tarefa de computação intensiva é colocada em um arquivo chamado *kernel,* que será compilado e executado nos dispositivos de computação. O arquivo de *kernel* conterá as implementações das seguintes camadas de CNN's: Convolução, Polling, Relu, Sigmoid, arcoTan, BatchNormalization, Padding, Multiplicação de Matrizes, Convolução DepthWise, AddResidual. Essas camadas serão explicadas na seção 2.4.1. O arquivo chamado *host* carrega o modelo de CNN, inicializa os objetos de memória e gerencia a execução das tarefas.

Sendo assim, o objetivo deste trabalho é propor uma biblioteca em OpenCL que dê suporte às camadas mais utilizadas em modelos CNN's, executando a inferência desses modelos em CPU's, GPU's e FPGA's. Em outras palavras, a biblioteca será capaz de executar a inferência de qualquer arquitetura de CNN suportada pelas camadas implementadas e independerá do fabricante ou do dispositivo que execute a fase de inferência.

1.3 Contribuições

Nesta seção descrevemos as principais contribuições deste trabalho:

- **Implementação de um *parser* de pesos em Tensorflow**

Foi implementado um *parser* de pesos permitindo que arquiteturas treinadas em outros frameworks como Tensorflow, Caffe, Pytorch, possam ser exportadas e carregadas na InferenCNN. O parser recebe como entrada um modelo treinado em um desses frameworks e gera uma especificação dos pesos e configuração das arquiteturas que serão executadas pela biblioteca.

- **Implementação da biblioteca para inferência de CNNs em C/C++**

A primeira versão da biblioteca criada em C e C++ é dita canônica por não apresentar nenhum tipo de paralelismo que não seja o de instruções. Essa versão implementa as camadas utilizadas pelas arquiteturas emergentes de CNN's, faz o carregamento da arquitetura e dos pesos através do *parser,* e executa a inferência do modelo. O principal papel desta versão foi localizar gargalos no processamento das camadas da CNN e pontos de otimizações utilizando um *profiling*, e fazer a validação funcional da biblioteca.

- **Implementação da biblioteca em OpenCL para CPU's/GPU's**

Para dar suporte a GPU's de fabricantes diferentes (AMD. . . , 1969; INTEL. . . , 1968; NVIDIA. . . , 1993) foi implementada uma versão da biblioteca em OpenCL (THE OPENCL, 2009) dando suporte à instanciação da arquitetura e pesos através do *parser,* e execução da inferência do modelo em uma determinada entrada. Esta versão em OpenCL faz uso de *threads* para execução das camadas de CNN e aceleração da inferência.

- **Implementação de uma versão em OpenCL para FPGA's**

Para dar suporte a FPGA's os *kernel's* OpenCL foram sintetizados através do SDK OpenCL da Intel, a fim de processar as camadas e realizar a inferência dos modelos de CNN's em FPGA's.

1.4 Organização da dissertação

O capítulo 2 apresenta os conceitos básicos das redes neurais, as camadas mais utilizadas e tipos de arquiteturas das redes neurais convolucionais, a especificação OpenCL utilizada para o desenvolvimento da InferenCNN, as bibliotecas e frameworks que auxiliam no desenvolvimento das redes neurais.

No capítulo 3 são apresentados os trabalhos relacionados da literatura. Este capítulo está dividido em 2 seções: trabalhos que envolvem o uso de GPU, e trabalhos que envolvem o uso de FPGA.

No capítulo 4 a biblioteca e o fluxo proposto são apresentados, assim como a metodologia de implementação, o parser feito para instanciar os modelos desenvolvidos em frameworks diferentes, e o detalhamento da arquitetura proposta para processar as camadas das redes neurais convolucionais.

No capítulo 5 apresentamos os experimentos e resultados gerados a partir da implementação proposta, a configuração das máquinas, recursos utilizados, tempo de processamento e consumo de energia. E por fim, comparamos com os demais trabalhos com FPGA e frameworks que trabalham com GPU.

No capítulo 6 nós resumimos a importância deste trabalho e os resultados alcançados, bem como melhorias que podem ser implementadas como trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

A introdução do aprendizado de máquina permitiu que os computadores aprendessem, por um processo chamado treinamento, a resolver problemas que não foram explicitamente programados.

No campo do aprendizado de máquina existe uma subárea chamada Redes Neurais com foco na computação inspirada no cérebro, tendo em vista que o cérebro humano é conhecido como a melhor “máquina” para aprender e resolver problemas, termo utilizado por Vivienne Sze (SZE et al., 2017).

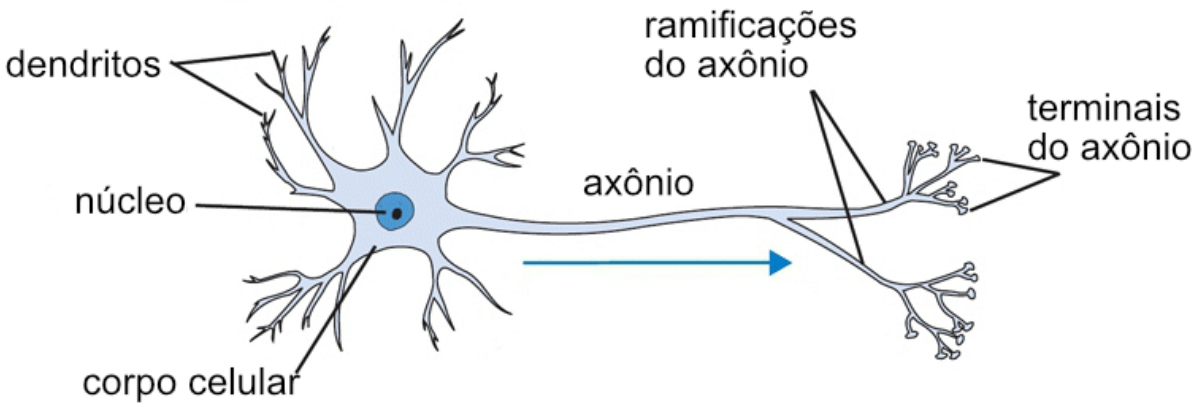
Neste capítulo apresentamos os conceitos básicos das Redes Neurais, assim como os principais marcos que estimularam o crescimento da área.

2.1 Redes Neurais

Muitos cientistas têm explorado o cérebro humano para saber detalhes de seu funcionamento e da capacidade de armazenar e propagar informações, chegando a conclusão que o principal elemento do cérebro é o neurônio. Além disso, os mesmos são células que se formam no nosso cérebro compostas por 3 partes: dendrito, o corpo celular e o axônio, conforme apresentado na Figura 1.

Os neurônios são conectados em conjunto, com os dendritos atuando como captadores de sinais, o corpo celular processando o sinal captado e o axônio distribuindo o sinal para outros neurônios. A conexão entre o axônio de um neurônio com o dendrito de outro neurônio é chamado de sinapse, que é capaz de conduzir os sinais por longas distâncias.

Figura 1 – Neurônio



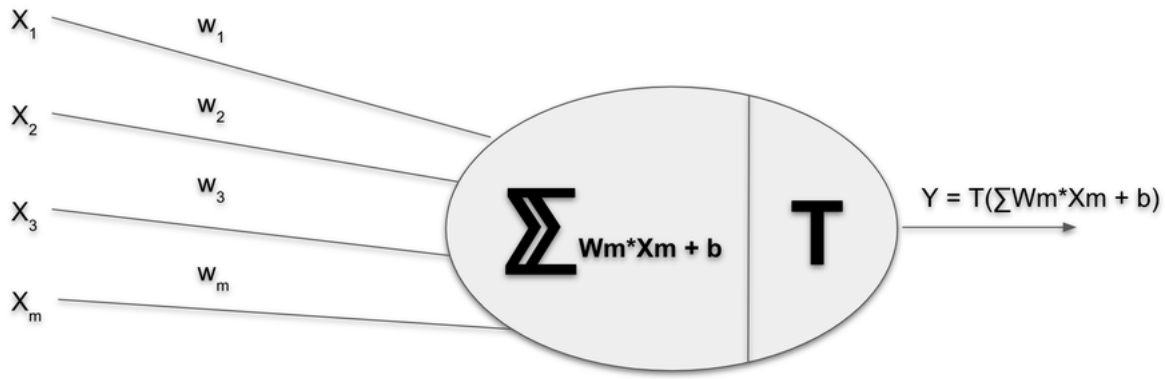
Fonte: CS231N (2018)

As sinapses têm a característica de influenciar outros neurônios de forma excitante ou inibitório através da sua força sináptica, sendo representada por um valor escalar W chamado de peso, assim a sua variação resultará em respostas diferentes. A ideia é que as forças sinápticas sejam aprendidas e controlem a influência em outros neurônios.

A história das redes neurais artificiais começa com a criação do primeiro modelo de neurônio artificial feito por McCulloch (MCCULLOCH; PITTS, 1943), expresso na Figura 2. De acordo com Sze (SZE et al., 2017) as redes neurais se inspiram na noção de que o cálculo de um neurônio envolve uma soma ponderada dos valores de entrada.

A soma ponderada das “m” entradas com os pesos sinápticos (conexão entre as entradas e o neurônio) é submetida à uma função não-linear que define se o neurônio é ativado ou não. Essa função também é conhecida como Função de Ativação do Neurônio, e faz com que o neurônio gere uma saída apenas se a entrada for maior que um limiar. As funções de ativação serão descritas na seção 2.4.1.5.

Figura 2 – Neurônio Artificial

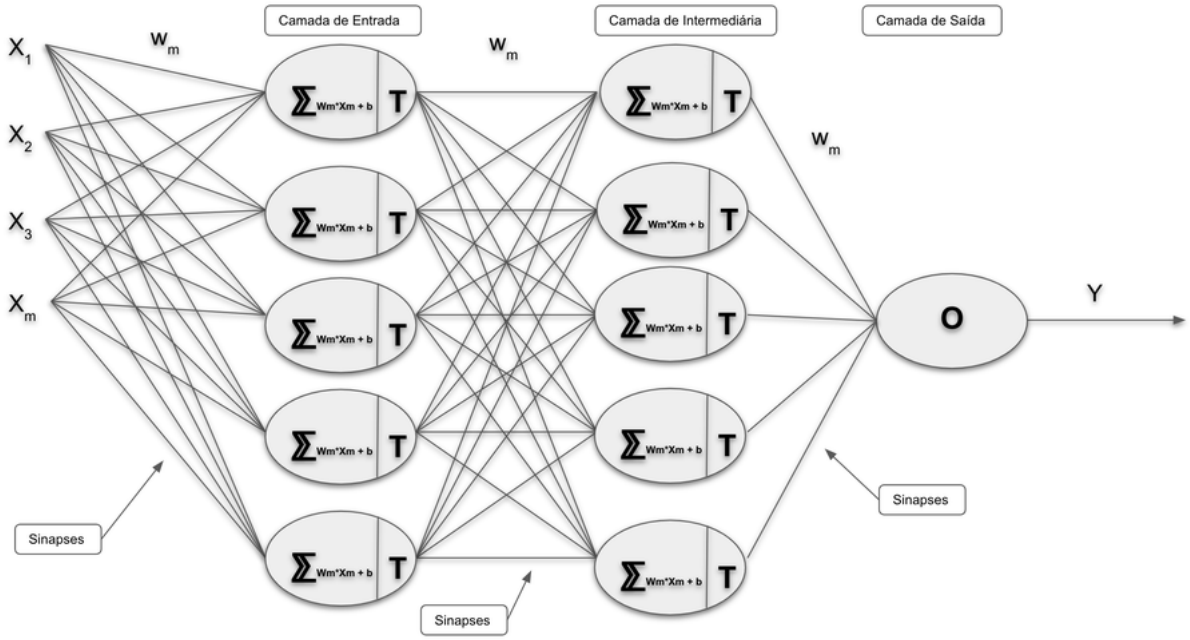


Fonte: O autor (2019)

A Figura 3 mostra a configuração de uma rede neural artificial com o agrupamento de neurônios, em que cada um realiza a soma ponderada das entradas com os pesos sinápticos e posteriormente são somados com o bias (peso com valor de entrada 1 e com objetivo de ajustar o limiar da função de ativação).

Os neurônios da camada de entrada recebem valores e os propaga para os neurônios da camada intermediária ou camada escondida. Posteriormente os sinais são propagados para a camada de saída gerando um sinal Y.

Figura 3 – Multilayer Perceptron (MLP)

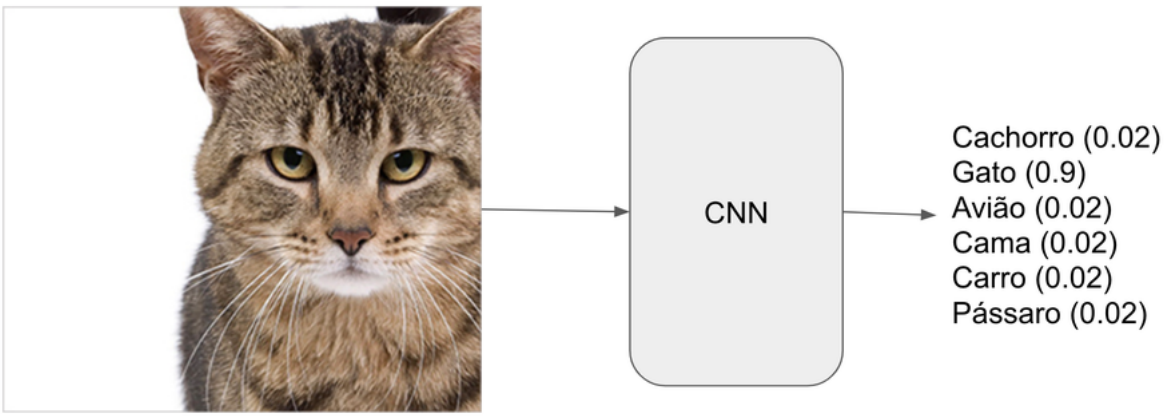


Fonte: O autor (2019)

2.2 Treinamento e Inferência de uma Rede Neural

No problema de classificação de imagens como mostrado na Figura 4, uma imagem é apresentada à uma rede neural que foi treinada para 6 classes. Posteriormente, em um conjunto de saídas a rede expressa as probabilidades para as classes à que foi treinada.

Figura 4 – Rede Neural para Classificação de Imagens



Fonte: O autor (2019)

O objetivo de uma rede neural artificial é encontrar um padrão de pesos que maximizem a probabilidade de saída da classe desejada. Esse processo é chamado de

"treinamento" da rede neural. A Figura 4 apresenta o processo chamado de inferência, em que a rede já se encontra treinada.

No processo de treinamento, a classe da imagem normalmente é apresentada à rede e comparada com as classes de saídas. A diferença entre a classe desejada e as classes previstas pela rede geram um erro E. Assim, o treinamento tem o objetivo de ajustar os pesos a fim de minimizar o erro médio. Esse ajuste acontece através de um processo chamado gradiente descendente que indica como os eles devem ser atualizados.

Os pesos são atualizados através da derivada parcial do erro E em relação ao peso multiplicado pela taxa de aprendizado γ. Essa atualização é descrita pela equação abaixo:

$$w_{ij}^{t+1} = w_{ij}^{t} - \gamma \frac{\partial E}{\partial w_{ij}} \tag{2.1}$$

David Rumelhart, Geoffrey Hinton e Ronald Williams apresentaram em seu trabalho (RUMELHART et al., 1986) um algoritmo chamado backpropagation que calcula eficientemente as derivadas parciais do gradiente descendente. O cálculo é feito de forma eficiente, realizando as derivadas mais rápido que as abordagens da época, e possibilitando o uso de redes neurais para resolver problemas que no momento eram considerados insolúveis (GOODFELLOW et al., 2016) .

2.3 Redes Neurais Profundas

2.3.1 Aprendizagem Profunda

Com o desenvolvimento do backpropagation, tornou-se possível o treinamento das camadas intermediárias de uma rede neural artificial, permitindo adicionar mais camadas entre as camadas de entrada e de saída.

O treinamento com o backpropagation é dividido em duas etapas. Na primeira etapa o dado de entrada é apresentado a rede e propagado camada por camada, até que a saída resultante seja produzida pela última camada. Na segunda etapa a saída resultante é comparada com a saída desejada, gerando assim um erro se a comparação resultar em saídas diferentes. O erro é retropropagado camada por camada até a camada de entrada e os pesos de suas conexões atualizados.

Em uma rede Multilayer Perceptron - MLP treinada com o algoritmo de backpropagation, o processo de treinamento pode demorar algumas horas pela quantidade de passos necessários para se obter o menor erro possível. Quanto mais camadas intermediárias forem adicionadas mais tempo demandará no treinamento, dado que mais pesos precisam ser treinados.

As Redes Neurais Profundas (Deep Neural Network - DNN) geralmente necessitam de uma GPU para acelerar a etapa de treinamento. Essas redes normalmente são empregadas em problemas de maior complexidade (LECUN et al., 1998).

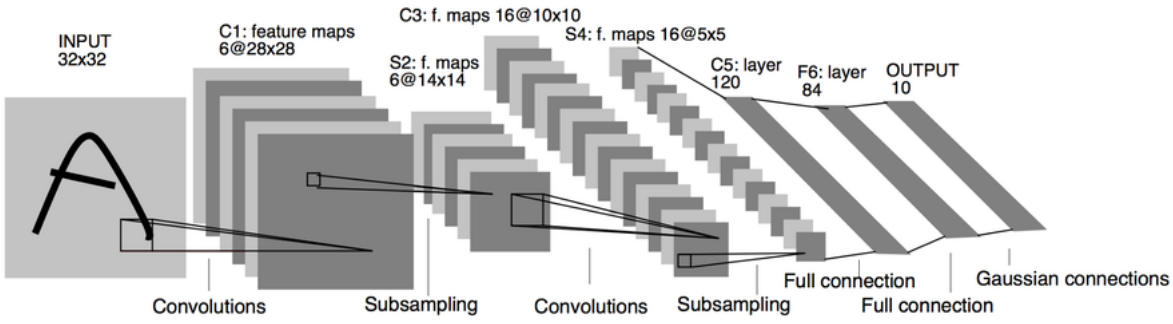
2.3.2 Aplicação Prática das Redes Neurais Profundas

A primeira aplicação prática das redes neurais artificiais apareceu com o trabalho de Lecun (LECUN et al., 1989) apresentando uma MLP de três camadas ocultas, para a época considerada profunda, e que era capaz de realizar o reconhecimento de dígitos manuscritos. A contribuição apresentada levantou interesse da comunidade científica ao reportar bons resultados em um problema real, se tornando na época o estado da arte para reconhecimentos de dígitos manuscritos. Esta foi a primeira aplicação do algoritmo de backpropagation em uma rede neural profunda.

Apesar do sucesso, com a tecnologia da época a MLP apresentada por Lecun possuía um longo tempo de processamento, e demorava cerca de 3 dias para treinar. Um tempo tão grande se tornava impraticável para uso geral.

Mais tarde Lecun apresenta um novo trabalho fazendo comparativo de várias arquiteturas de rede neurais para reconhecimento de dígitos manuscritos (LECUN et al., 1998). Neste trabalho é apresentada a arquitetura LeNet mostrada na Figura 5, uma Rede Neural Convolucional - CNN para reconhecimento de dígitos manuscritos composta por duas camadas convolucionais, duas camadas de subsampling e 3 camadas totalmente conectadas. As redes neurais convolucionais serão explicadas na seção 2.4, detalhando as camadas convolucionais, subsampling e totalmente conectadas.

Figura 5 – LeNet



Fonte: LECUN et al. (1998)

Com a tecnologia da época, processar as redes neurais profundas ainda era uma barreira, pois quanto mais profunda a rede neural for maior será a quantidade de operações, necessitando de um poder computacional maior.

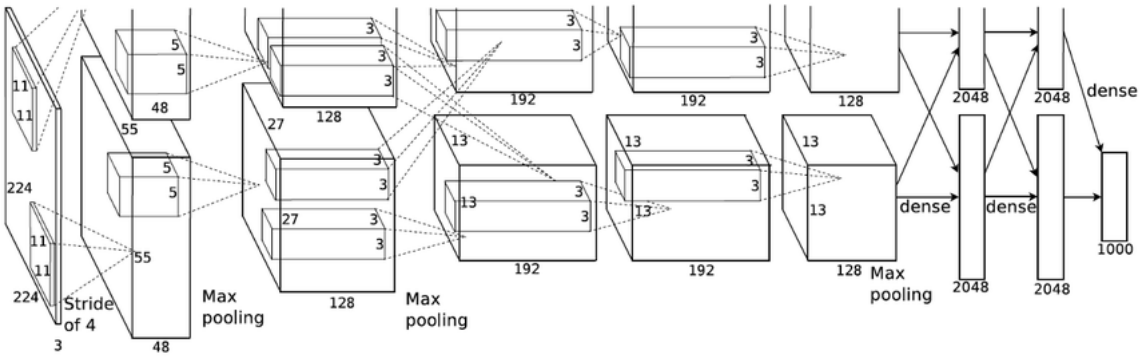
Dois trabalhos foram pioneiros ao utilizar as Unidades de Processamento Gráfico (GPU's) para treinar uma rede neural, os trabalhos de Kyoung-Su Oh (OH; JUNG, 2004)

e Kumar Chellapilla (CHELLAPILLA et al., 2006). Como as GPU's são especializadas para processamento matricial, Chellapilla apresenta uma solução para um rápido processamento das camadas convolucionais transformando a operação de convolução em multiplicação de matrizes. Esta mudança apresentou uma melhora de até 4 vezes no tempo de treino de uma CNN.

Mais adiante Alex Krizhevsky (KRIZHEVSKY et al., 2012) apresenta a arquitetura AlexNet com 5 camadas convolucionais, 3 camadas de MaxPolling e 3 camadas totalmente conectadas. A Figura 6 descreve a AlexNet com a imagem de entrada se dividindo em duas, pois a rede foi treinada com 2 GPU's distribuindo assim a carga de trabalho e reduzindo o tempo de treinamento. A AlexNet é uma Rede Neural Convolucional Profunda (Deep Convolution Neural Network - DCNN), treinada para classificar 1.2 milhões de objetos na competição ImageNet LSVRC-2012 (IMAGENET, 2009), que apresenta 1000 classes diferentes de objetos.

A AlexNet foi vencedora da competição no Top-5 com uma taxa de erro de 15.3% no conjunto de teste, sendo muito superior aos 26.2% do segundo colocado. Este resultado incentivou o aumento nas pesquisas das Redes Neurais Convolucionais Profundas.

