MARK CAPPELLO FERREIRA DE SOUSA

Método para execução de redes neurais convolucionais em FPGA

São Paulo
2019

MARK CAPPELLO FERREIRA DE SOUSA

Método para execução de redes neurais convolucionais em FPGA

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Mestre em Ciências.

Área de concentração: Microeletrônica

Orientador: Prof. Dr. Emílio Del Moral Hernandez, Livre Docente

São Paulo

2019

Este exemplar foi revisado e corrigido em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, _____ de _____ de _____

Assinatura do autor: _____

Assinatura do orientador: _____

Catalogação-na-publicação

Sousa, Mark Cappello Ferreira de
Método para execução de redes neurais convolucionais em FPGA / M. C. F. Sousa -- versão corr. -- São Paulo, 2019.
115 p.

Dissertação (Mestrado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Sistemas Eletrônicos.

1.Redes Neurais Convolucionais 2.AlexNet 3.Reconhecimento Embarcado de Padrões 4.FPGA 5.Reconhecimento de Imagens I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Sistemas Eletrônicos II.t.

AGRADECIMENTOS

Ao Prof. Dr. Emilio Del Moral Hernandez, pela orientação que apoiou o progresso deste trabalho.

Ao Prof. Miguel Angelo de Abreu de Sousa, pela dedicação de tempo em discussões técnicas que foram essenciais para este trabalho, pelas valiosas opiniões e também pelas conversas.

À minha esposa, Thammiris M. El Hajj, pelas discussões, orientações e revisões que contribuíram com a qualidade deste trabalho.

À instituição que fomentou esta pesquisa, o Conselho Nacional de Desenvolvimento Científico e Tecnológico – CNPq.

“Estude e não esmoreça, pois traz erguida a cabeça quem sempre cumpre o dever. A vida é luta e batalha e nela só quem trabalha merece e deve vencer."

(Rossandro Klinjey, de seu avô)

“Rationality of thought imposes a limit on a person’s concept of his relation to the cosmos.”

(John Forbes Nash)

RESUMO

Redes Neurais Convolucionais têm sido utilizadas com sucesso para reconhecimento de padrões em imagens. Porém, o seu alto custo computacional e a grande quantidade de parâmetros envolvidos dificultam a execução em tempo real deste tipo de rede neural artificial em aplicações embarcadas, onde o poder de processamento e a capacidade de armazenamento de dados são restritos.

Este trabalho estudou e desenvolveu um método para execução em tempo real em FPGAs de uma Rede Neural Convolucional treinada, aproveitando o poder de processamento paralelo deste tipo de dispositivo. O foco deste trabalho consistiu na execução das camadas convolucionais, pois estas camadas podem contribuir com até 99% da carga computacional de toda a rede. Nos experimentos, um dispositivo FPGA foi utilizado conjugado com um processador ARM *dual-core* em um mesmo substrato de silício. Apenas o dispositivo FPGA foi utilizado para executar as camadas convolucionais da Rede Neural Convolucional AlexNet.

O método estudado neste trabalho foca na distribuição eficiente dos