Figura 6 – AlexNet



Fonte: ALEX KRIZHEVSKY et al. (2012)

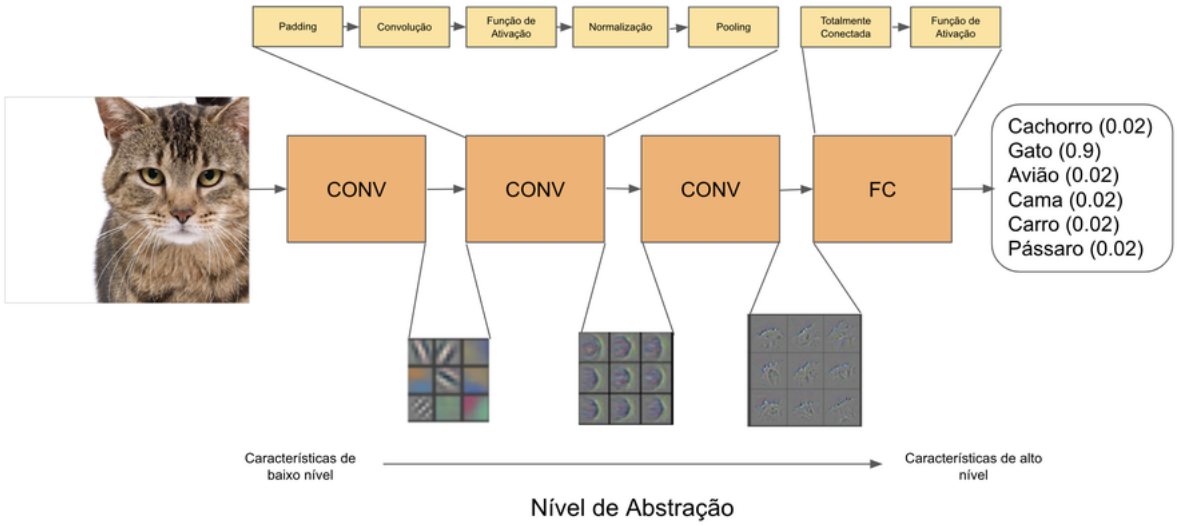
2.4 Redes Neurais Convolucionais

As Redes Neurais Convolucionais - CNN's são redes bioinspiradas no cérebro humano partindo de primitivas básicas como linhas, curvas e formas. Essas redes ganharam o nome de Redes Neurais Convolucionais pela sua camada mais importante, a camada de Convolução, e se tornaram o estado da arte para problemas de visão computacional despertando interesse de grandes empresas como o (GOOGLE, 2017) e (MICROSOFT, 2018).

A Figura 7 apresenta um modelo padrão de uma rede neural convolucional,

composta em sua primeira parte por uma sequência de camadas responsáveis por extrair sucessivos níveis de abstração, conhecido como mapa de característica, construindo conceitos mais abstratos do dado de entrada. A segunda parte é composta por camadas totalmente conectadas (Fully Connected - FC) que são responsáveis por selecionar os atributos que mais se relacionam com a saída desejada.

Figura 7 – Modelo de uma Rede Neural Convolucional



Fonte: O autor (2019)

As CNN's geralmente são utilizadas em problemas em que os dados de entrada são imagens, ou dados em que seja possível aproveitar a sua vizinhança para extrair informações. Essa característica de extrair informações da sua vizinhança vem das camadas convolucionais, em que cada filtro de convolução varre toda a imagem e se especializa em uma determinada característica.

2.4.1 Tipos de Camadas

As Redes Neurais Convolucionais são formadas por uma sequência de camadas por onde os dados serão propagados. Nesta seção são apresentadas as principais camadas utilizadas pelas várias arquiteturas existentes de CNN's.

2.4.1.1 Camada de Convolução

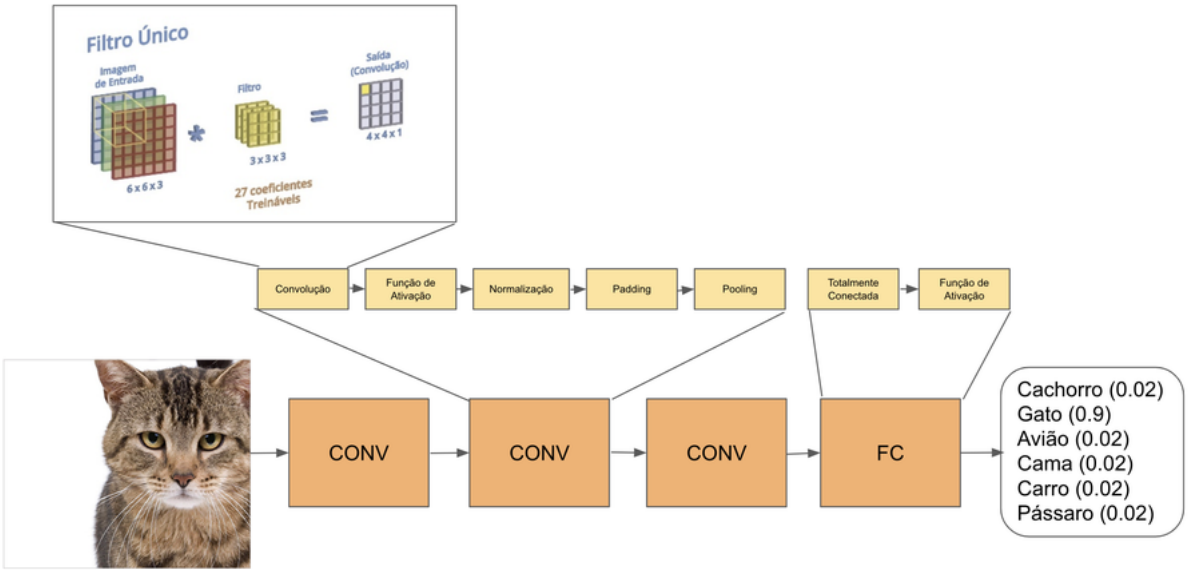
A camada de convolução é responsável pela extração de características dos dados apresentados, aplicando a operação de convolução sobre os dados a partir dos filtros convolucionais. Em uma camada de convolução a operação age em 4 dimensões: largura do filtro (x'), altura do filtro (y'), canais do filtro (c) e quantidade de filtros (f).

$$Fout(f,x,y) = \sum_{c=1}^{C}\sum_{x'=1}^{X}\sum_{y'=1}^{Y} Fin(c, x-x', y-y')W(c,x',y',f) \tag{2.2}$$

A feature de saída (Fout) de dimensões (Rx x Ry) são produzidas pela convolução dos filtros (W), de dimensões (x' x y'), com a feature de entrada (Fin), de dimensões (X x Y).

A Figura 8 apresenta um exemplo com uma imagem de dimensão 6x6 e 3 canais de profundidade (RGB), assim, são necessários filtros com 3 canais de profundidade em que cada canal do filtro deve convoluir com os canais da imagem e combinar-se para formar o mapa de features resultante 4x4x1. Na Figura 8 apenas um filtro é utilizado, e por isso a quantidade de mapas de features resultante é 1.

Figura 8 – Camada de Convolução



Fonte: O autor (2019)

2.4.1.2 Camada de Convolução Depthwise

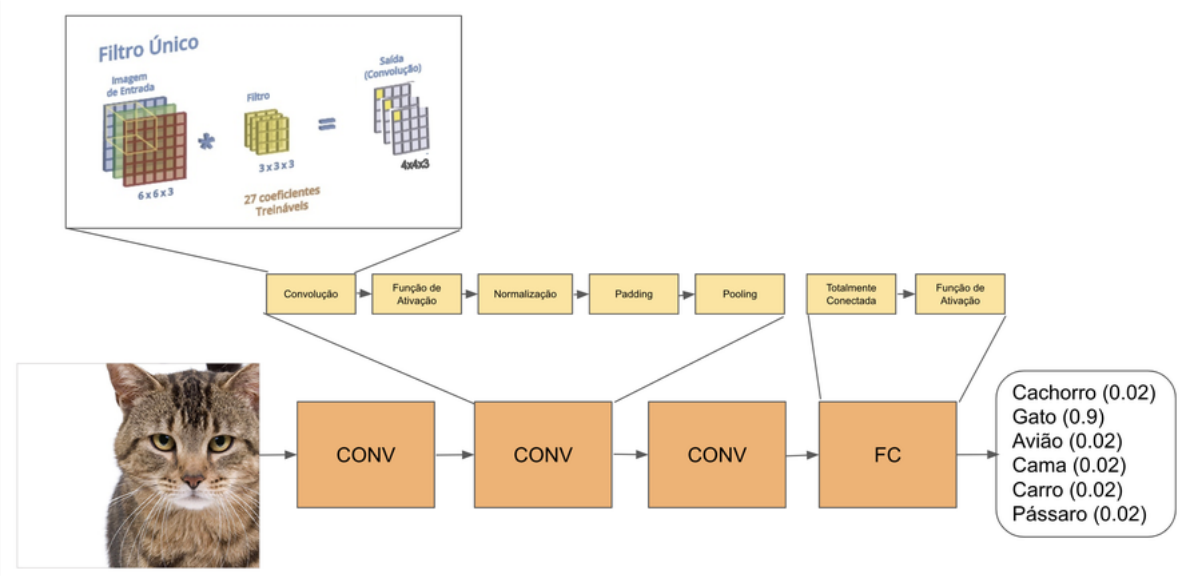
A camada de convolução depthwise é mostrada na Figura 9 e é uma camada similar a camada de convolução padrão, diferenciando pela quantidade de mapas de features geradas. A camada de convolução depthwise também age sobre 4 dimensões: largura do filtro (x'), altura do filtro (y'), canais do filtro (c) e quantidade de filtros (f).

$$Fout(f*c,x,y) = \sum_{x'=1}^{X}\sum_{y'=1}^{Y} Fin(c, x-x', y-y')W(c,x',y',f) \tag{2.3}$$

A feature de saída (Fout) de dimensões (Rx x Ry) é produzida pela convolução dos filtros (W), de dimensões (x' x y'), com a feature de entrada (Fin), de dimensões (X x Y).

Diferentemente da convolução padrão que aplica um filtro sobre os canais da imagem e os combina para gerar o mapa de features resultante, a convolução depthwise utiliza um filtro para cada canal da imagem gerando mais mapas de features. A convolução depthwise apenas filtra os canais de entrada, mas não os combina para criar novos recursos (ANDREW G. HOWARD et al., 2017).

Figura 9 – Camada de Convolução Depthwise

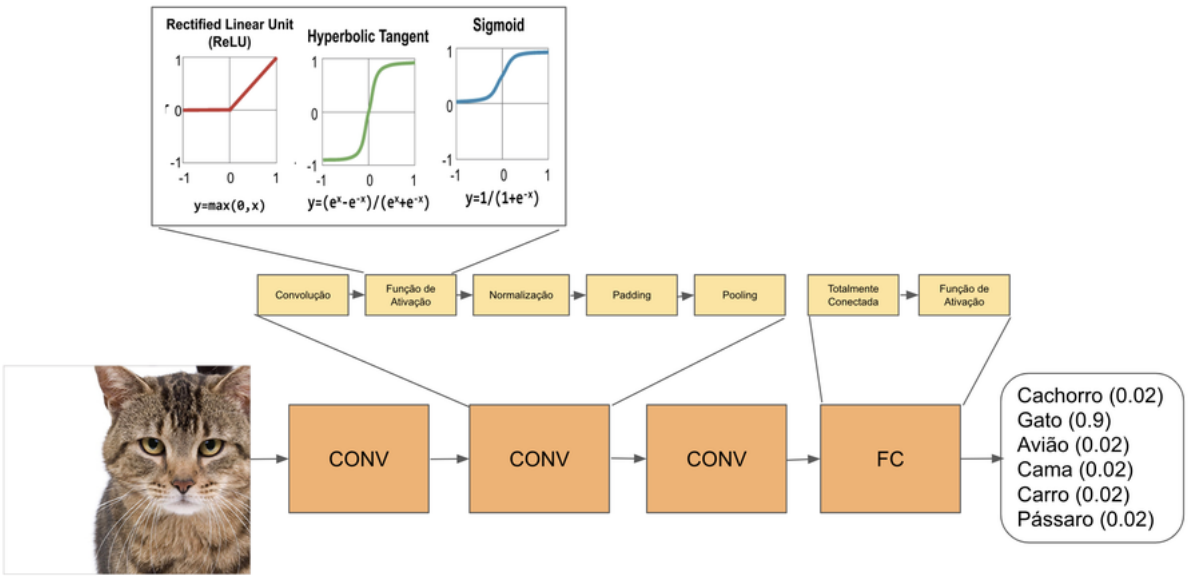


Fonte: O autor (2019)

2.4.1.3 Camada de Ativação

As camadas de ativação normalmente são utilizadas após as camadas convolucionais, aplicando uma função não-linear em todos os pixels de saída. As camadas de ativações mais utilizadas são: ReLU, Tangente Hiperbólica e Sigmoide que são mostradas na Figura 10a, Figura 10b, e Figura 10c respectivamente. A ReLU tem apresentado melhores resultados em CNN's, resultando em um treinamento mais rápido e com desempenho parecido ou superior às outras camadas de ativação.

Figura 10 – Funções de Ativação

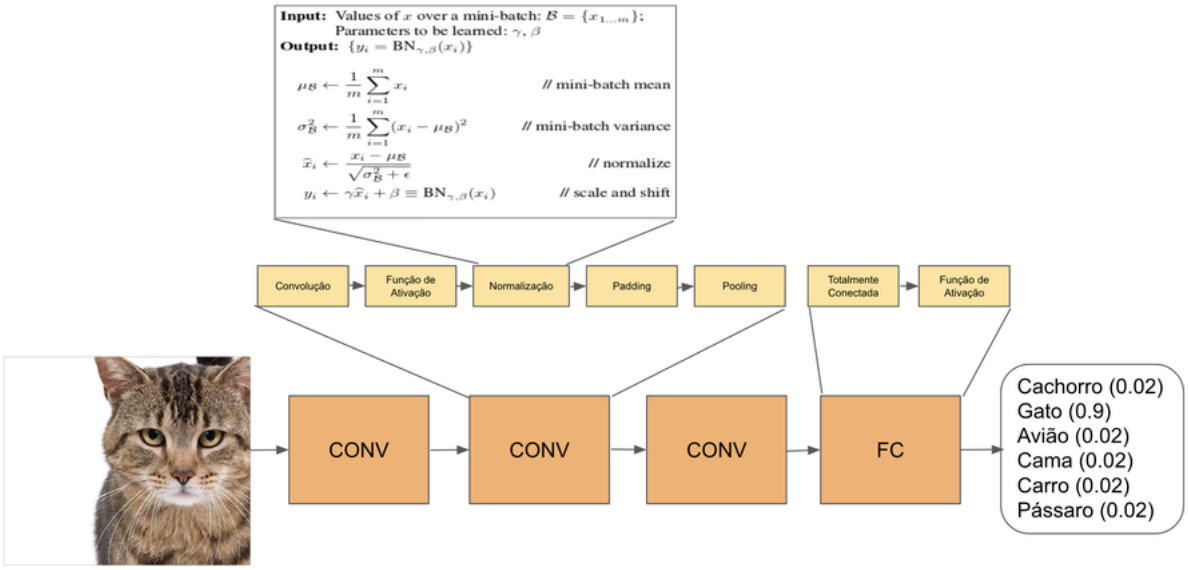


Fonte: O autor (2019)

2.4.1.4 Camada de Batch Normalization

As camadas de Batch Normalization são utilizadas para acelerar o treinamento em casos que os dados apresentam escalas muito diferentes. Um exemplo básico é quando temos dados entre 0 e 1 e outros dados entre 0 e 10000, apresentando uma distribuição muito variável entre cada camada e assim dificultando o aprendizado dos pesos. Normalizar faz com que os dados sejam padronizados e o aprendizado seja acelerado. A Figura 11 apresenta o algoritmo para o Batch Normalization em que cada dado do Batch é normalizado subtraindo a média do Batch e dividindo pelo desvio padrão do Batch.

Figura 11 – Batch Normalization



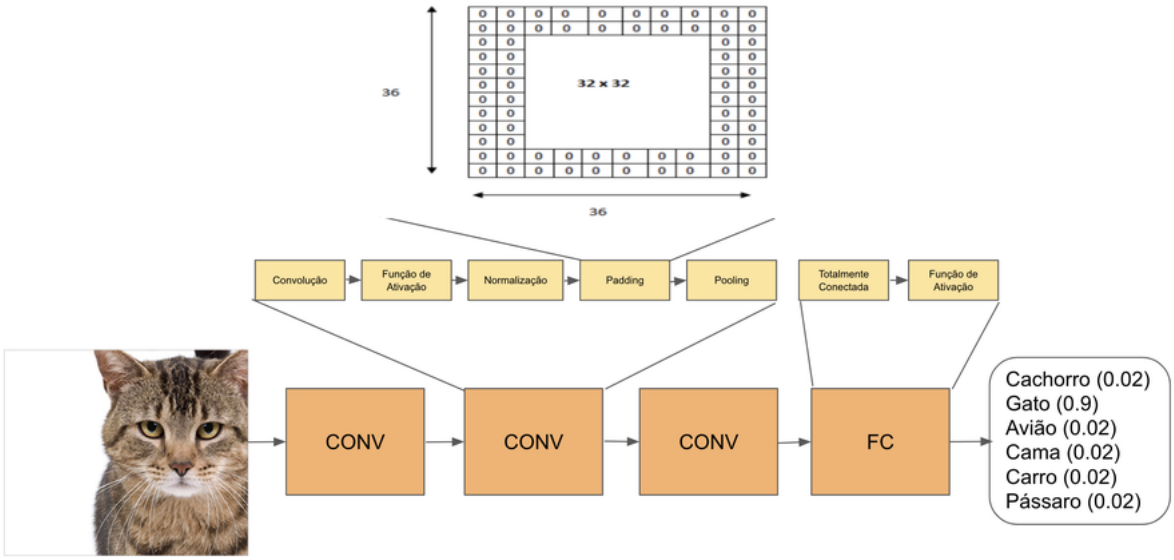
Fonte: O autor (2019)

2.4.1.5 Camada de Preenchimento (Padding)

A camada de preenchimento ou padding normalmente é utilizada nas camadas convolucionais, sendo uma das formas de lidar com as bordas no deslizamento dos filtros convolucionais, como mostrado Figura 12. Essa camada poder ser utilizada com 2 técnicas, valid padding e same padding.

Ao se deslizar o filtro sobre a imagem de entrada o tamanho original da imagem é degradado pela operação de convolução. Esse efeito é chamado de valid padding, onde apenas os dados válidos da imagem serão utilizados para a convolução e as bordas do filtro não ultrapassam a imagem. O same padding é utilizado quando queremos manter o tamanho original da imagem, então a imagem original é preenchida com valores ’0’ em suas bordas e a imagem resultante apresenta mesma dimensão que a imagem de entrada.

Figura 12 – Camada de Padding



Fonte: O autor (2019)

A quantidade de preenchimento é definido pela fórmula:

$$P = \frac{K - 1}{2} \tag{2.4}$$

Onde K é o tamanho o filtro de convolução.

O tamanho da imagem de saída é determinado pela fórmula:

$$O = \frac{(W - K - 2P)}{S} + 1 \tag{2.5}$$

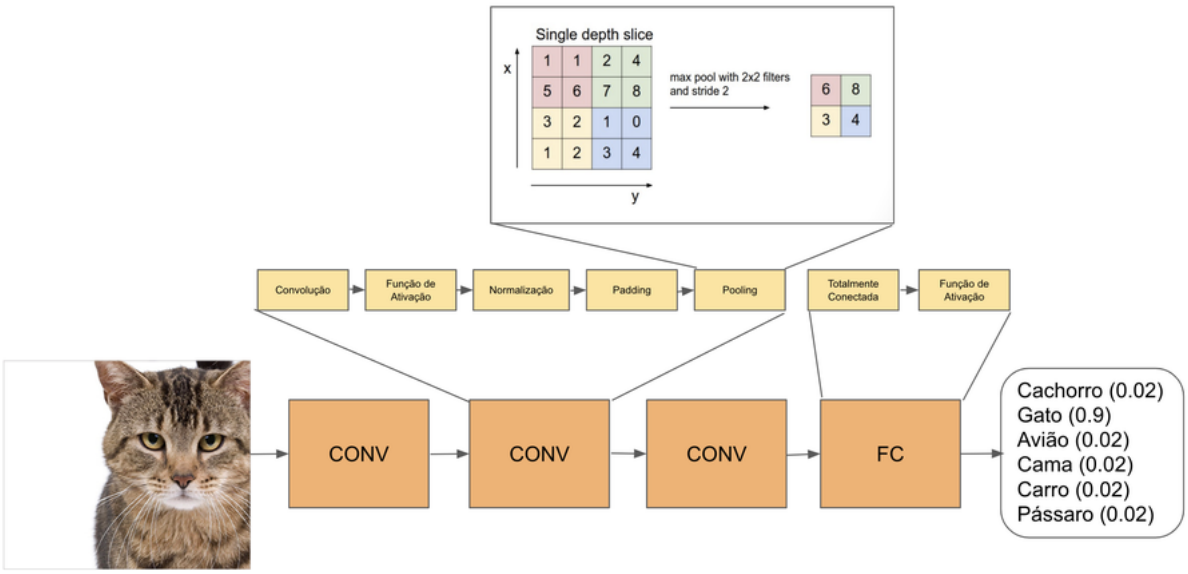
Onde W é a largura da imagem e S é o passo em pixels (stride) que o kernel de convolução desliza.

2.4.1.6 Camada de Subamostragem (Pooling)

A camada de subamostragem ou pooling é utilizada para compactar a informação, reduzir a quantidade de pesos utilizados, e assim reduzir o custo computacional necessário para treinar a rede. É um filtro que extrai apenas as informações consideradas úteis em uma região, focando nas features mais significativas da camada anterior. Ela simplifica a informação resumindo a região em apenas uma informação.

A operação de pooling pode ser feita pela função de mínimo, média, ou o máximo da região. A função de máximo na camada de polling, ou maxpooling, é a função mais utilizada e filtra a região pelo seu valor máximo, como mostrado na Figura 13

Figura 13 – Camada de Pooling

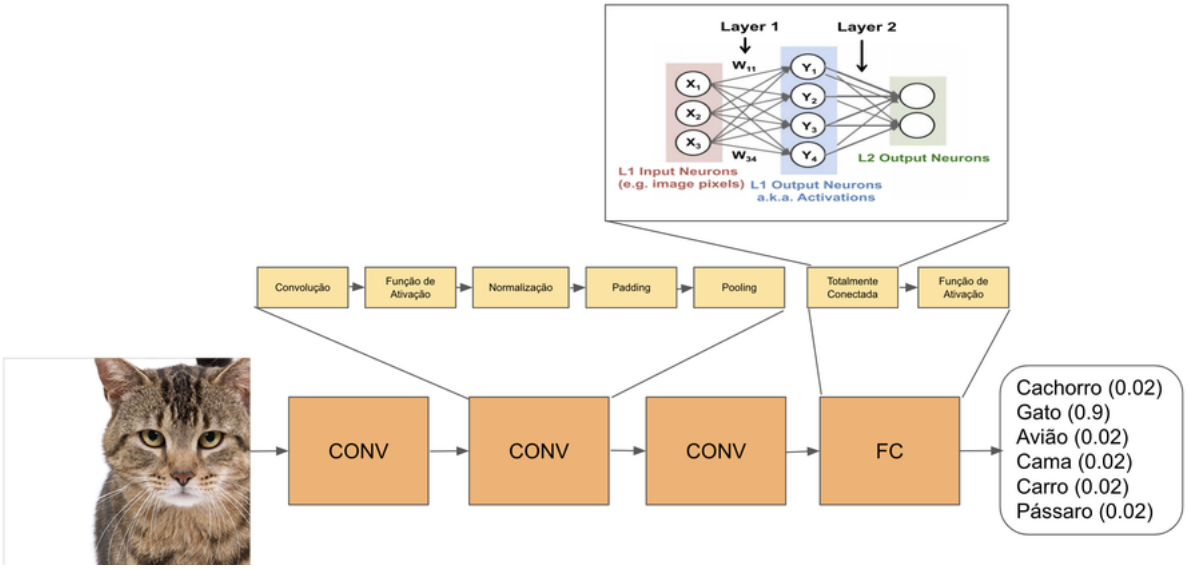


Fonte: O autor (2019)

2.4.1.7 Camada Densa

As camadas densas são camadas totalmente conectadas que normalmente são utilizadas no final das redes convolucionais. Essas camadas tem o objetivo de classificar as features extraídas das camadas convolucionais, como mostrado na Figura 14.

Figura 14 – Camada Densa



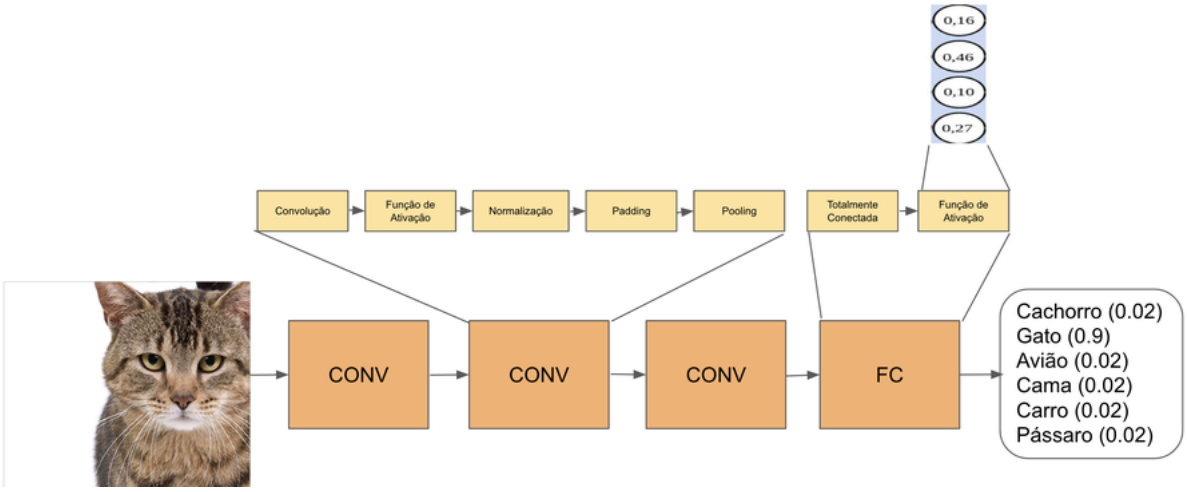
Fonte: O autor (2019)

2.4.1.8 Camada Softmax

A camada de softmax é a última camada densa a ser utilizada na rede, como mostrado na Figura 15. Ela executa a função de ativação softmax abaixo para predizer a classe do dado de entrada, transformando as saídas para cada classe em valores entre 0 e 1, dividindo o resultado pela soma das saídas e dando a probabilidade de a entrada pertencer à uma determinada classe.

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{k} e^{z_k}} \longleftarrow for: j = 1, ..., K \tag{2.6}$$

Figura 15 – Camada de Softmax



Fonte: O autor (2019)

2.4.1.9 Arquiteturas de Redes Neurais Convolucionais

As CNN's possuem a característica de apresentarem um alto consumo computacional e necessidade de armazenamento por suas sucessivas convoluções, e por isso, desde a Lenet (LECUN et al., 1998) e AlexNet (KRIZHEVSKY et al., 2012), muitas arquiteturas de CNN's foram propostas buscando otimizações como maior acurácia, menor tempo de processamento e menos necessidade de armazenamento.

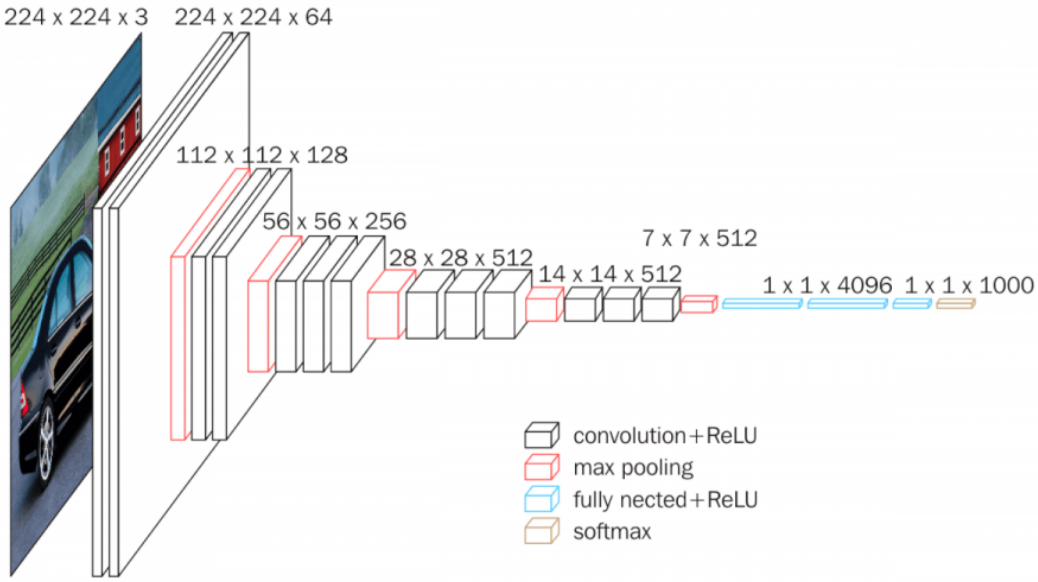
Algumas das principais arquiteturas de CNN's são descritas abaixo, cada uma com suas características intrínsecas.

2.4.1.10 Vgg

Karen Simonyan apresentou em seu trabalho (SIMONYAN; ZISSERMAN, 2014) a VGG que pode ser vista na Figura 16. A VGG é uma rede neural convolucional

com variações de quantidade de camadas, podendo ter uma profundidade de 16 a 19 camadas, com 3 camadas densas no final da rede.

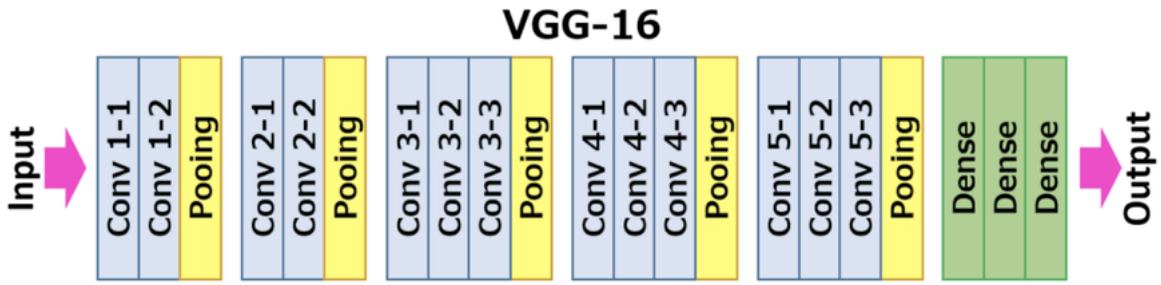
Figura 16 – VGG



Fonte: SAXENA (2017)

A Figura 16 apresenta a arquitetura da VGG16, tendo como entrada uma imagem de tamanho fixo 224x224 RGB. A imagem é passada por uma pilha de filtros convolucionais de tamanho 3x3 e algumas camadas de pooling como mostrado na Figura 17.

Figura 17 – Pinha de Camadas VGG16



Fonte: NEUROHIVE (2018)

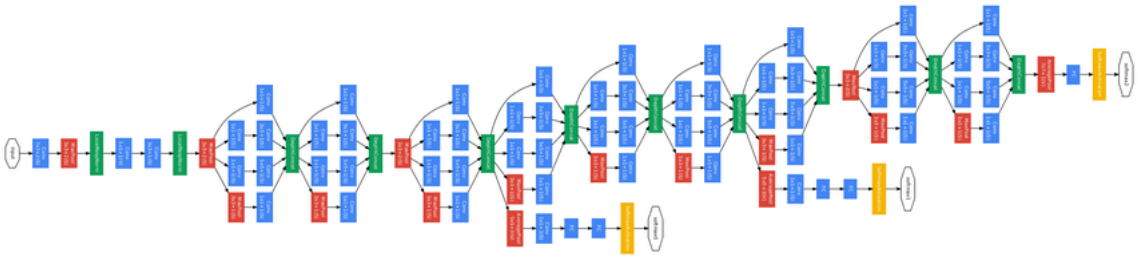
A principal característica da VGG é explorar filtros com dimensões 3x3 com pequenos campos receptivos para uma arquitetura profunda, e mostrar que filtros 7x7, que eram utilizados na Alexnet, podem ser substituídos por uma sequência de 3 filtros 3x3, reduzindo assim a quantidade de parâmetros. A VGG foi a campeã da competição ILSVRC 2014 em localização e conseguiu o segundo lugar em classificação.

2.4.1.11 Inception

O modelo proposto por Szegedy (CHRISTIAN SZEGEDY et al., 2015) pode ser visto na Figura 18. A Inception é uma rede neural convolucional que visa diminuir a quantidade de parâmetros (23 milhões comparado com a AlexNet de 60 milhões) com o bloco Inception, sem perder a eficiência.

Suas principais características foram sair do padrão de redes neurais sequenciais apresentando bifurcações dentro da rede, e o uso de mais de um classificador que aceleram o treinamento e podem ser utilizados para aprender características de visões diferentes.

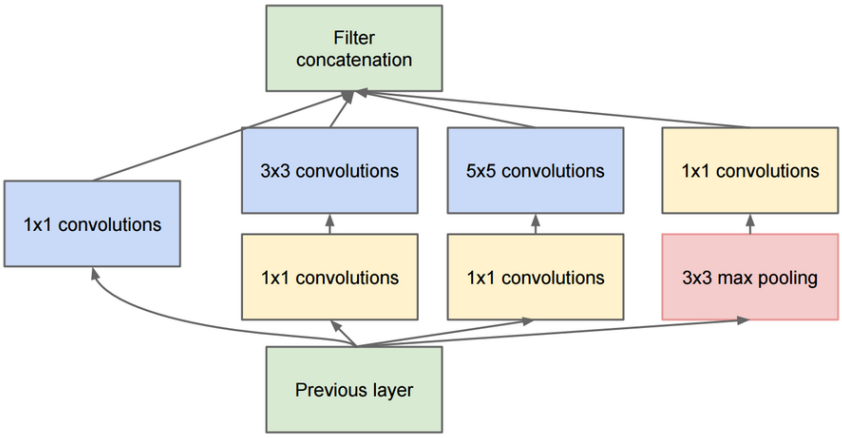
Figura 18 – Inception



Fonte: SZEGEDY et al. (2014)

Diferente da VGG16, que é uma pilha de camadas, a rede Inception é composta por 9 blocos denominados “blocos inceptions” que propaga a informação de uma camada em 4 fluxos de dados. A Figura 19 apresenta, de baixo pra cima, uma camada sendo propagada em 4 fluxos de dados, na qual cada fluxo é responsável por extrair informações em níveis de abstração diferente. No final, o resultado dos 4 fluxos são concatenados e propagados para o próximo bloco inception.

Figura 19 – Bloco Inception



Fonte: SZEGEDY et al. (2014)

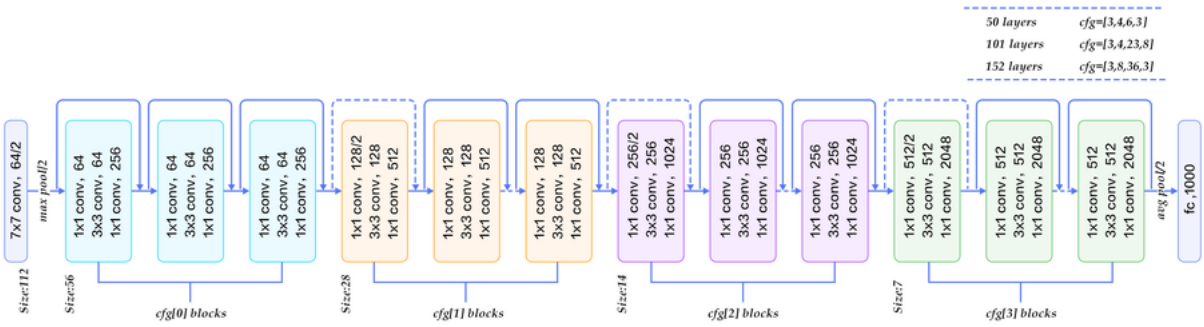
A rede Inception foi a campeã da competição ILSVRC em 2014, reduzindo significativamente a quantidade de parâmetros e demonstrando que resultados de alta qualidade podem ser alcançados com imagens tão pequenas de tamanho 79 × 79.

2.4.1.12 ResNet

Kaiming He et. al apresentaram em seu trabalho a ResNet (HE et al., 2016), com arquitetura apresentada na Figura 20. A ResNet é uma rede neural convolucional com mais de 150 camadas, mostrando uma nova estrutura de aprendizagem para redes neurais profundas ao reformularem as camadas com funções residuais como referência às entradas das camadas. Outra característica importante da ResNet é que ela apresenta um uso intenso das camadas BatchNormalization e não possui camadas densas a não ser a camada de saída.

A Figura 20 apresenta a arquitetura da ResNet da esquerda pra direita. A imagem de entrada é submetida à uma camada de convolução de filtro 7x7, logo após segue uma sequência de blocos residuais somando o dado residual a cada 3 camada de convolução. A ResNet foi a campeã da competição ILSVRC em 2015.

Figura 20 – ResNet



Fonte: UTRERA (2018)

2.4.1.13 MobileNet

Andrew G. Howard et. al apresentaram uma classe eficiente de redes neurais convolucionais para aplicativos embarcados, que usa convoluções separáveis para construir uma arquitetura leve (ANDREW G. HOWARD et al., 2017). A MobileNet é baseada em convoluções depthwise separáveis, que é uma forma de convolução fatorada em uma convolução depthwise e uma convolução pontual 1x1 para combinar as saídas da convolução depthwise, como mostrado na Figura 21 .

Essas redes apresentaram uma grande redução dos parâmetros, de 23.2 milhões de parâmetros (Inception) para 3.3 milhões de parâmetros (MobileNet), apre-

sentando uma melhora na acurácia de 69.8% (Inception) para 70.6% (MobileNet) no dataset da Imagenet.

Figura 21 – MobileNet

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Fonte: ANDREW G. HOWARD et al. (2017)

2.5 Bibliotecas e Frameworks

Nesta seção são apresentados os frameworks mais utilizados pra a construção das arquiteturas de CNN's, apresentando conceitos, vantagens e limitações diferentes. Uma das limitações comuns a todos os frameworks que podemos citar é o uso de apenas GPU's Nvidias para treinamento e inferência dos modelos.

Abaixo descrevemos os conceitos e características do Tensorflow, Pytorch e Caffe.

2.5.1 Tensorflow

O TensorFlow é uma biblioteca de código aberto para computação numérica que usa grafos de fluxo de dados (GOOGLE, 2015). Desenvolvido pelos pesquisadores do grupo Google Brain Team com a finalidade de realizar pesquisas sobre redes neurais profundas e aprendizado de máquina. O TensorFlow é um dos frameworks

mais utilizados no mundo por pesquisadores e grandes empresas como: Google, Intel, Qualcomm, ARM, Accenture, e SAP.

Qualquer tipo de dados no Tensorflow é representado por tensores, que são arrays ou listas com N dimensões. Uma matriz em Tensorflow é representada por um tensor de 2 dimensões, um vetor é representado por um tensor de 1 dimensão, e um escalar é representado como um tensor sem dimensão.

2.5.1.1 Fluxograma Tensorflow

Um programa TensorFlow é dividido em duas etapas: fase de construção do grafo, e fase de execução do grafo.

Um grafo pode ser entendido como uma sequência de operações como: subtração, multiplicação e adição. Essas operações são chamadas de “nós” e são instanciadas em uma estrutura chamada *tf.Graph(),* que é a representação do grafo. A Figura 22 apresenta um exemplo de um código em tensorflow que carrega constantes no tensor ’b’. Note que todas as operações devem ser realizadas dentro de um grafo.

Figura 22 – Código Tensorflow

```
graph = tf.Graph()
with graph.as_default():
    # criamos constante com valor 8
    b = tf.constant(8, name='b')
    print(b)

    # criamos constante com valor 0
    b = tf.constant(0, name='b')
    print(b)
```

Fonte: O autor (2019)

Para executar o grafo e mostrar os valores que foram armazenados no tensor ’b’, é preciso criar uma estrutura chamada *Session* utilizada para acessar os valores dos tensores.=- A Figura 23 mostra a execução de uma soma e multiplicação de 2 tensores.

Figura 23 – Soma e Multiplicação de Tensores

```
graph = tf.Graph()
with graph.as_default():

    x_1 = tf.constant(2.0, dtype=tf.float32)
    x_2 = tf.constant(5.0, dtype=tf.float32)

    mult = x_1 * x_2
    soma = x_1 + x_2

    sess = tf.Session()

    # Aqui executamos a multiplicação
    print(sess.run(mult))

    # Aqui executamos a soma.
    print(sess.run(soma))
```

Fonte: O autor (2019)

Com os grafos de fluxo o tensorflow possui algumas vantagens entre as quais descata-se:

- Paralelismo: Permite a execução de operações em paralelo e facilmente detecta dependências entre as operações.
- Execução distribuída: É possível particionar um programa em vários dispositivos (CPU's, GPU's e TPU's - Unidade de Processamento de Tensores).
- Compilação: O compilador do TensorFlow pode usar as informações do grafo de fluxo de dados para gerar um código mais otimizado.
- Portabilidade: O grafo de fluxo independe da linguagem, sendo possível criar um grafo em python, salvar, e restaurar o grafo em um programa em C++.

O TensorFlow foi o framework utilizado para comparações das implementações feitas neste trabalho, juntamente com o Keras (KERAS. . . , 2015), uma API de redes neurais de alto nível que executa sob o framework TensorFlow, escrita em Python e desenvolvida para suportar a experimentação rápida. Com o Keras a criação de modelos TensorFlow se torna mais rápida e fácil para programadores de linguagens de alto nível, acelerando o desenvolvimento, e a realização de experimentos em CPU's

e GPU's. Com o Keras o grafo e a session são abstraídos, dando ênfase apenas em construir e executar o modelo. A Figura 24 mostra a construção de um modelo em Keras para classificação de dígitos manuscritos. A estrutura *Sequential* define o grafo como uma sequência de camadas, onde cada camada é adicionada ao modelo.

O primeiro passo é fazer a leitura das imagens do dataset. O Keras contém alguns dataset como: Mnist, Cifar10 e Cifar100 já definidos e distribuídos em imagens que serão utilizadas para treinamentos e imagens para teste. X_train e Y_train são vetores de imagens e labels, respectivamente, que serão utilizadas para treinamento. X_test e Y_teste são vetores de imagens e labels, respectivamente, que serão utilizadas para teste. A camada Flatten é utilizada como entrada para estirar cada imagem em forma de vetor, que se conectará com a camada totalmente conectada com 512 neurônios e ativação ReLU. A camada de saída é uma camada densa com ativação Softmax, que dará as probabilidades de uma imagem pertencer à uma das 10 classes do Mnist. Assim, o grafo é construído sem a preocupação de uma definição explícita.

O segundo passo é a execução do modelo, em que o modelo é compilado configurando detalhes do treinamento como: o tipo de otimizador (optmizer), a função objetivo que deverá ser otimizada durante o treinamento (loss) e as métricas que o modelo deverá medir (metrics). A função *model.fit* inicia o treinamento esperando como argumentos o número de épocas que o treinamento executará, e as imagens e labels que serão utilizadas no treinamento.

Figura 24 – AlexNet em Keras

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist

(x_train, y_train),(x_test, y_test) = mnist.load_data()

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(512, activation=tf.nn.relu),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)

model.predict(x_test)
```

Fonte: O autor (2019)

2.5.2 Pytorch

O Pytorch é um pacote de *machine learning* em Python baseado no torch que é um pacote de *machine learning* implementado em Lua (ADAM PASZKE et al., 2016). Diferente do TensorFlow a construção dos grafos em Pytorch é feita de forma dinâmica, fazendo com que a construção de redes neurais complexas seja fácil e intuitiva.

O Pytorch apresenta duas características principais:

- Computação de tensor: O tensor Pytorch é muito parecido com o *numpy array* de python (um array n-dimensional), o que o torna simples e fácil de ser utilizado em um programa Python. Uma importante característica, comparado com o numpy é que a computação desses tensores pode ser processada por GPU's, transferindo o tensor de CPU para GPU com CUDA (CUDA. . . , 2006) e acelerando os cálculos do tensor.
- Diferenciação automática: É a avaliação da derivada de um tensor, calculando os retrocessos da Rede Neural. A Diferenciação Automática tem como principal função a construção e treinamento de redes neurais, necessitando de mudanças mínimas no código existente, sendo necessário apenas declarar o

Tensor para o qual os gradientes devem ser calculados com a palavra-chave *requires_grad=True*.

2.5.3 Caffe

Caffe é um framework para Redes Neurais Profundas desenvolvido pelo Centro de Visão e Aprendizagem de Berkeley (BVLC), em linguagem C++ e com algumas implementações em Python e Matlab (YANGQING JIA, 2017).

O Caffe foi desenvolvido para apresentar as seguintes características:

- Expressão: Os modelos e as otimizações são definidos como esquemas de texto simples, ao invés de código dependente de alguma linguagem de programação.
- Desempenho: Visando desenvolvimento para projetos de pesquisa e indústria, o Caffe apresenta desempenho quando submetido à grandes massas de dados para realizar o treinamento e a inferência, aproveitando assim os aspectos naturais da programação em C++.
- Modularidade: O Caffe foi projetado para ser o mais modular possível, sendo possível a inclusão de novos formatos de entrada, camadas e funções de perda. Cada módulo Caffe é submetido a uma sequência de testes, e nenhum novo código é aceito sem que passe pelos testes correspondentes.
- Abertura: Caffe fornece uma licença para uso acadêmico não-comercial e modelos de referência como classificação com AlexNet e detecção com R-CNN.
- Comunidade: Caffe foi projetado para pesquisas acadêmicas, protótipos de startup e desenvolvimento industrial que compartilham experiências, discussões e desenvolvimento em conjunto.

2.6 OpenCL

O OpenCL (Open Computing Language) é um padrão para computação paralela em arquiteturas heterogêneas, gerando código executável em dispositivos de computação como CPU's, GPU's, FPGA's e DSP's (THE OPENCL, 2009). O OpenCL foi lançado em 2008 e até então é gerido pelo grupo Khronos, sendo aceito por grandes empresas como Intel, AMD, Apple e Nvidia.

Diferente da especificação CUDA (CUDA. . . , 2006) a especificação OpenCL oferece portabilidade para diversos dispositivos, onde seus kernels OpenCL podem ser executados em dispositivos de arquiteturas e configurações diferentes.

OpenCL pode ser visto em quatro ângulos: modelo de plataforma, modelo de execução, modelo de memória e modelo de programação. Nesta seção é apresentado o OpenCL e seus 4 modelos.

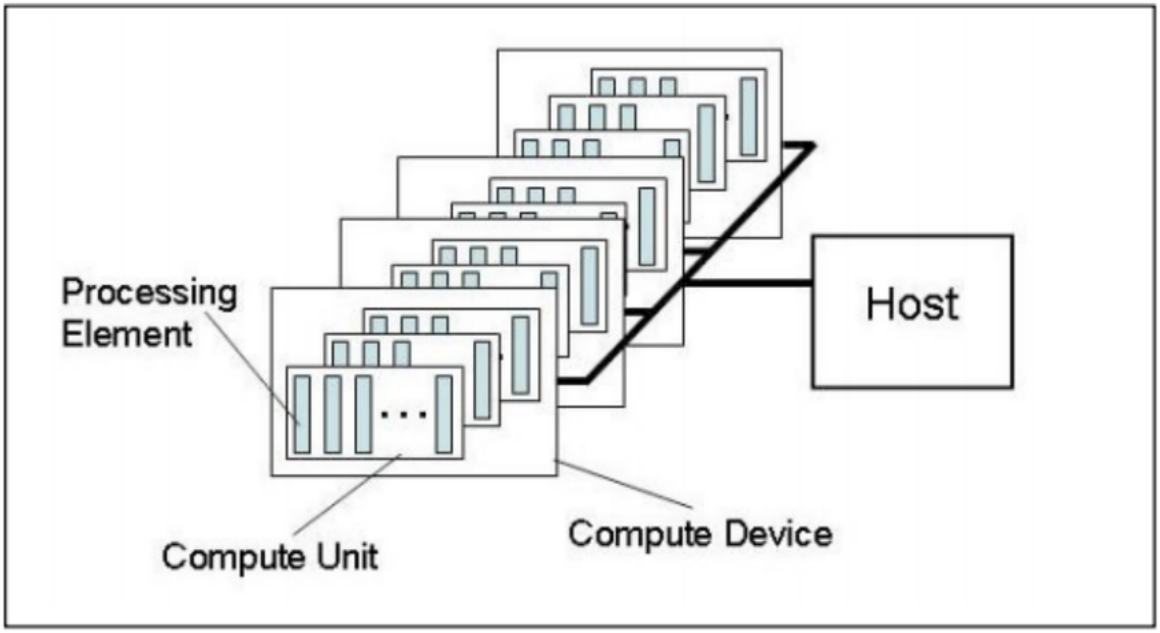
2.6.1 Modelo de Plataforma

O modelo de plataforma descreve as entidades presentes como mostrado na Figura 25: *Host*, *Compute Unit*, *Compute Device* e o *Processing Element*.

O host é responsável por gerenciar toda a alocação, inicialização, transferência de dados e execução dos trabalhos realizados pelos dispositivos (*compute devices*). Cada dispositivo possui um ou mais unidades de computação (*compute unit*), que por sua vez possuem um ou mais elementos de processamento (*processing element*).

Os dispositivos são equivalentes a várias CPU's e GPU's, as unidades de computação são equivalentes aos vários processadores de um dispositivo, e os elementos de processamento são equivalentes aos núcleos desses processadores.

Figura 25 – Modelo de Plataforma OpenCL



Fonte: THE OPENCL (2009)

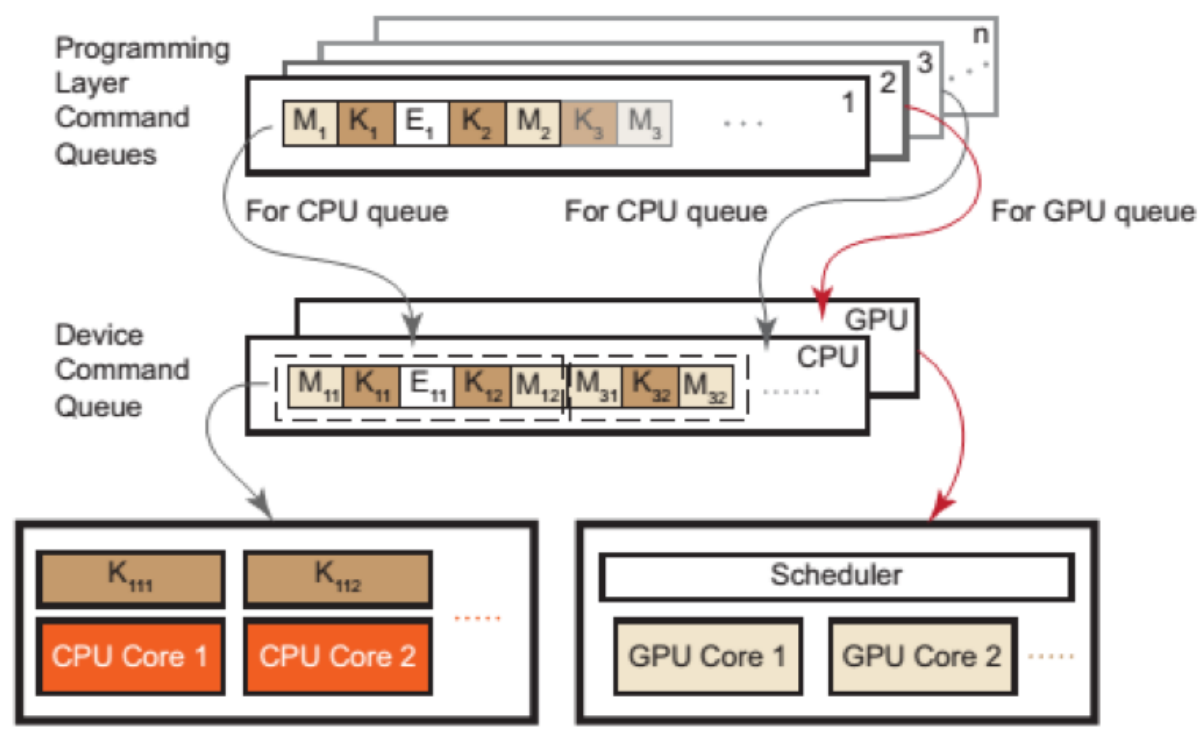
2.6.2 Modelo de Execução

O modelo de execução do OpenCL é dividido em duas partes: o código host e os kernels. O host é responsável por executar partes sequenciais do código e inicializar os objetos de memória, filas e instâncias de kernels OpenCL. Os kernels são funções paralelas que são executadas nos dispositivos. Através da API OpenCL o host constrói

uma estrutura chamada “contexto” que carrega todas as informações necessárias do dispositivo como: tipo do dispositivo, memória disponível, quantidade de unidades de computação entre outras, e gerencia as execuções dos kernels. Através do Contexto são criadas Filas de Comando que são utilizadas para interagir com os dispositivos na transferência de memória, lançamento dos Kernels e sincronização, na qual, cada dispositivo tem a sua Fila de Comando.

A Figura 26 mostra uma Fila de Comando (*Command Queue*) para cada dispositivo. As filas de comando são utilizadas para comunicação com os dispotitivos, enfileirando tarefas para sere executadas e lidas dos dispositivos.

Figura 26 – Modelo de Execução OpenCL



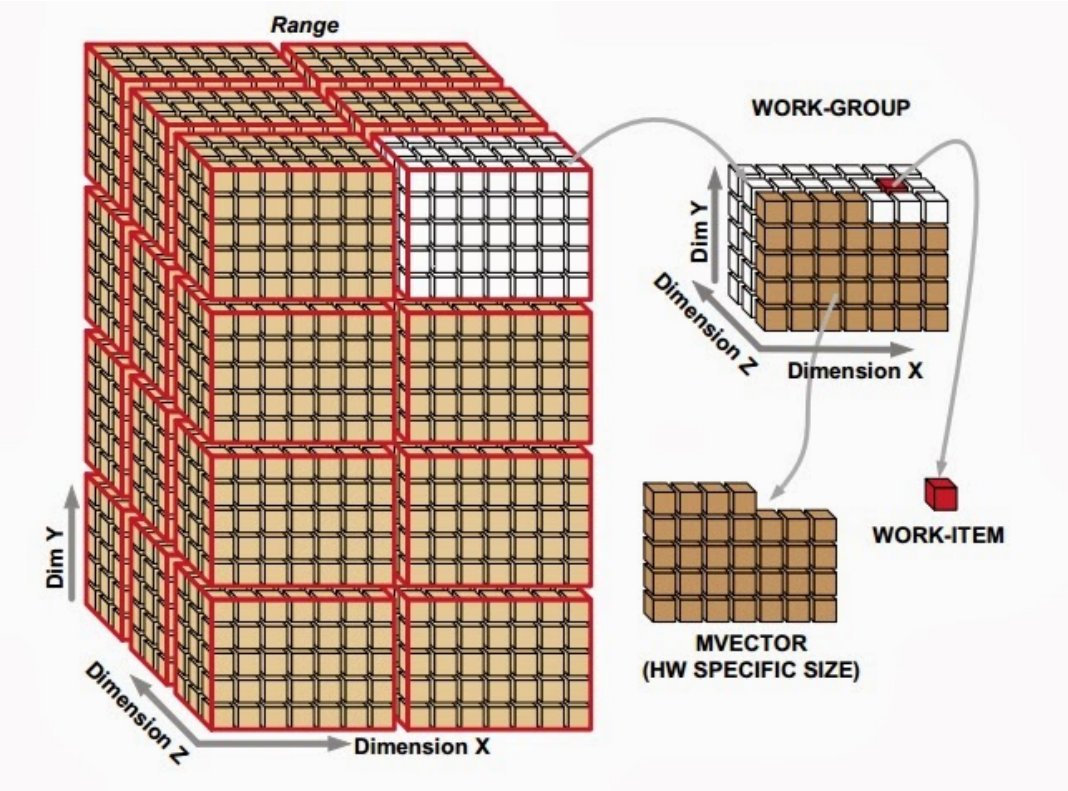
Fonte: AMD (2015)

Com as Filas de Comando um Kernel pode ser executado em um espaço de índices n-dimensional, chamado NDRange e apresentado na Figura 27. No OpenCL uma instância do kernel, conhecida como work-item, é executada em cada ponto do espaço de índices X, Y, Z, e é identificada por um ID. Cada work-ítem executa o mesmo código, porém com dados diferentes.

A Figura 27 mostra como os work-ítens são organizados nas 3 dimensões do NDRange. Os work-items são organizados em work-groups, onde todos os work-ítens de um mesmo work-group são executados paralelamente em um elemento de processamento, e cada work-group é executado em uma unidade de computação. Assim, se um dispositivo conter mais de uma unidade de computação, work-groups

distintos e os work-items dentro de cada work-groups serão todos executados em paralelo.

Figura 27 – NDRange OpenCL



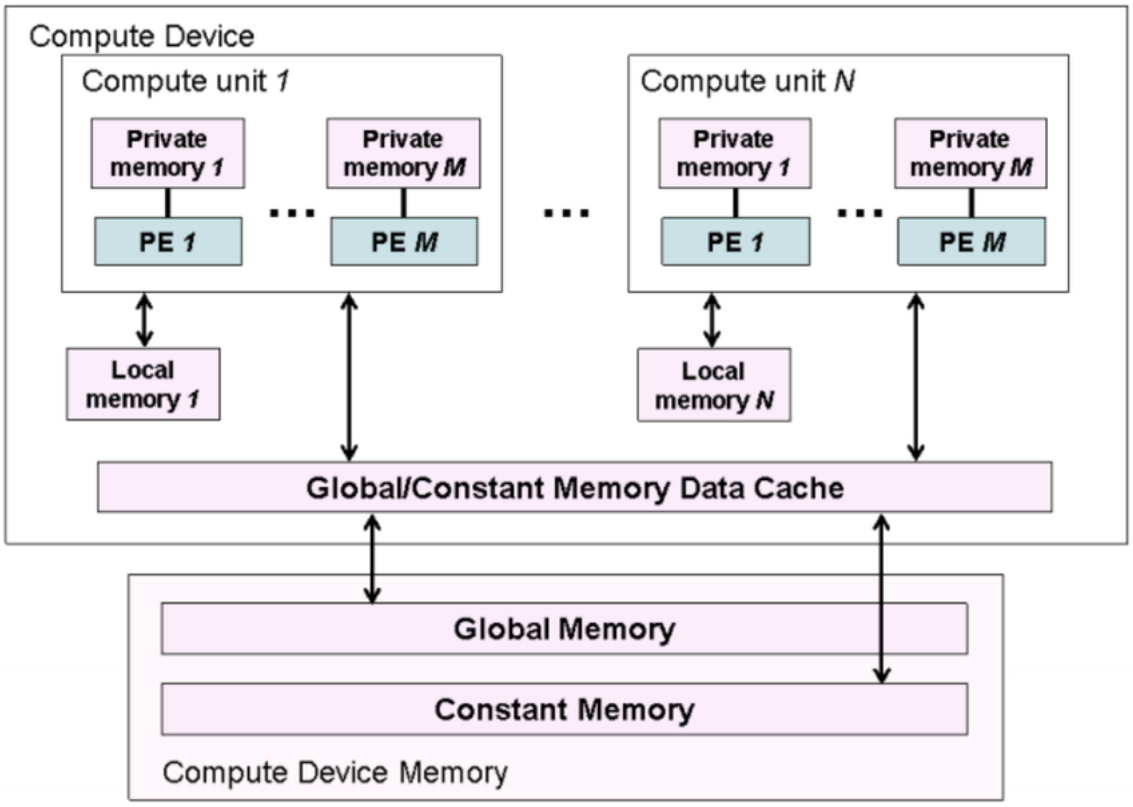
Fonte: AMD (2015)

2.6.3 Modelo de Memória

A Figura 28 apresenta o modelo de memória OpenCL dividido em 4 regiões que podem ser acessadas pelos work-items:

- Memória Global: Esta memória pode ser acessada por todos os work-items de todos os work-groups para leitura e gravação. Essa memória é a maior do modelo OpenCL, porém é a memória com maior latência.
- Memória Constante: É uma região da memória global que é alocada e inicializada pelo host, permanecendo constante por toda a execução dos work-items.
- Memória Local: É uma região local de um work-group e pode ser compartilhada por todos os work-items de um work-group. É uma memória menor que a memória global apresentando com menor latência, e utilizada para compartilhar variáveis em um work-group.
- Memória Privada: É uma região de memória privada para o work-item, e é utilizada para definir as variáveis usadas pelo work-item.

Figura 28 – Modelo de Memória OpenCL



Fonte: THE OPENCL (2009)

2.6.4 Modelo de Programação

O modelo de programação OpenCL é dividido em dois tipos de modelos: Paralelismo de Dados e Paralelismo de Função.

- Paralelismo de Dados: Neste modelo múltiplos dados são processados paralelamente por work-items em um endereço de memória, onde cada work-item é responsável por uma posição de memória a ser processada. Assim, múltiplos work-ítens são instanciados para processar um kernel.
- Paralelismo de Tarefas: É a execução paralela de um simples work-item no dispositivo de computação. Diferente do Paralelismo de dados, onde cada work-ítem fica responsável por processar um endereço de memória, no paralelismo de função o work-item receberá toda a carga de trabalho. Isso permite que múltiplos kernels diferentes sejam executados em um mesmo conjunto de dados, ou em conjunto de dados diferentes.

3 TRABALHOS RELACIONADOS

Muitos trabalhos foram desenvolvidos em GPU's e FPGA's, fazendo uso de suas arquiteturas paralelizadas para acelerar o processo de inferência desses modelos.

Neste capítulo apresentamos as implementações existentes na literatura para GPU's e FPGA's.

3.1 Trabalhos que envolvem o uso de GPU

As Unidades de processamento gráfico - GPU's são processadores especializados em processar dados estruturados em matrizes, atingindo milhares de operações em ponto flutuante por segundo. Essa característica fez com que as GPU's fossem utilizadas não só para processamento gráfico, mas para acelerar algoritmos e o desenvolvimento dos trabalhos de muitos pesquisadores como o treinamento de redes neurais, que apresentam muitos cálculos com matrizes.

Dois trabalhos foram pioneiros com o uso de GPU's para processamento de redes neurais.

Kyoung e Keechul em seu trabalho (OH; JUNG, 2004) demonstraram que uma GPU poderia ser utilizada para acelerar o processamento em uma rede neural. Eles apresentam que a operação do produto interno para cada camada de um Rede Neural pode ser substituída por uma multiplicação de matrizes baseada na acumulação dos vetores de entrada e vetores de peso. A implementação proposta por Kyoung utilizou um ATI RADEON 9700, conseguindo um *speedup* de 20X na multiplicação de matrizes, comparado com uma CPU, para o problema de classificação de textos em uma imagem.

Kumar em seu trabalho (CHELLAPILLA et al., 2006) apresenta uma solução para um rápido processamento das camadas convolucionais, transformando a operação de convolução em multiplicação de matrizes. A mudança apresentou uma melhoria de até 4 vezes no tempo de treinamento de uma CNN em dois problemas de reconhecimento de caracteres, MNIST e Latim. Kumar utilizou uma GPU NVIDIA GeForce 7800 Ultra Graphics Card1 e os resultados foram comparados com o Basic linear algebra subroutines (BLAS).

Algumas bibliotecas mais recentes foram implementadas em OpenCL para explorar sua portabilidade para outras GPU's:

Perkins em seu trabalho (PERKINS, 2015) implementou uma biblioteca em OpenCL para treinamento e inferência de redes neurais convolucionais, apresentando também suporte para Python e Lua. Esta biblioteca implementa as camadas básicas de uma CNN em kernels OpenCL como: camadas de convolução, max-pooling, camadas de ativação e camadas de normalização. A biblioteca é capaz de realizar treinamento e

inferência de modelos de CNN's em GPU's e APU's (Accelerated Processing Unit).

A Intel desenvolveu uma biblioteca em OpenCL para Deep Learning - DL destinados à aceleração da inferência de modelos de DL em processadores gráficos Intel - incluindo HD Graphics e Iris Graphics (COMPUTE. . . , 2017). O clDNN inclui blocos de construção altamente otimizados para implementação de redes neurais convolucionais (CNN) com interfaces C e C++, apresenta suporte à problemas de reconhecimento de imagem, detecção e segmentação de objetos. O clDNN também é uma biblioteca orientada por grafos, em que para se executar um modelo de CNN é preciso construir o grafo e executar. A construção do grafo é feita de forma sequencial, onde cada camada é adicionada uma à uma em uma estrutura chamada *topology* que representa a topologia ou arquitetura da CNN. Após o grafo ser construído o programa é compilado e o treinamento da CNN pode ser iniciado.

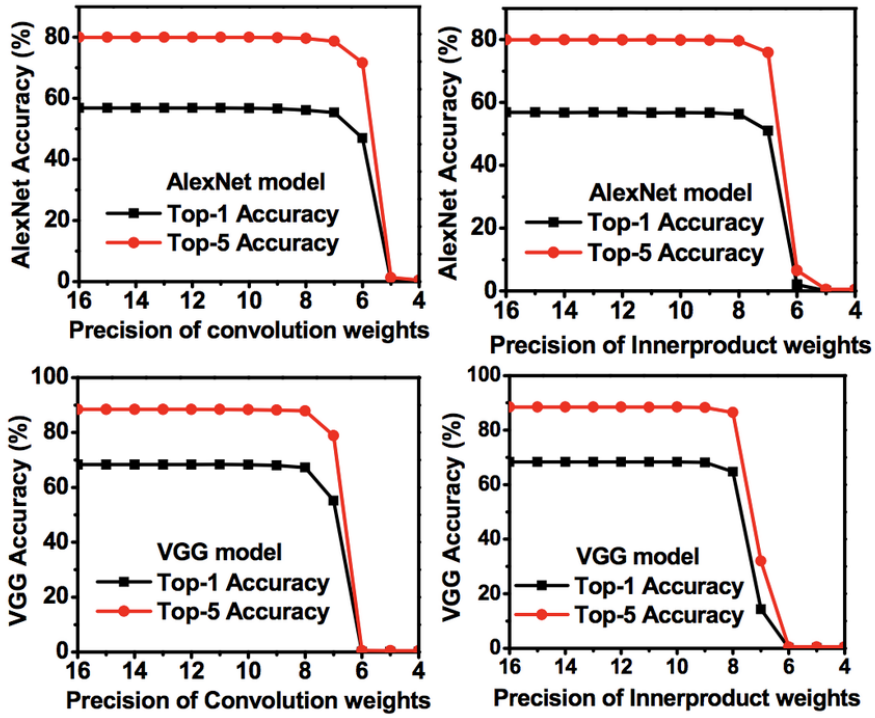
3.2 Trabalhos que envolvem o uso de FPGA

Recentemente houve um aumento no número de trabalhos que têm explorado FPGA's para processamento de Redes Neurais Convolucionais - CNN's, apresentando trabalhos com bilhões de operações de ponto flutuante por segundo - GFLOPS.

Suda em seu trabalho (SUDA et al., 2016) propôs um acelerador CNN em FPGA embarcado para classificação de imagem em larga escala, apresentando um design baseado em multiplicação de matrizes eficiente para todos os tipos de camada na CNN, atingindo 72.4 GOPS (Giga Operações de Por Segundo) e 117.8 GOPS na AlexNet e VGG, respectivamente, em uma Altera Stratix-V FPGA P395-D8

Na Figura 29 os autores apresentam um estudo entre a taxa de acerto e a precisão dos pesos para as arquiteturas VGG e AlexNet. Foi desenvolvido um método de quantização de precisão dinâmica em ponto fixo, dando suporte à precisão de 8 bits e 4 bits para camadas convolucionais e de multiplicação.

Figura 29 – Análise de precição de Ponto Fixo

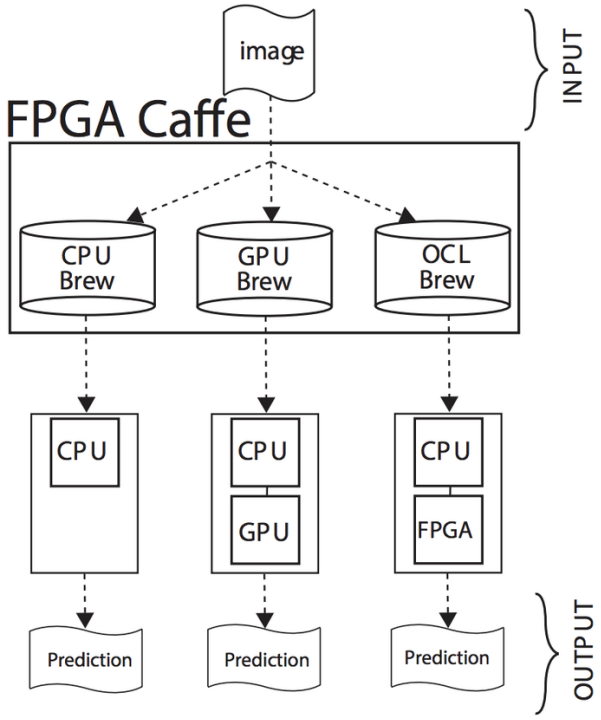


Fonte: SUDA et al. (2016)

DiCecco apresenta em seu trabalho (DICECCO et al., 2016) uma versão modificada do framework Caffe chamada de Caffeinated, permitindo a classificação de CNN com suporte à FPGA. O framework Caffe apresenta um modo de operação chamado de Brew que determina em que dispositivo será executada a CNN. Por padrão o Caffe apresenta Brews para CPU's e GPU's, assim DiCecco implementa um OCL Brew, fornecendo suporte para FPGA da Xilinx. Nessa implementação foi alcançado 55 GFLOPS em uma VGG com float de 32 bits, 42 GFLOPS para a GoogleNet e 45.8 GFLOPS para a AlexNet.

A Figura 30 apresenta a metodologia de projeto do Caffeinated. Assim que uma imagem é passada para o framework Caffe, um dos modos de operação (Brew) é escolhido para processar a inferência da imagem.

Figura 30 – Caffeinated FPGAs

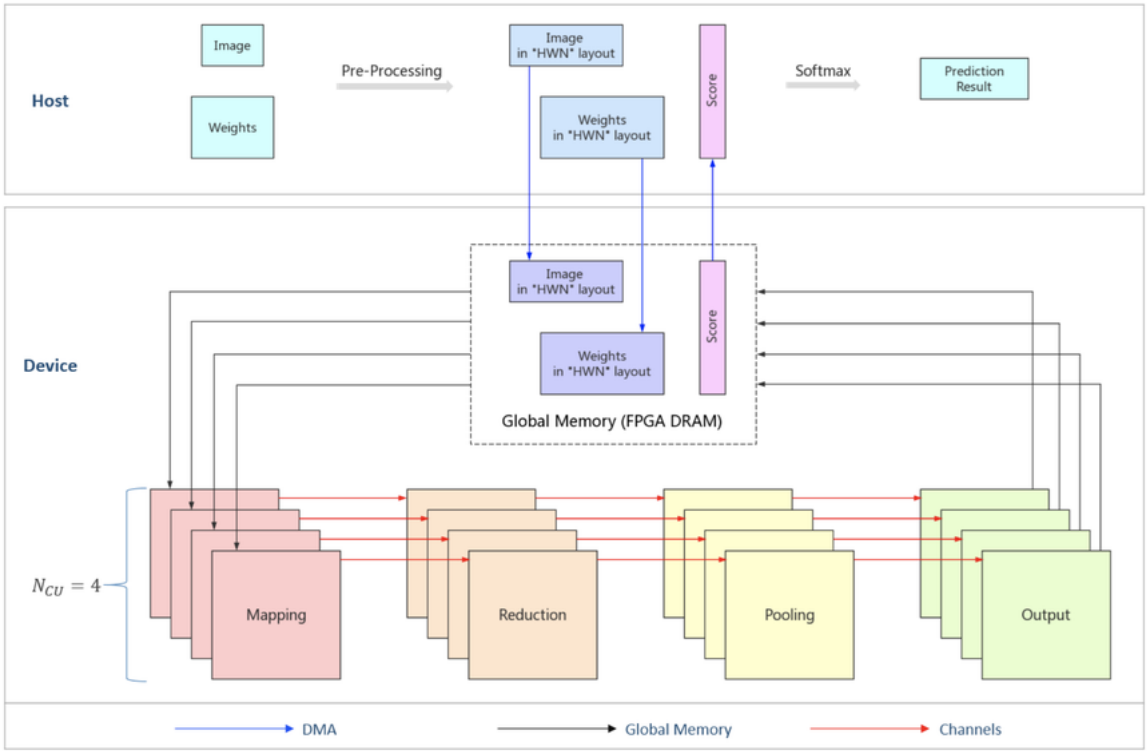


Fonte: DICECCO et al. (2016)

Li (LI, 2017) propôs uma implementação de CNN's com quatro tipos de kernels OpenCL que aproveitam paralelismo de dados e paralelismo de loop. A metodologia de projeto é apresentada na Figura 31 e os kernels implementados são descritos abaixo:

- Kernel de Mapeamento: É projetado como um kernel de NDRange (múltiplos work-items) responsável por ler da memória as imagens de entrada e seus filtros correspondentes.
- Kernel de Redução: É implementado com um único work-item que calcula o resultado da convolução das imagens e executa a função ReLU em cada resultado. O kernel itera sobre todos os pixels da imagem.
- Kernel de Pooling: É implementado com um único work-item projetado para analisar os resultados convolucionais, e aplicar a função de AveragePooling ou MaxPooling em uma imagem.
- Kernel de Saída: É projetado como um kernel de NDRange (múltiplos work-items) responsável por lê os resultados do kernel de pooling e os gravar de volta na memória global.

Figura 31 – Acceleration of Deep Learning on FPGA

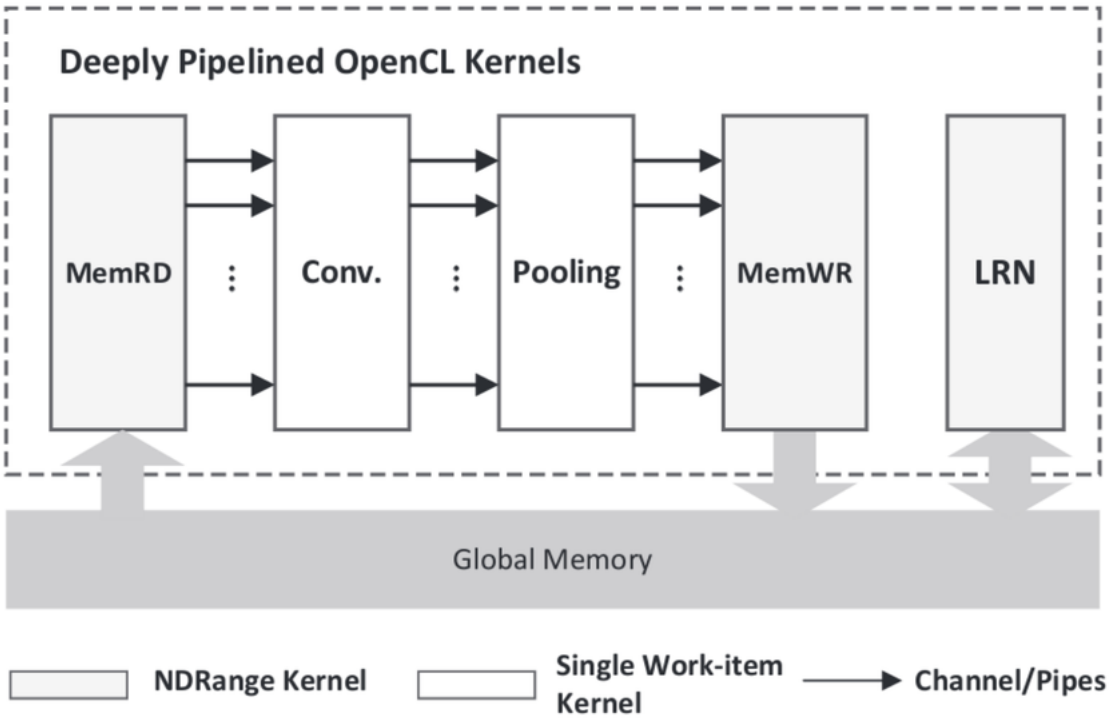


Fonte: Li (2017)

A implementação de Li executa um pipeline de kernels e alcançou em uma arquitetura VGG, com precisão float de 32 bits, 78.24 GFLOPS com consumo de energia de 25W em uma FPGA Intel OpenCL Stratix-V GXA7, e alcançou 145.67 GFLOPS com consumo de energia de 18.6W em uma FPGA Intel OpenCL Arria 10AX115.

Wang (WANG et al., 2017) desenvolveu o PipeCNN, mostrado na Figura 32, um acelerador de redes neurais convolucionais que consiste de um grupo de kernels OpenCL que estão em cascata usando os canais de extensão OpenCL da Altera.

Figura 32 – PipeCNN



Fonte: WANG et al. (2017)

Os kernels formam um pipeline profundo que pode executar uma série de operações básicas da CNN sem a necessidade de armazenar dados da camada intermediária de volta à memória externa. PipeCNN implementa as camadas de Convolução, Pooling, LRN (Local Response Normalization) e Densas, conseguindo um processamento de 15 ms/img para uma arquitetura AlexNet, e 254ms/img para uma arquitetura VGG, para o problema de classificação de imagens em uma FPGA Stratix-V GXA7.

A Tabela 1 apresenta um resumo dos trabalhos relacionados fazendo uma análise de cada trabalho quanto aos dispositivos suportados, a implementação de outras arquiteturas de CNN e a flexibilidade em aceitar arquiteturas de CNN's treinadas em outros frameworks.

Tabela 1 – Resumo dos Trabalhos Relacionados

	Dispositivos Suportados	Arquiteturas Suportadas	Flexibilidade à outros Frameworks	Precisão

	Dispositivos Suportados	Arquiteturas Suportadas	Flexibilidade à outros Frameworks	Precisão
DeepCL - Hugh Perkins	GPU's e APU's	VGG, AlexNet	–	float 32 bits
clDNN - Intel	Intel® Processor Graphics	VGG, Resnet, AlexNet, Inception,	Parser TensorFlow, Caffe	float 32 bits
(Naveen Suda et al. (2016))	FPGA's	VGG, Alexnet	Caffe	Fixo (4 - 32) bits
(Roberto DiCecco et al.(2016))	FPGA's	VGG, AlexNet, Inception	Caffe	float 32 bits
(Huyuan Li (2017))	FPGA's	VGG, AlexNet	Caffe	Fixo 16 bits float 32 bits
(Dong Wang et al. (2017))	FPGA's	VGG, AlexNet	–	float 32
InferenCNN	CPU's, GPU's FPGAS	VGG, Resnet, Inception, AlexMet, MobileNet	Parser Tensorflow	float 32 bits

Fonte: O autor (2019)

4 IMPLEMENTAÇÃO PROPOSTA

Com a grande diversidade de arquiteturas de CNN's e plataformas para a execução desses modelos em diferentes dispositivos e cenários, se faz necessário uma biblioteca que dê suporte para essa variedade de arquiteturas de CNN's e dispositivos de computação.

Neste capítulo apresentamos a InferenCNN[1] uma biblioteca para inferência de CNN's em dispositivos de computação como CPU's, GPU's e FPGA's. A InferenCNN dá suporte à uma grande variedades de arquiteturas de CNN's, implementando as seguintes camadas: Convolução, Convolução Depthwise, ReLU, Sigmoid, arTang, Softmax, Polling, AddResidual, Multiplicação de Matrizes, Padding e BatchNormalization.

Para dar suporte à arquitetura Inception a InferenCNN aceita camadas com mais de uma entrada. Para dar suporte a arquitetura ResNet a InferenCNN implementa o kernel AddResidual para atribuir o dado residual ao dado processado. O kernel de Multiplicação de Matrizes é utilizado para as camadas totalmente conectadas e a camada de Softmax é aplicada na saída da rede para expressar as pontuações de saídas como probabilidades.

Uma breve análise computacional é feita demonstrando o esforço das camadas mais importantes das arquiteturas de CNN's.

4.1 Análise da Complexidade Computacional

A complexidade computacional é medida pela soma das operações de cada camada da CNN. As camadas que apresentam maior complexidade computacional são as camadas de Convolução, Polling e Totalmente Conectadas, por apresentaram a maior quantidade de pesos da rede operando sobre os dados.

4.1.1 Camada de Convolução

A camada de convolução, expressa pela equação 2.3, opera em 4 dimensões: largura do filtro (x'), altura do filtro (y'), canais do filtro (c) e quantidade de filtros (f).

A feature de saída (Fout) de dimensões (Rx x Ry) são produzidas pela convolução dos filtros (W), de dimensões (x' x y'), com a feature de entrada (Fin), de dimensões (X x Y). Assim, a quantidade de operações em uma camada de convolução é expressa pela seguinte equação.

$$Conv_{op} = 2 * X * Y * c * x' * y' * f \tag{4.1}$$

[1] Código, parser e dados disponíveis em: https://github.com/jeffersonramoslucas/InferenCNN

O fator 2 corresponde às duas operações por elemento: Adição e Multiplicação.

A complexidade de tempo para cada convolução é expressa pela equação abaixo, em que X e Y são as dimensões da image e x'e y' são as dimensões dos filtros de convolução.

$$T(n) = O(x'y'XY)$$

Considerando que as dimensões da imagem são iguais a *N* e as dimensões dos filtros são iguais a *n*, a complexidade de tempo é expressa por:

$$T(n) = O(n^2N^2)$$

4.1.2 Camada de Pooling

A camada de Polling utilizada nesta implementação foi a MaxPooling, expressa pela equação abaixo, que executa a função de máximo sobre uma image de dimensões (X x Y x c), com uma janela de tamanho determinado. A janela percorre a feature de entrada (Fin) sem sobreposição, fazendo com que as dimensões de Fout sejam menores que as das dimensões de Fin, por um fator equivalente a dimensão da janela.

$$Fout(x', y', c) = max_{Jw,Jh}{}^{x,y} Fin(i, j) \tag{4.2}$$

A complexidade de computação na camada de Polling é expressa pela seguinte equação:

$$x = Jw * Jh * c * x' * y' \tag{4.3}$$

Jx e Jh são as dimensões da janela de pooling, (x'e y') as dimensões da feature de saída e (c) o número de canais que os filtros de pooling operam.

A complexidade de tempo é expressa pela equação abaixo, em que a parte fracionária se dá pelo fato de não haver sobreposição entre as janelas, apresentando um passo (*stride*) do tamanho das dimensões da janela Jw e Jh.

$$T(n) = O(J_w J_h \frac{x'y'}{J_w J_h})$$

Considerando que as dimensões da imagem são iguais a *N,* a equação se reduz à equação abaixo:

$$T(n) = O(N^2)$$

4.1.3 Camada Totalmente Conectada

A camada totalmente conectada é a multiplicação de todas as features de entrada (Fin) com todos os pesos da camada totalmente conectada (Wfc). Assim, a

complexidade de computação das camadas totalmente conectadas é expressa pela seguinte equação:

$$x = 2 * F_{in} * W_{fc} \tag{4.4}$$

O fator 2 corresponde às duas operações por elemento: Adição e Multiplicação.

A complexidade de tempo para a camada totalmente conectada é expressa pela equação abaixo:

$$T(n) = O(F_{in} * W_{fc})$$

Considerando que a quantidade de neurôneos e a quantidade de features de entrada são iguais a *N,* a equação se reduz à equação abaixo:

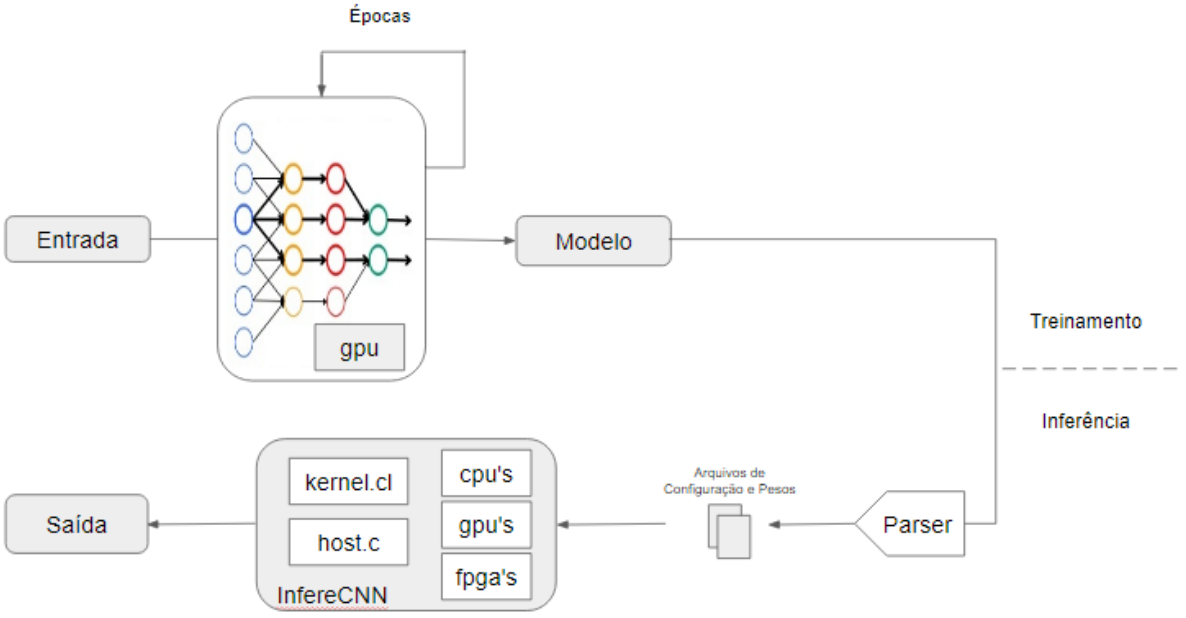
$$T(n) = O(N^2)$$

4.2 Metodologia de Projeto

Nesta seção apresentamos detalhes da metodologia de projeto utilizada e o passo a passo da execução da biblioteca, partido do treinamento da CNN até a fase de inferência. A metodologia proposta é apresentada na Figura 33, descrevendo a fase de treinamento, o parser de pesos, e a fase de inferência da biblioteca proposta.

A fase de treino é descrita na seção 4.2.1 descrevendo as arquiteturas e datasets que foram utilizados. O parser é explicado na seção 4.2.2 descrevendo os arquivos de configuração e de pesos gerados. O mapeamento da InferenCNN nos dispositivos de computação é explicado na seção 4.2.3.

Figura 33 – Metodologia de Projeto da InferenCNN



Fonte: O autor (2019)

4.2.1 Fase de Treino

O processo de treinamento é realizado em uma GPU com os frameworks Tensorflow e Keras, implementando em python as arquiteturas de CNN's: VGG16, AlexNet, Inception, Resnet e MobileNet.

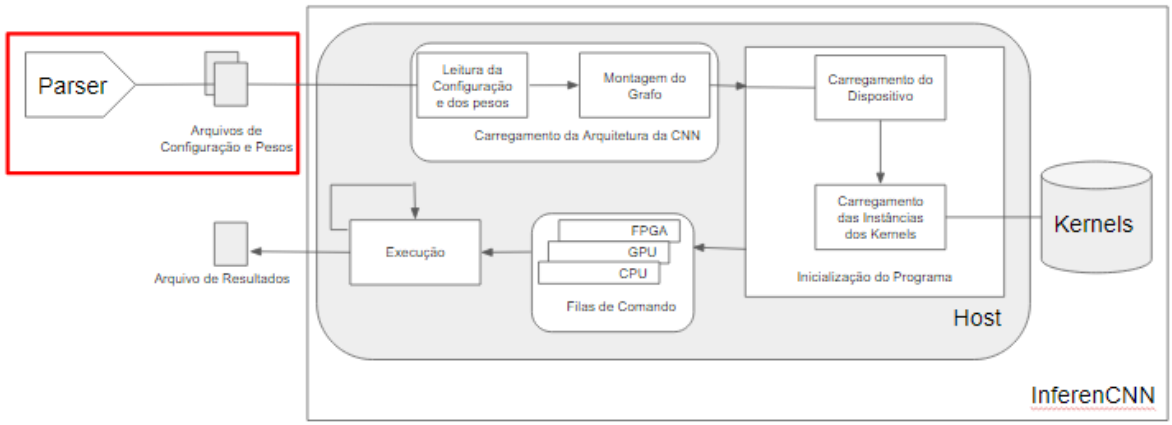
As arquiteturas de CNN's foram treinadas em diferentes datasets: Mnist, Cifar10 e FashionMnist. O treinamento é realizado com imagens de tamanho 128x128 e 224x224, e os pesos treinados são exportados e mapeados para a biblioteca a fim de se realizar a inferência.

4.2.2 Parser Tensorflow

O parser é uma camada entre o treinamento e a inferência, e tem a finalidade de mapear arquiteturas de CNN's treinadas em Tensorflow exportando-as para uma especificação que será carregada pela InferenCNN. O resultado deste mapeamento gera 2 tipos de arquivos: o arquivo de configuração e os arquivos de pesos.

A Figura 34 esquematiza a geração dos arquivos de configuração e pesos que servem como entrada para a InferenCNN, os outros processos apresentados na imagem serão discutidos nas próximas seções.

Figura 34 – Parser Tensorflow



Fonte: O autor (2019)

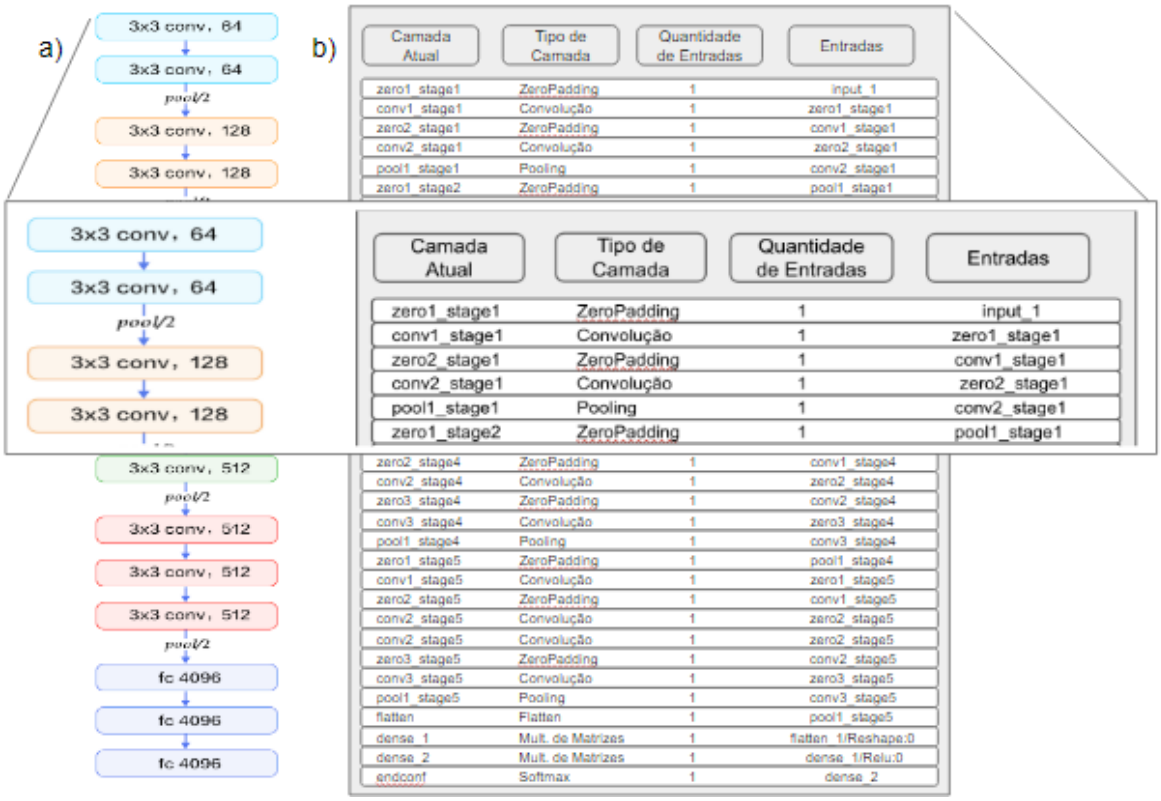
4.2.2.1 Arquivo de Configuração

O arquivo de configuração apresenta características da arquitetura da CNN a nível de fluxo de camadas. Ele é gerado automaticamente a partir do parser e descreve a CNN na biblioteca como um grafo. A Figura 35 apresenta um exemplo para uma VGG16. A imagem *a)* é a arquitetura VGG16 apresentada por Simonyan em (KAREN SIMONYAN; ANDREW G. HOWARD, 2014) e a imagem *b)* é o arquivo de configuração gerado pelo parser.

O arquivo de configuração é composto por uma sequência de 4 atributos para cada camada. O primeiro atributo é o nome da camada em destaque, o segundo atributo é o tipo dessa camada que pode ser um dos tipos implementados na InferenCNN, o terceiro atributo é a quantidade de entradas que a camada receberá, e o quarto atributo é o nome da camada de entrada.

Como a VGG é uma arquitetura sequencial, a arquitetura não apresenta nenhuma camada com mais de uma entrada e todas as camadas têm como entrada a camada anterior.

Figura 35 – Arquivo de Configuração VGG16



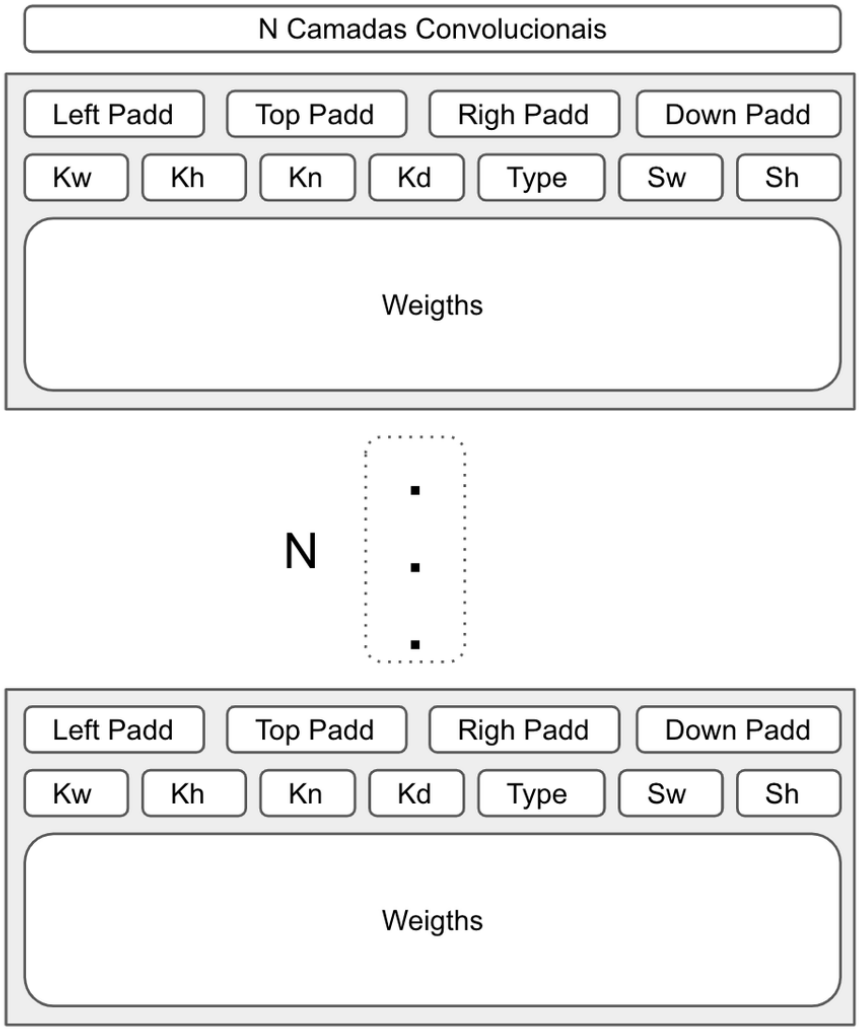
Fonte: O autor (2019)

4.2.2.2 Arquivos de Pesos

Os arquivos de pesos contém os pesos da arquitetura CNN treinada para um determinado problema, apresentando também as características de cada camada. Os arquivos são divididos em 3 tipos:

- Arquivos de pesos convolucional: O arquivo de pesos é expresso por uma sequência de informações para cada camada convolucional. A Figura 36 mostra o padrão para N camadas convolucionais, apresentando informações de *Padding* (*Left Padd*, *Top Padd*, *Righ Padd*, *Down Padd*), dimensões dos filtros convolucionais (Kw x Kh), quantidade de filtros convolucionais (Kn), quantidade de canais de cada filtro (Kd), tipo da convolução (Type), e o stride de cada convolução (Sw x Sh). Logo após essas informações, segue a sequência dos pesos convolucionais.

Figura 36 – Configuração do Arquivo de Pesos Convolucionais



Fonte: O autor (2019)

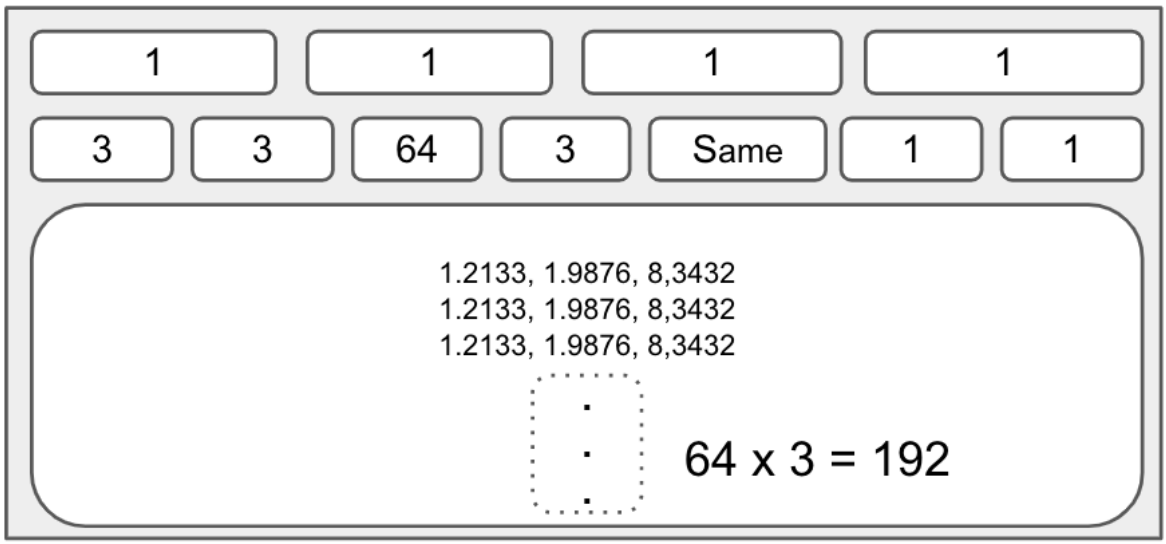
A Figura 37 apresenta um exemplo de arquivo de peso para a primeira camada de uma VGG16 com a seguinte configuração:

- Left Padd = 1,
- Top Padd = 1,
- Righ Padd = 1
- Down Padd = 1
- Dimensões dos filtros convolucionais (Kw x Kh) = 3 x 3
- Quantidade de filtros convolucionais (Kn) = 64
- Quantidade de canais de cada filtro (Kd) = 3

- Tipo da convolução (Type) = same
- Stride de cada convolução (Sw x Sh) = 1 x 1

Logo após essas informações, segue a sequência de pesos dos 64*3 filtros convolucionais, que é a multiplicação da quantidade de filtros (Kn) pela quantidade de canais de cada filtro (Kd).

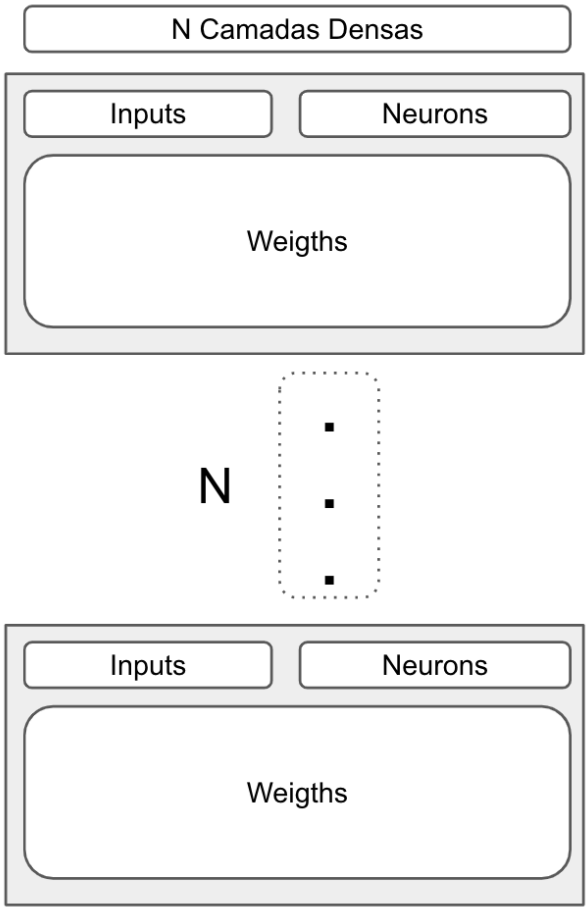
Figura 37 – Exemplo do arquivo de pesos para a primeira camada da VGG16



Fonte: O autor (2019)

- Arquivos de pesos denso: Contém as informações da parte densa das CNN's, como a quantidade de camadas densas, e os pesos de cada camada. A Figura 38 mostra a configuração do arquivo de pesos densos para N camadas densas, em que *Inputs* é a quantidade de entradas da camada densa e *Neurons* é a quantidade de neurônios da camada. Logo após, segue a sequência de pesos da camada.
- Arquivos de pesos de saída: Contém as informações da última camada densa (a camada de softmax) especificando os pesos da camada e a quantidade de saídas para a classificação. O arquivo de pesos de saída segue o mesmo padrão da Figura 38.

Figura 38 – Configuração do Arquivo de Pesos Densos



Fonte: O autor (2019)

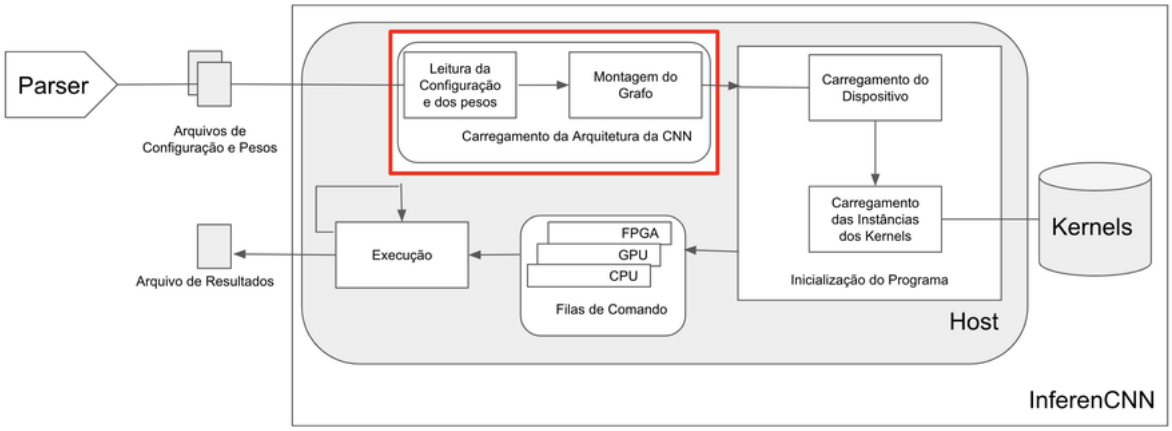
Esses dois tipos de arquivos serão utilizados para fazer o mapeamento das arquiteturas de CNN's, realizando o carregamento da configuração e dos pesos de cada camada da arquitetura.

4.2.3 Fase de Mapeamento

O processo de mapeamento é dividido em 4 etapas que são descritas abaixo:

- Carregamento da Arquitetura de CNN: O mapeamento da CNN nos dispositivos de computação se inicia com o carregamento da arquitetura da CNN ao ler o arquivo de configuração que contém todas as informações da sequência de camadas, sendo possível já saber as camadas envolvidas e o fluxo de propagação dos dados. Este processo é apresentado na Figura 39, em que todas as camadas são carregadas uma a uma, e buffers de memória são criados para carregar os pesos de cada camada a partir dos arquivos de pesos.

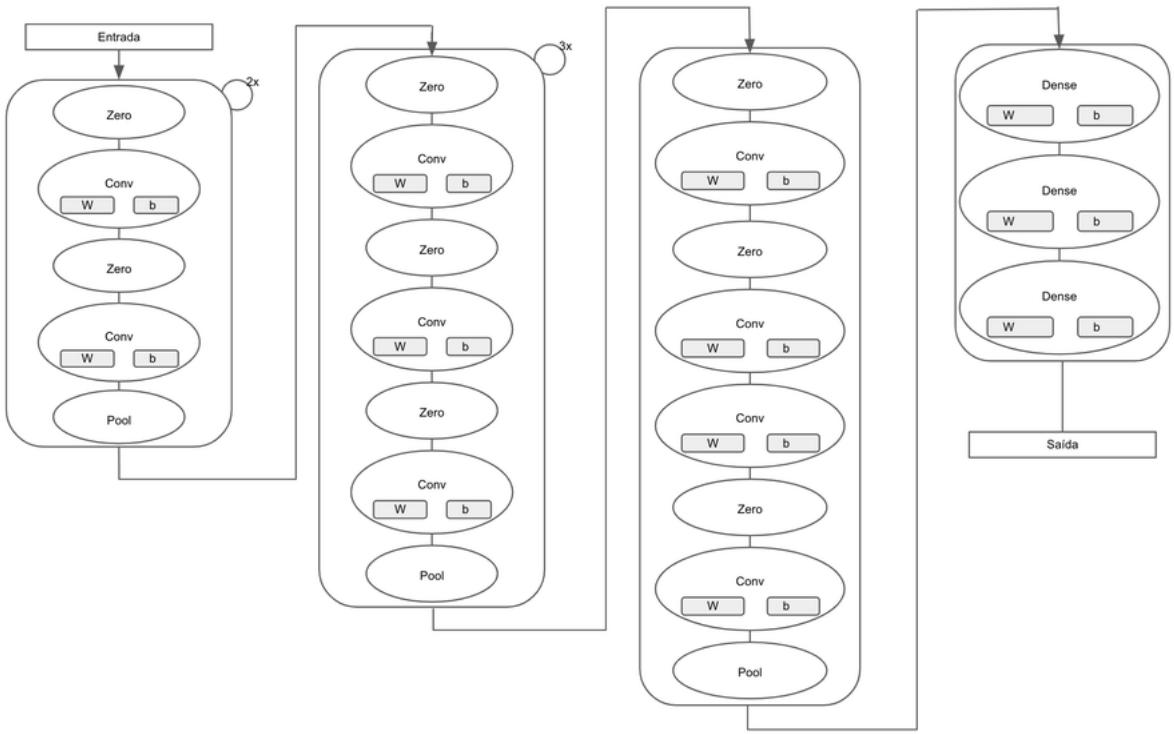
Figura 39 – Carregamento da Arquitetura da CNN



Fonte: O autor (2019)

Com a leitura da arquitetura e dos pesos da CNN um grafo de fluxo é gerado para ser executado dos dispositivos de computação seguindo a ordem das camadas e carregando os pesos associados a cada camada. A Figura 40 mostra o grafo de fluxo para uma arquitetura VGG. O grafo é utilizado na etapa de execução que será descrito mais à frente. Na camada de convolução W e b são referentes aos pesos e bias que serão multiplicados com a imagem de entrada.

Figura 40 – Grafo de Fluxo para uma VGG16

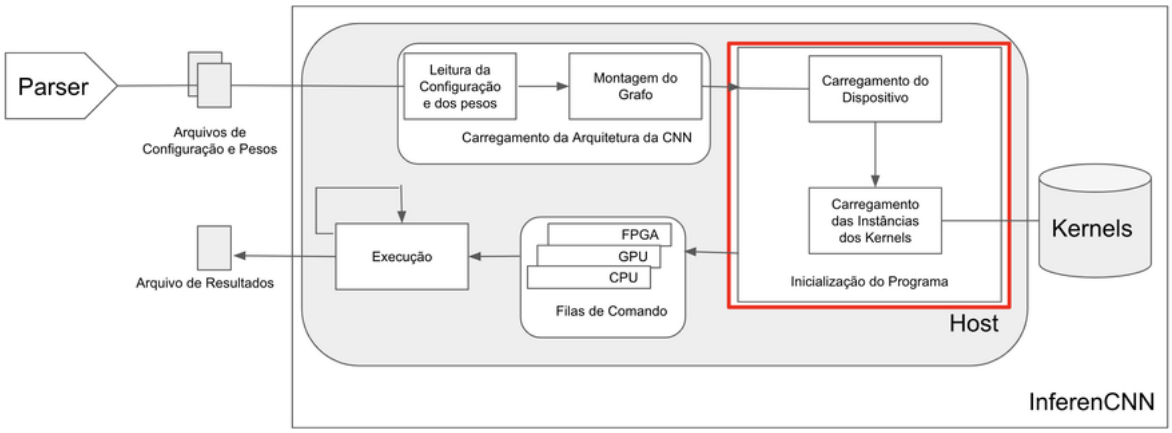


Fonte: O autor (2019)

- Inicializando do Programa: Na inicialização do programa o dispositivo de compu-

tação é carregado através da função Ocl::init_opencl()[2], apresentado na Figura 41. Este processo é iniciado definindo primeiramente a plataforma OpenCL do dispositivo, sendo algumas possíveis plataformas: Intel, AMD, Altera, Nvidia e Apple. Após definido a plataforma, uma lista de dispositivos disponíveis é carregada, e um contexto é definido carregando todas as informações necessárias do dispositivo como: memória disponível e unidades de computação, para gerenciar o lançamento dos kernels e a alocação de memória no dispositivo. O próximo passo é o carregamento das instâncias dos kernels, que é feito pelo programa OpenCL, um binário que contém as implementações que serão executadas nos dispositivos. Para CPU"s e GPU's, o programa é criado compilando os arquivos de kernels através da função *creat_program()*[2], para FPGA's o SDK da Intel é utilizado para sintetizar os kernels e gerar o binário do programa. A InferenCNN já contém o binário gerado para FPGA's contendo todas as camadas descritas no início deste capítulo.

Figura 41 – Inicialização do Programa OpenCL

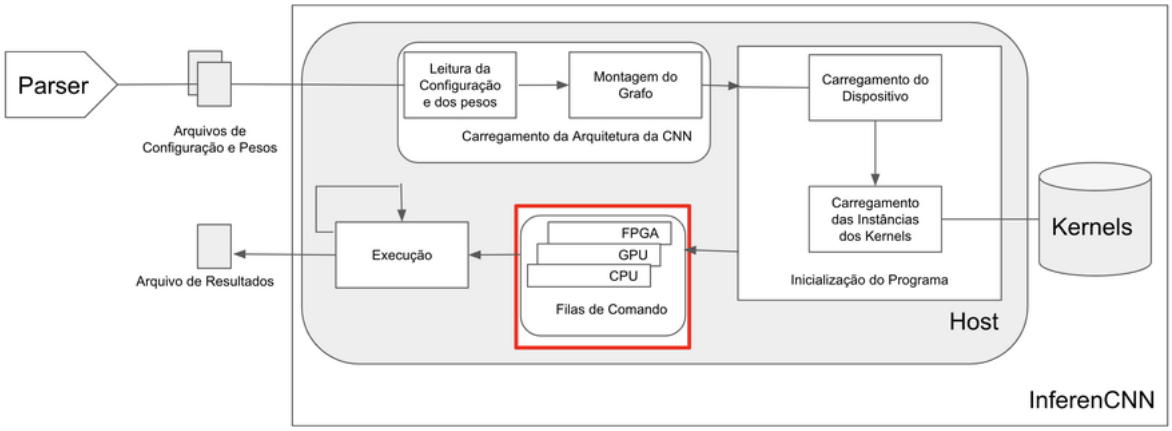


Fonte: O autor (2019)

- Inicialização das Filas de Comando: A Figura 42 mostra a etapa de Inicialização das Filas de Comando. São as filas de comandos que gerenciam os lançamentos e sincronização das instâncias dos kernels que serão executados no dispositivo.

[2] Arquivo Ocl.cpp: https://github.com/jeffersonramoslucas/InferenCNN

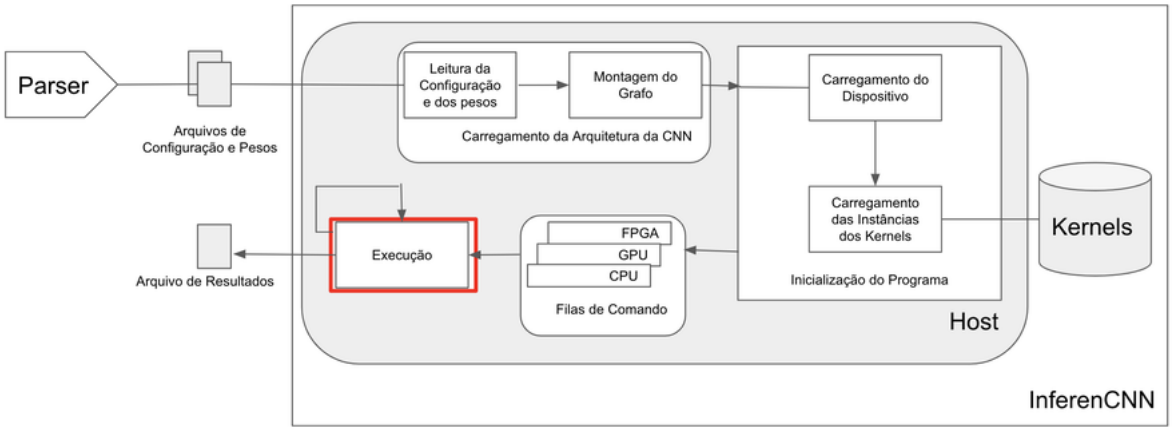
Figura 42 – Inicialização das Filas de Comando



Fonte: O autor (2019)

- Execução: A Figura 43 apresenta a etapa de execução com todos os objetos necessários inicializados. Na etapa de execução as imagens são propagadas camada por camada, o grafo de fluxo recebe uma imagem e inicia o processo de inferência processando a imagem na primeira camada da CNN. A informação resultante é propagada para a próxima camada até chegar a última camada da CNN, onde o resultado é exportado em um arquivo de resultados e a próxima imagem é carregada para se realizar a inferência. Cada camada tem o seu modelo de memória e de lançamento de work-ítens que será descrito na seção 4.4.

Figura 43 – Execução



Fonte: O autor (2019)

Os passos apresentados anteriormente são comuns para qualquer aplicação em OpenCL, desde a inicialização do dispositivo OpenCL, até o lançamento dos kernels, assim, o host do OpenCL não sofre muita alteração com problemas distintos.

4.3 Implementação C/C++ Canônica

A versão Canônica é uma implementação em C/C++ desenvolvida sem nenhum tipo de paralelismo, a não ser o de instruções realizado pelo próprio compilador. Foi a primeira versão desenvolvida da biblioteca, mostrando que o grafo de fluxo é compatível para ser implantado em outras linguagens imperativas.

Cada camada da CNN foi desenvolvida de forma modular, onde cada uma delas é uma função parametrizada que é chamada pelo grafo de fluxo assim que for necessário. O desenvolvimento dessa versão foi essencial para a biblioteca, pois foi possível enxergar pontos de otimizações de processamento e de memória, realizar debugers facilmente, profiles para identificar gargalos, e pontos independentes que foram facilmente paralelizados.

Abaixo descrevemos pontos importantes da versão em C/C++:

- Alocação dos endereços de memória: A alocação dos endereços de memória, é feita antecipadamente, já que é possível calcular todos os tamanhos dos buffers de entrada e saída.
- Validação funcional: A versão em C/C++ serviu para validação funcional dos componentes. Cada camada foi implementada como uma função que é chamada pelo grafo de fluxo, processando o dado camada por camada.
- Gargalos: Nesta versão o profile do Visual Studio (MICROSOFT, 1997) foi executado, mostrando que a maior parte do processamento está nas Camadas Convolucionais e Totalmente Conectadas, necessitando de um foco maior em otimizações.
- Consumo de memória: O profile também mostrou que o maior consumo de memória está nas camadas convolucionais, pelo armazenamento dos seus filtros de convolução.

4.4 Implementação OpenCL

Parte da implementação OpenCL foi descrita na seção 4.2.3, mostrando como o host é construído. Nesta seção iremos focar na descrição dos Kernels OpenCL, apresentando detalhes de implementação e algumas possíveis otimizações.

4.4.1 Construção dos Kernels:

Abaixo descrevemos os detalhes de implementação dos kernels GPU's, e no capítulo 5 descrevemos algumas otimizações necessárias para FPGA's. Para cada

camada é apresentada uma imagem mostrando um pouco mais de detalhes da implementação em OpenCL.

4.4.1.1 Kernel Convolução

A Figura 44 apresenta uma pseudo-código da implementação do kernel de convolução.

O kernel de convolução é lançado em 3 dimensões do NDRange que foi descrita na seção 2.6.2. Cada work-item fica responsável por processar cada pixel da imagem, apresentando paralelismo nas linhas, colunas e canais da imagem. A imagem é lida da memória global, e parte dos pesos de convolução são carregados na memória local, apresentando um ganho no desempenho por ter rápido acesso e ser compartilhada por work-items de um mesmo work-group. O resultado da convolução é armazenado de volta na memória global.

Figura 44 – Kernel de Convolução

Algorithm 1: Pseudo-Código Kernel de Convolução

Result: Convolução entre uma imagem e uma camada convolucional

```
size_t global_id = get_global_id(0);
size_t idRow = get_global_id(1);
size_t idCol = get_global_id(2);

for (int kerD=0; kerD < depth; kerD++) do
    for (int i = 0; i < ker_rows_vs_cols; i++) do
        ker_local =: Carregar parte os pesos da rede na memória local
    end

    for (int i = 0; i < ker_rows_vs_ker_cols; i++) do
        sum +=: Computar a multiplicação do kernel local com a imagem
    end

    conv_out[Id] += sum
end
```

Fonte: O autor (2019)

4.4.1.2 Kernel de Convolução DepthWise

O kernel de DepthWise é uma convolução similar com a convolução padrão, mudando o fato de que os canais da imagem não são combinados. A alteração na Figura 44 se daria que a variável conv_out não seria acumulada, e um ID diferente é gerado para cada mapa de features.

4.4.1.3 Kernel de Polling

O kernel de Polling executa em apenas uma direção, onde cada work-item fica responsável por processar toda a imagem. A imagem de entrada é lida da memória global, processada pegando o valor máximo da janela de dimensões (pollingSizeX x pollingSizeY), e o resultado é armazenado de volta na memória global.

Figura 45 – Kernel de Polling

Algorithm 2: Pseudo-Código Kernel de Pooling

Result: Operação de MaxPolling aplicado em uma imagem

```
size_t global_id = get_global_id(0);

for (i = 0; i < pool_out_rows; i = i + strideX) do
    for (j = 0; j < pool_out_cols; j = j + strideY) do
        pool_out[Id] =: Máximo da janela (poolingSizeX x poolingSizeY)
    end
end
```

Fonte: O autor (2019)

4.4.1.4 Kernel de FC

A Figura 46 apresenta o pseudo código do kernel de multiplicação de matrizes que é utilizado para processar as camadas totalmente conectadas.

O kernel da camada Totalmente Conectada realiza a multiplicação das features de entrada e os pesos. O kernel é executado em apenas uma direção, em que cada work-item processa paralelamente cada posição dos dados. Parte das features e dos pesos são carregados para a memória local, logo após a multiplicação é feita em um loop desenrolado, e o resultado da multiplicação é armazenado de volta na memória global.

Figura 46 – Kernel de Totalmente Conectada

Algorithm 3: Pseudo-Código Kernel da Camada Totalmente Conectada

Result: Operação de Multiplicaccão entre as features e os pesos da FC

```
size_t global_id = get_global_id(0);

for (int a = as, b = bs; a ≤ ae; a += BLOCK, b += (BLOCK * B_width) do
    features_local =: carrega features na memória local
    weigth_local =: carrega pesos da rede na memória local
    barrier_memory

    #pragma unroll
    for (int k = 0; k < BLOCK; ++k) do
        sum =: computa a multiplicação e soma de cada bloco
    end
    barrier_memory
end

OUT = sum
```

Fonte: O autor (2019)

4.4.1.5 Kernel de BatchNormalization

O algoritmo BatchNormalization foi descrito na seção 2.4.1.8, e na Figura 47 o descrevemos com mais detalhes de implementação. O kernel é lançado em 1 dimensão, e cada work-item é lançado paralelamente, responsável por varrer as linhas e colunas de uma imagem e computar o batch normalization em cada pixel. A imagem de entrada é lida da memória global, processada e os resultados são armazenados de volta na memória global.

Figura 47 – Kernel de Batch Normalization

Algorithm 4: Pseudo-Código Kernel de Batch Normalization

Result: Operação de Batch_Normalization aplicado em uma imagem

```
size_t global_id = get_global_id(0);

for (int i = 0; i < img_rows; i++) do
    for (int j = 0; j < img_cols; j++) do
        out[Id] =: Batch Normalization em cada (i,j) da imagem
    end
end
```

Fonte: O autor (2019)

4.4.1.6 Kernel addResidual

A Figura 48 mostra o pseudo código do kernel de AddResidual, que faz a soma de uma imagem processada por uma sequência de camadas com outra imagem que não foi processada.

O kernel addResidual realiza a soma da imagem com um dado residual, camada descrita na seção 2.4.1.12. Similar ao Kernel de Convolução, o addResidual é lançado em 3 dimensões do NDRange, em que cada work-item ficam responsáveis por processar cada pixel da imagem de entrada. As imagens de entrada são carregadas da memória global, processadas e os resultados são armazenados de volta na memória global.

Figura 48 – Kernel de AddResidual

Algorithm 5: Pseudo-Código Kernel AddResidual

Result: Operação de Add aplicado em uma imagem e o dado residual

```
size_t global_id = get_global_id(0);
size_t IDRow = get_global_id(1);
size_t IDCol = get_global_id(2);
id =: Calcular posição do (i,j) para cada image

Out[id] = img[id] + res[id];
```

Fonte: O autor (2019)

4.4.1.7 Kernels Ativação

Os kernels de ativação executam uma função de ativação em cima do próprio buffer de entrada, sem necessitar de um novo buffer de saída. A Figura 49 apresenta alguns kernels de ativação, paralelizados em apenas 1 dimensão, em que cada work-ítem fica responsável por processar a função de ativação em 1 pixel da imagem. A imagem é carregada da memória global, processada, e o resultado é armazenado de volta na memória global.

Figura 49 – Kernels de Ativação

Algorithm 6: Pseudo-Código Kernel ReLU

Result: Operação de ReLU aplicado em uma imagem

```
size_t global_id = get_global_id(0);

img[globalId] = (img[globalId] > 0?img[globalId] : 0);
```

Algorithm 7: Pseudo-Código Kernel Tanh

Result: Operação de Tanh aplicado em uma imagem

```
size_t global_id = get_global_id(0);

img[globalId] = tanh(img[globalId]);
```

Algorithm 8: Pseudo-Código Kernel Sigmoide

Result: Operação de Sigmoide aplicado em uma imagem

```
size_t global_id = get_global_id(0);

img[globalId] = (1.0f / (1 + exp(-img[globalId])));
```

Algorithm 9: Pseudo-Código Kernel Elu

Result: Operação de Elu aplicado em uma imagem

```
size_t global_id = get_global_id(0);

img[globalId] = (img[globalId] > 0?img[globalId] : exp(img[globalId]) − 1);
```

Fonte: O autor (2019)

4.4.1.8 Kernel de Padding

A Figura 50 mostra o pseudo código do kernel de Padding, que faz a cópia dos dados para o novo buffer de memória com padding.

Como descrito na seção 4.2.3, os objetos de memória já são alocados na fase do Carregamento dos Pesos da Rede. Assim, o kernel de padding faz apenas o preenchimento do novo buffer a partir da posição inicial do padding. O kernel é lançado em 1 dimensão, e cada work-item é responsável por fazer a cópia de 1 pixel de imagem. A imagem é lida da memória global e copiada para o novo buffer na memória global.

Figura 50 – Kernel de Padding

Algorithm 10: Pseudo-Código Kernel Padding

Result: Operação de Padding aplicado em uma imagem

```
size_t global_id = get_global_id(0);

Id =: Calcular posição do padding

Out[Id] = img[globalId];
```

Fonte: O autor (2019)

5 EXPERIMENTOS E RESULTADOS

A InferenCNN é uma biblioteca que faz uso da portabilidade do framework OpenCL, para processar a inferência de arquiteturas de CNN em CPU, GPU e FPGA.

Neste capítulo apresentamos os experimentos e resultados da InferenCNN, comparando a biblioteca com outros frameworks, implementações em FPGA, e fazemos uma análise das versões implementadas em dispositivos diferentes.

Para os experimentos com FPGA foi utilizado o vLab Academic Cluster (INTEL, 2017a) que oferece um ambiente para o suporte e o avanço de pesquisas acadêmicas. O vLab atualmente consiste em Xeon, Xeon Phi e FPGAs, em que cada nó de computação são conectados entre si pela interconexão Intel Omni-Path (Intel OPA) 100 Series (INTEL, 2017b).

5.1 Configuração dos Experimentos

Os experimentos realizados neste trabalho foram feitos em uma CPU Intel(R) Core(TM) i7-7700HQ CPU 2.80GHz, em uma GPU Nvidia Geforce GTX 1070, e uma FPGA Arria 10 GX1150 FPGA (10AX115U3F45E2SGE3) construída com a tecnologia de 20 nm.

Para realizar os experimentos com CPU e GPU é necessário o uso do Driver OpenCL para alguns dos fabricantes citados anteriormente: Intel, AMD, NVIDA, Apple. A versão Canônica pode ser executada em CPU utilizando apenas um compilador de c++, por não ser exclusivamente OpenCL.

A Tabela 2 mostra os recursos da Geforce GTX 1070, bem como o seu consumo de energia.

Tabela 2 – Especificação Nvidia Geforce GTX 1070

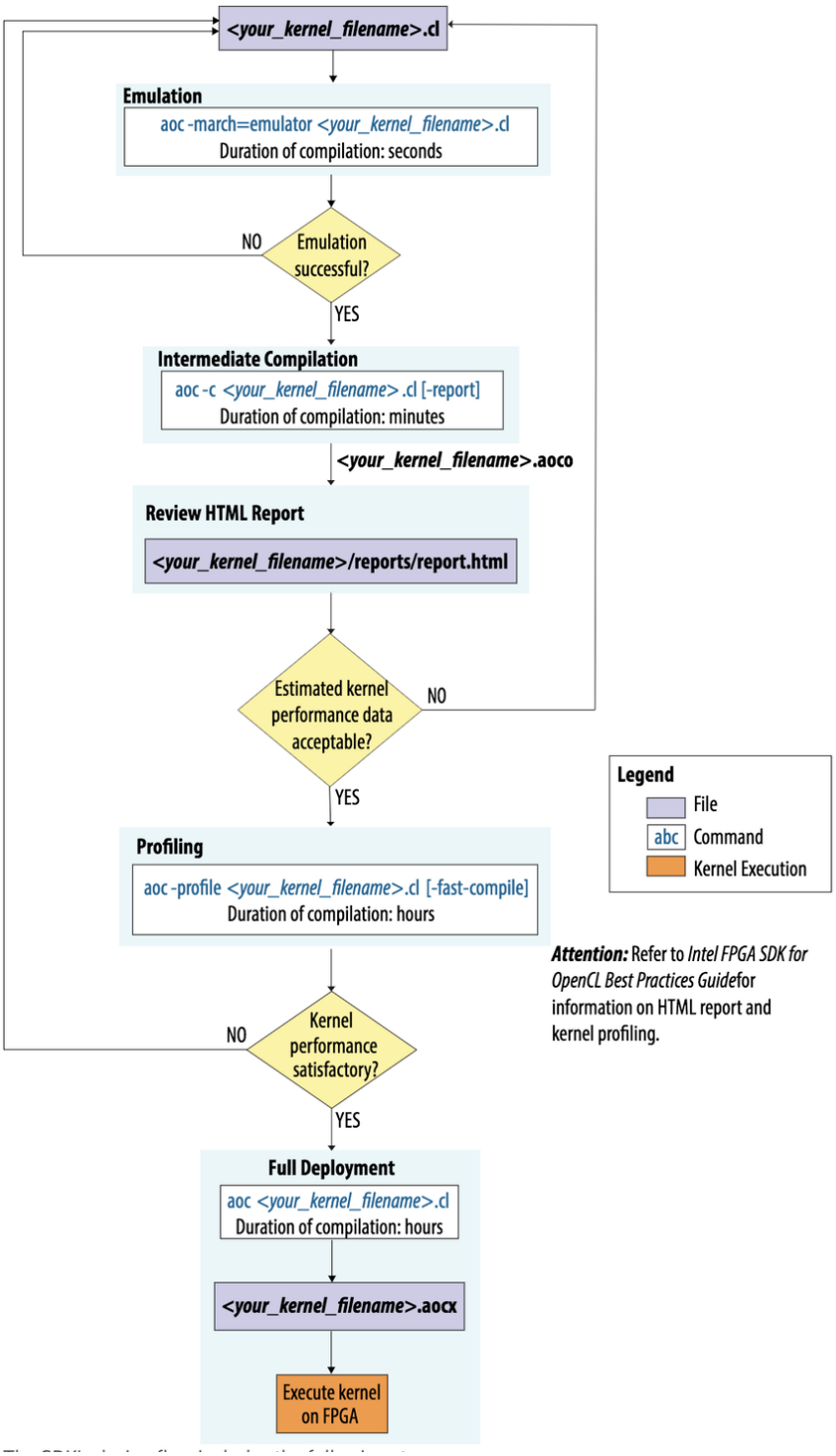
NVIDIA CUDA® Cores	1920
Clock básico (MHz)	1506
Configuração de memória padrão	8 GB GDDR5

NVIDIA CUDA® Cores	1920
Largura de banda de memória (GB/s)	256
Suporte ao bus	PCIe 3.0
Energia da placa de vídeo (W)	150 W

Fonte: O autor (2019)

Para realizar os experimentos com FPGA é necessário o uso do SDK da Intel que segue o fluxo da Figura 51. As etapas ajudam a identificar erros funcionais a partir do emulador, e gargalos de desempenho, passando por uma análise sintática na fase "Intermediate Compilation", sem a necessidade de criar o arquivo de configuração de hardware. Assim que o kernel estiver como desejado, o arquivo *.aocx* é utilizado para ser executado na FPGA.

Figura 51 – Fluxo de Projeto do SDK do Intel® FPGA para OpenCL™



Fonte: INTEL FPGA SDK (2012)

A Tabela 3 mostra as configurações da FPGA Arria GX 1150 que foi utilizada nos experimentos.

Tabela 3 – Especificação Arria 10 GX 1150

Logic Elements (LE) (K)		1,150
ALM		427,200
Register		1,708,800
Memory (Kb)	M20K	54,260
	MLAB	12,984
Variable-precision DSP Block		1,518
18 x 19 Multiplier		3,036
PLL	Fractional Synthesis	32
	I/O	16
17.4 Gbps Transceiver		96
GPIO		768
LVDS Pair		384
PCIe Hard IP Block		4
Hard Memory Controller		16

Fonte: O autor (2019)

As implementações feitas da biblioteca serão comparadas na seção 5.2, a fim de verificar o desempenho em diferentes tipos de dispositivos.

5.2 Análise e Comparações da Biblioteca Proposta

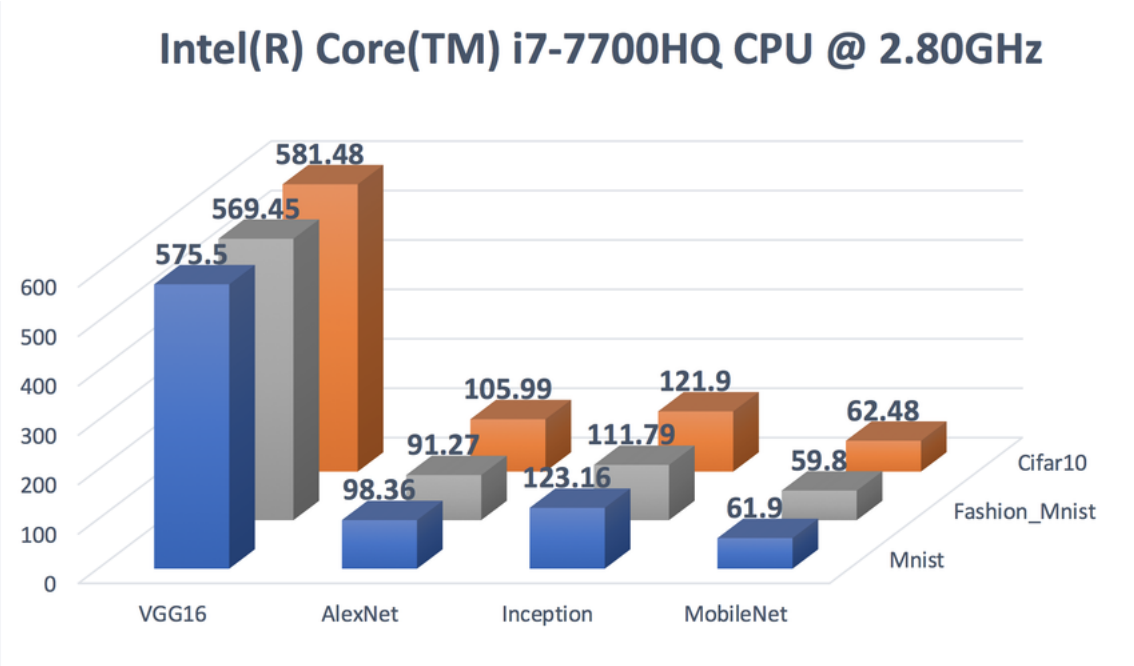
Neste seção apresentamos uma análise de tempo entre todas as implementações feitas da InferenCNN, e comparamos os resultados com o Tensorflow, e aceleradores em FPGA. Também é feito uma análise da versão em FPGA, comparando os recursos utilizados e o tempo para se processar uma imagem, à medida que replicamos o kernel de convolução.

5.2.1 Análise de Tempo

A análise de tempo de processamento é mostrado nas figuras abaixo, detalhando os tempos dos datasets: Mnist, Cifar10 e Fashion Mnist, nas arquiteturas: AlexNet (224x224), VGG16 (128x128), Inception (128x128) e MobileNet (128x128), executando os experimentos com um batch de 32 imagens de cada dataset.

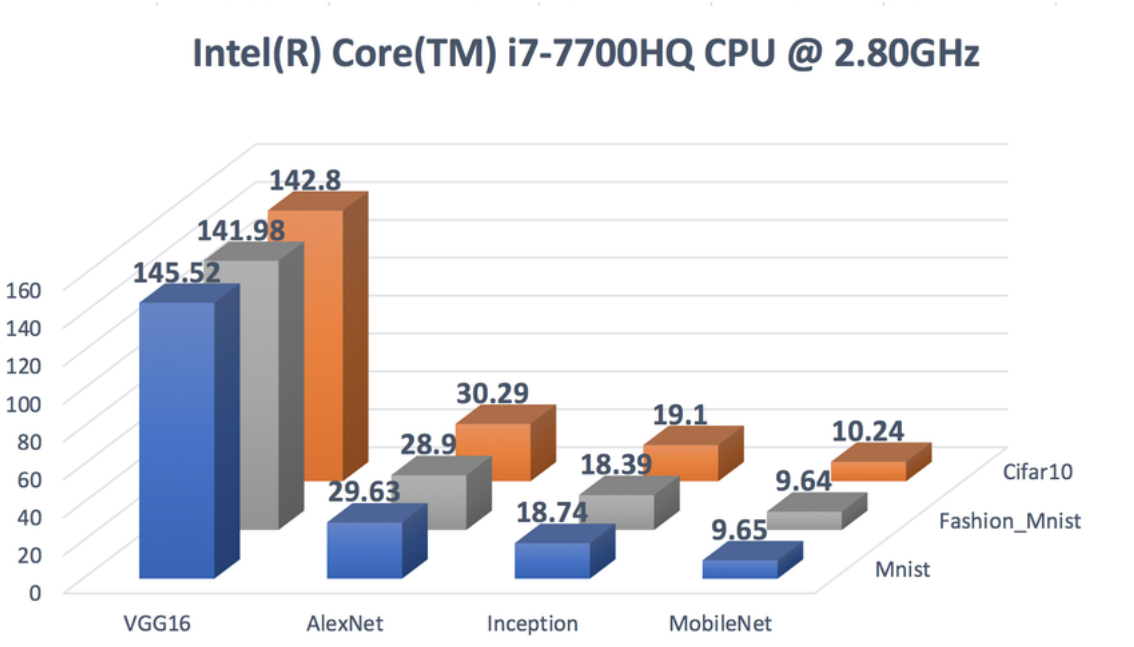
A Figura 52 mostra a análise realizada na versão Canônica, que é executada na CPU sem nenhum paralelismo de *threads*. A Figura 53 mostra a análise realizada na versão OpenCL CPU, explorando o paralelismo de threads OpenCL. Ambas as versões foram executadas em um processador Intel Core i7-7700, apresentando um comportamento similar entre as versões, divergindo apenas pela escala.

Figura 52 – Canônico



Fonte: O autor (2019)

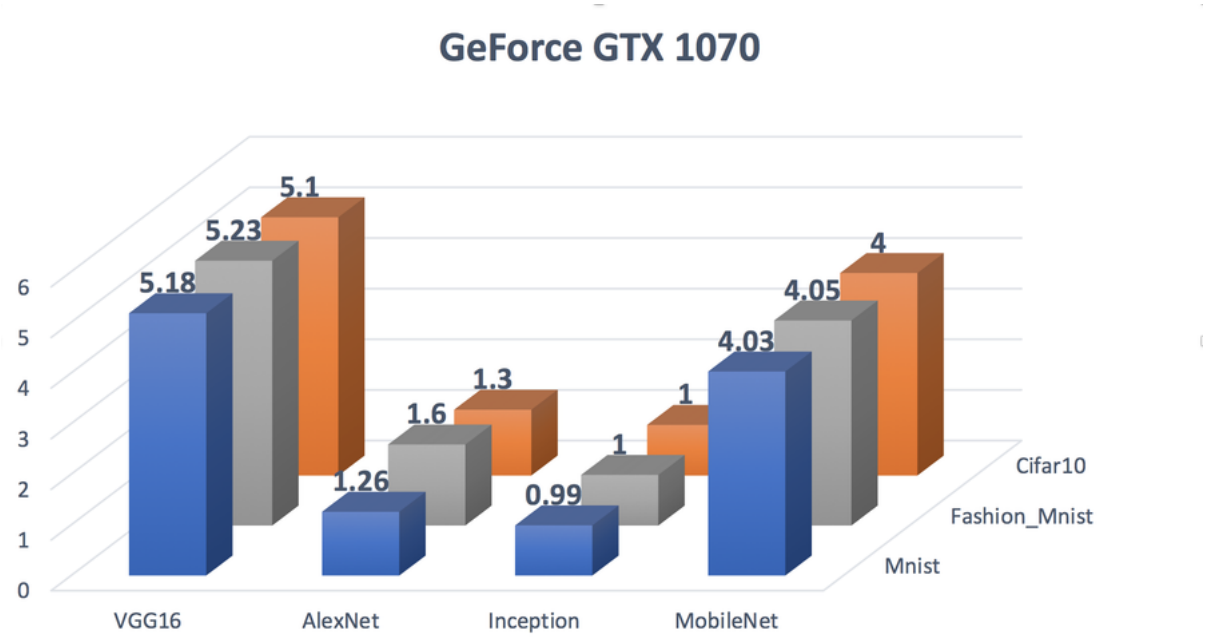
Figura 53 – OpenCL CPU



Fonte: O autor (2019)

A Figura 54 mostra a análise realizada na versão OpenCL GPU, apresentando um desempenho superior sobre as demais implementações. Nesta versão, a arquitetura MobileNet aparece com um tempo maior do que o esperado, por ser uma CNN com menos parâmetros. Fazendo uma análise, notou-se que cerca de 50% do tempo da mobilenet estava sendo consumido pelo kernel de Padding. A MobileNet é uma arquitetura que utiliza muito Padding, e uma grande troca de objetos de memória OpenCL, assim essa análise mostra que uma melhoria deve ser feita quanto ao kernel de Padding.

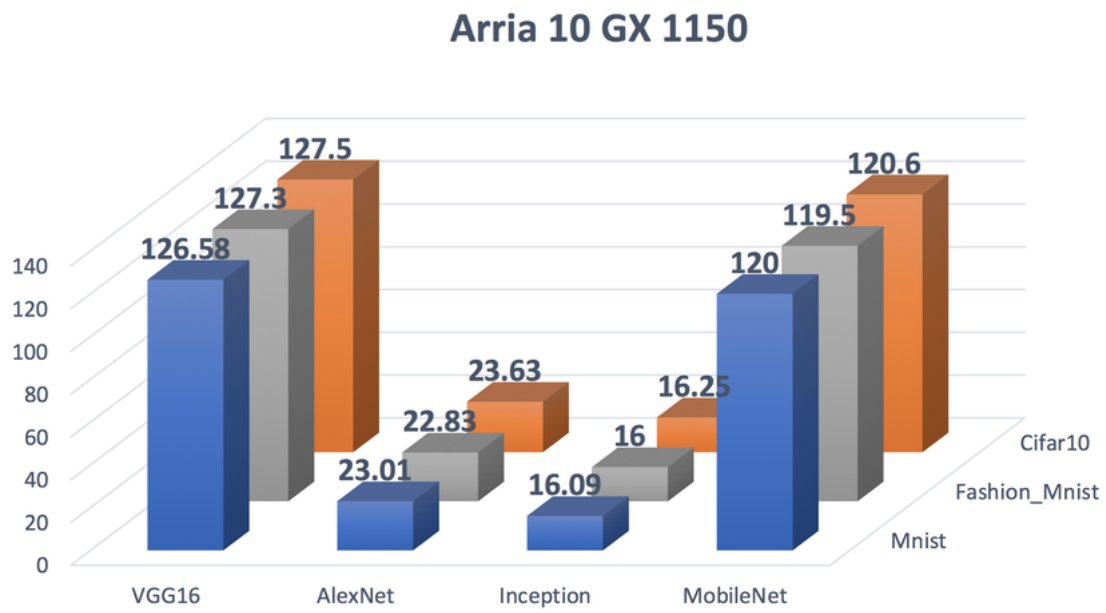
Figura 54 – OpenCL GPU



Fonte: O autor (2019)

A Figura 55 mostra a análise realizada na versão OpenCL FPGA, apresentando um baixo desempenho quando comparada com a versão em GPU. O mesmo comportamento com a MobileNet aconteceu na versão da FPGA, a versão da CPU não sofre esse efeito, pelo fato de todos os objetos de memória estarem em cache. Esta versão, se mostra ineficiente quando comparada com as outras implementações, porém mais estudos podem ser feitos para alcançar um bom desempenho. Uma abordagem possível é realizar um pipeline de Kernels, apresentado no trabalho de (WANG et al., 2017).

Figura 55 – OpenCL FPGA



Fonte: O autor (2019)

Os experimentos mostraram tempos parecidos quando fixamos as arquiteturas de CNN e variamos o dataset, mostrando que a influência no tempo de processamento está na característica da CNN. A VGG apresentou maior tempo de processamento em todos os dispositivos, por apresentar muitas camadas convolucionais com uma grande quantidade de filtros e parâmetros.

5.2.2 Análise de Potência

A análise de potência da biblioteca em diferentes dispositivos é mostrada na Tabela 4. A execução é feita com a arquitetura VGG16, que apresenta maior tempo de processamento.

Ao executar o modelo na CPU, o consumo de energia médio é 114W com 7 GOPS (Giga Operações por Segundo). Ao executar o modelo na GPU, o consumo de energia médio é 150W com 190,15 GOPS. Ao executar o modelo na FPGA, o consumo de energia médio é 42W com 8 GOPS. Assim, a eficiência da biblioteca na GPU é superior a outras implementações, apresentando 1,267 GOPS/W, em comparação com a CPU com 0,061 GOPS/W e a FPGA com 0,19 GOPS/W.

Tabela 4 – Análise de Potência

	CPU	GPU	FPGA
Dispositivo	Intel Core i7-7700HQ	GTX 1070	Arria GX 1150
Frequência	2.8GHz	1506MHz	303MHz
Power (Wats)	114	150	42
Throughput (GOPS)	7	190,15	8
Eficiencia (GOPS/W)	0,061	1,267	0,19

Fonte: O autor (2019)

5.2.3 Utilização de Recursos

O Kernel de convolução é o que mais demanda tempo de processamento na CNN, sendo assim, na versão FPGA utilizamos a flag *__attribute((num_compute_units(N)))*, onde N é a quantidade de vezes que o kernel será replicado na FPGA. Analisamos o kernel de Convolução em 3 versões: 2, 4 e 8 unidades de computação, executando as versões em uma arquitetura VGG e coletando o tempo para se processar uma imagem 128x128. A Tabela 5 mostra a utilização dos recursos de cada versão, mostrando que quanto mais replicamos o kernel, mais recursos serão usados e menos tempo para processar uma imagem será necessário.

Tabela 5 – Utilização de Recursos FPGA

	DSP	Reg	Memória (bits)	BRAM	Lógica	T
CU2	190/1,518 (13%)	313,362	14,964,472/55,562,240 (27%)	1,101 / 2,713 (41%)	176,704/427,200 (41%)	12.13
CU4	288/1,518 (19%)	370,429	20,032,768/55,562,240 (36%)	1,423/2,713 (52%)	200,982/427,200 (47%)	6.16
CU8	480/1,518 (32%)	485,679	29,978,896/55,562,240 (54%)	2,051/2,713 (76%)	251,092/427,200 (59%)	3.71

	DSP	Reg	Memória (bits)	BRAM	Lógica	T

Fonte: O autor (2019)

Mais replicações não foram possíveis graças a limitação de espaço na FPGA, mas essa análise abre espaço para melhorias na implementação dos kernels em FPGA, e para que novos testes possam ser realizados em FPGA's maiores. No cenário atual, se fosse possível replicar o kernel de convolução em até 16 unidades de computação, o desempenho chegaria próximo aos resultados obtidos na GPU.

5.2.4 Comparação com outros Frameworks e Bibliotecas

No final do Capítulo 3 foi apresentada uma comparação da InferenCNN com os trabalhos relacionados na Tabela 1, mostrando que ela é capaz de suportar mais dispositivos de computação e mais arquiteturas de CNN's, não se prendendo à otimizações de apenas uma arquitetura ou dispositivo específico.

Na Tabela 6 é mostrado o comparativo da InferenCNN com o Tensorflow.

A versão em CPU e FPGA apresentam desempenhos inferiores ao Tensorflow em todas as arquiteturas. A versão em GPU apresenta desempenho superior na arquitetura Inception, e desempenho parecido nas outras arquiteturas.

Tabela 6 – Comparativo de Desempenho com o Tensorflow

	AlexNet	VGG16	Inception	MobileNet
Tensorflow GPU	0.96	1.20	1.21	1.37
InferenCNN GPU	1.3	5.1	1	4
InferenCNN FPGA	23,63	127,5	16,25	120,6
InferenCNN CPU	30.29	142.8	19.1	10.24

Fonte: O autor (2019)

A Tabela 7 mostra o comparativo da versão em FPGA com outras implementações em FPGA. A taxa de processamento da InferenCNN foi inferior aos demais aceleradores, sendo impactada por sua implementação flexível, limitando os recursos disponíveis na FPGA.

Tabela 7 – Comparativo com Outros Trabalhos de FPGA's

	Dispositivo	DSP	Precisão	Tamanho do Modelo (GOP)	Taxa de Processamento (GFLOPS)
InferenCNN	Arria 10 GX 1150		32 bits float	30.9	8
(SUDA et al., 2016)	Stratix-V GXA7	-	8-16 bits fixo	30.9	47.5
(DICECCO et al., 2016)	Virtex 7 XC7VX690T	1307	32 bits float	30.9	55
(LI, 2017)	Arria 10AX115	775	32 bits float	30.9	145.67
(WANG et al., 2017)	Stratix-V GXA7	247	8 bits fixo	30.9	121,65

Fonte: O autor (2019)

Na Tabela 8 fazemos um comparativo dos recursos disponíveis de cada trabalho. A InferenCNN é flexível para a execução de outras arquiteturas, dispositivos e inserção de novos kernels. O Tensorflow é flexível para outras arquiteturas, inserção de novos módulos, porém limitado apenas a CPU e GPU Nvidia. O trabalho de (SUDA et al., 2016) é limitado para outras arquiteturas, dispositivos FPGA e inserção de novos módulos. O trabalho de (DICECCO et al., 2016) é flexível para outras arquiteturas e inserção de novos módulos, porém limitado para FPGA. O trabalho de (LI, 2017) é limitado para a outras arquiteturas, dispositivos FPGA's e inserção de novos módulos. O trabalho de (WANG et al., 2017) é flexível para outras arquiteturas, porém limitado para dispositivos FPGA's e inserção de novos kernels.

Tabela 8 – Comparação de Recursos

	Flexibilidade com Outras Arquiteturas	Dispositivos Suportados	Flexibilidades para Inserção de Novos Módulos (Kernels)

	Flexibilidade com Outras Arquiteturas	Dispositivos Suportados	Flexibilidades para Inserção de Novos Módulos (Kernels)
InferenCNN	sim	CPU/GPU /FPGA	sim
TensorFlow	sim	CPU/GPU (Nvidia)	sim
(SUDA et al., 2016)	não	FPGA	não
(DICECCO et al., 2016)	sim	FPGA	sim
(LI, 2017)	não	FPGA	não
(WANG et al., 2017)	sim	FPGA	não

Fonte: O autor (2019)

6 CONCLUSÃO

Neste capítulo, resumimos as principais contribuições deste trabalho de mestrado, e abrimos espaço para trabalhos futuros e possíveis melhorias.

6.1 Principais Contribuições

Neste trabalho discutimos que as Redes Neurais Convolucionais - CNN's têm se mostraram o estado da arte para problemas de classificação de imagens, e apresentamos possíveis dificuldades e limitações em se processar a inferência das arquiteturas de CNN's em plataformas heterogêneas.

O principal objetivo deste trabalho foi implementar uma biblioteca que processe a inferência das arquiteturas de CNN's em dispositivos diferentes e de fabricantes diferentes. Uma biblioteca que seja flexível para ser utilizada em CPU's, GPU's e FPGA's, e para receber redes treinadas em qualquer framework.

A utilização do framework OpenCL permitiu a flexibilidade em dispositivos de computação diferentes. O grafo de fluxo implementado permitiu que cada camada fosse implementada de forma modular, deixando espaço para fácil modificação e possíveis otimizações.

Um parser de pesos foi implementado para que fosse aceito redes treinadas em outros frameworks.

Uma implementação canônica da biblioteca foi feita em C++ para se achar gargalos, pontos de otimização, e para ter um conhecimento claro sobre o funcionamento de cada camada, pois a implementação de cada camada é feita da forma mais natural possível, sem apresentar paralelismo.

A versão em GPU se mostrou com resultados melhores que o Tensorflow na arquitetura Inception, e resultados próximos nas outras arquiteturas.

Também investigamos as limitações sofridas pela versão em FPGA e possíveis soluções para se obter um melhor desempenho. Mesmo diante de um resultado inferior aos demais trabalhos com FPGA, a implementação se mostra promissora por sua flexibilidade com qualquer arquitetura e com a facilidade em se modificar e aceitar otimizações.

E por fim, na próxima seção, apresentamos possibilidades de melhorias e trabalhos futuros a serem desenvolvidos a partir deste trabalho.

6.2 Trabalhos Futuros

Como trabalhos futuros propomos:

- Investigar uma forma diferente de se realizar o kernel de padding
  - Visto que houve uma perda de desempenho na arquitetura MobileNet causado pelo kernel de Padding, propomos um modo diferente de se realizar essa função, sem que haja uma grande troca de objetos de memória.
- Investigar uma forma diferente de se processar os kernels em FPGA's
  - Propomos realizar uma pipeline de kernel, assim como o trabalho de (WANG et al., 2017), para que menos acesso seja feito à memória global, mantendo o controle pelo host, para aceitar diversas arquiteturas.
- Investigar uma forma diferente de kernel de Convolução
  - O kernel de Convolução se apresentou como o gargalo para todas as versões, sofrendo um impacto maior na FPGA. Por isso, propomos um kernel mais simples com uso de instruções vetoriais para se processar o kernel de convolução
- Analisar o processamento em outras FPGA's
  - Vimos que o kernel de convolução não pôde ser replicado por limitação da FPGA utilizada, por isso, propomos a análise em FPGA's maiores.

REFERÊNCIAS

ADAM PASZKE et al. **An open source deep learning platform that provides a seamless path from research prototyping to production deployment**. 2016. PyTorch. Disponível em: https://pytorch.org/. Acesso em: 08/08/2018. Citado 2 vezes nas páginas 14 e 39.

ALEX KRIZHEVSKY et al. ImageNet Classification with Deep Convolutional Neural Networks. In: NIPS PROCEEDINGS, 2012. **Neural Information Processing Systems Conference**. [S.l.], 2012. p. 1 – 9. Citado na página 14.

AMD - Advanced Micro Devices. 1969. Disponível em: https://www.amd.com/en. Acesso em: 08/08/2018. Citado 2 vezes nas páginas 15 e 16.

ANDREW G. HOWARD et al. **MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications**. 2017. ArXiv:1704.04861. Disponível em: https://arxiv.org/abs/1704.04861. Acesso em: 08/08/2018. Citado 2 vezes nas páginas 26 e 34.

CHELLAPILLA, K. et al. High Performance Convolutional Neural Networks for Document Processing. **Tenth International Workshop on Frontiers in Handwriting Recognition**, La Baule, France, 2006. Citado 2 vezes nas páginas 23 e 45.

CHRISTIAN SZEGEDY et al. **Rethinking the Inception Architecture for Computer Vision**. 2015. ArXiv:1512.00567. Disponível em: https://arxiv.org/abs/1512.00567. Acesso em: 08/08/2018. Citado 2 vezes nas páginas 14 e 33.

COMPUTE Library for Deep Neural Networks (clDNN). 2017. Disponível em: https://github.com/intel/clDNN. Citado na página 46.

CUDA - Compute Unified Device Architecture. 2006. Disponível em: https://developer.nvidia.com/cuda-zone. Acesso em: 08/08/2018. Citado 2 vezes nas páginas 39 e 40.

DICECCO, R. et al. Caffeinated FPGAs: FPGA Framework For Convolutional Neural Networks. 2016. Citado 4 vezes nas páginas 14, 47, 80 e 81.

GOODFELLOW, I. et al. **Deep Learning**. [s.n.], 2016. Disponível em: http://www.deeplearningbook.org/. Citado na página 21.

GOOGLE. **An open source machine learning library for research and production**. 2015. TensorFlow. Disponível em: https://www.tensorflow.org/. Acesso em: 08/08/2018. Citado 2 vezes nas páginas 14 e 35.

GOOGLE. **Object Detection**. 2017. Disponível em: https://github.com/tensorflow/models/tree/master/research/object_detection. Citado 2 vezes nas páginas 14 e 23.

HE, K. et al. Deep Residual Learning for Image Recognition. 2016. Citado na página 34.

IMAGENET. 2009. Disponível em: http://www.image-net.org/. Acesso em: 08/08/2018. Citado 2 vezes nas páginas 14 e 23.

INTEL. **Intel vLab Academic Cluster**. 2017a. Disponível em: https://wiki.intel-research.net/Introduction.html. Citado na página 71.

INTEL. **Open Programmable Acceleration Engine (OPAE)**. 2017b. Disponível em: https://01.org/OPAE. Citado na página 71.

INTEL FPGAs and Programmable Devices. 1968. Disponível em: https://www.intel.com/content/www/us/en/products/programmable.html. Acesso em: 08/08/2018. Citado 2 vezes nas páginas 15 e 16.

KAREN SIMONYAN; ANDREW G. HOWARD. **Very Deep Convolutional Networks for Large-Scale Image Recognition**. 2014. ArXiv:1409.1556. Disponível em: https://arxiv.org/abs/1409.1556. Acesso em: 08/08/2018. Citado 2 vezes nas páginas 14 e 56.

KERAS: The Python Deep Learning library. 2015. Disponível em: https://keras.io/. Acesso em: 08/08/2018. Citado na página 37.

KRIZHEVSKY, A. et al. ImageNet Classification with Deep Convolutional Neural Networks. **Advances in Neural Information Processing Systems**, Harrahs and Harveys, v. 25, 2012. Citado 2 vezes nas páginas 23 e 31.

LECUN, Y. et al. Backpropagation applied to handwritten zip code recognition. **Neural computation**, MIT Press, v. 1, n. 4, p. 541 – 551, 1989. Citado na página 22.

LECUN, Y. et al. Gradient-based learning applied to document recognition. **Proceedings of the IEEE**, IEEE, v. 86, n. 11, p. 2278 – 2324, 1998. Citado 2 vezes nas páginas 22 e 31.

LI, H. **Acceleration of Deep Learning on FPGA**. 2017. Dissertação (Master of Applied Science) — University of Windsor. Citado 3 vezes nas páginas 48, 80 e 81.

MCCULLOCH, W. S.; PITTS, W. **A logical calculus of the ideas immanent in nervous activity**. The bulletin of mathematical biophysics, 1943. v. 5. 115 p. (4, v. 5). Disponível em: https://doi.org/10.1007/BF02478259. Citado na página 19.

MICROSOFT. Visual Studio. 1997. Disponível em: https://visualstudio.microsoft.com/. Citado na página 64.

MICROSOFT. **Text Analytics**. 2018. Disponível em: https://azure.microsoft.com/pt-br/services/cognitive-services/text-analytics/. Acesso em: 2018. Citado 2 vezes nas páginas 14 e 23.

NVIDIA Corporation. 1993. Disponível em: https://www.nvidia.com/pt-br/. Acesso em: 08/08/2018. Citado 2 vezes nas páginas 15 e 16.

OH, K.; JUNG, K. GPU implementation of neural networks. **Pattern Recognition**, v. 37, p. 1311 – 1314, 2004. Citado 2 vezes nas páginas 22 e 45.

PERKINS, H. **DeepCL**. 2015. Disponível em: https://github.com/hughperkins/DeepCL. Acesso em: 08/08/2018. Citado na página 45.

RONNEBERGER, O. et al. **U-Net: Convolutional Networks for Biomedical Image Segmentation**. 2015. Disponível em: https://arxiv.org/abs/1505.04597. Citado na página 14.

RUMELHART, D. E. et al. Learning representations by back-propagating errors. 1986. Disponível em: https://www.nature.com/articles/323533a0. Citado na página 21.

SIMONYAN, K.; ZISSERMAN, A. VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION. **ICLR**, 2014. Citado na página 31.

SUDA, N. et al. Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks. **Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ser. FPGA '16**, New York, NY, USA: ACM, p. 16 – 25, 2016. Disponível em: http://doi.acm.org/10.1145/2847263.2847276. Citado 3 vezes nas páginas 46, 80 e 81.

SZE, V. et al. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. 2017. Disponível em: https://arxiv.org/pdf/1703.09039.pdf. Citado 3 vezes nas páginas 14, 18 e 19.

THE KHRONOS GROUP. **The Khronos Group - Connecting Software to Silicon**. 2000. Disponível em: https://www.khronos.org/. Acesso em: 08/08/2018. Citado na página 15.

THE OPENCL. **The OpenCL Specification. Khronos OpenCL Working Group**. 2009. Disponível em: https://www.khronos.org/opencl/. Acesso em: 08/08/2018. Citado 3 vezes nas páginas 15, 16 e 40.

WANG, D. et al. PipeCNN: An OpenCL-Based Open-Source FPGA Accelerator for Convolution Neural Networks. 2017. Citado 5 vezes nas páginas 49, 76, 80, 81 e 83.

YANGQING JIA. **A New Lightweight, Modular, and Scalable Deep Learning Framework**. 2017. Caffe. Disponível em: https://caffe2.ai/. Acesso em: 08/08/2018. Citado 2 vezes nas páginas 14 e 40.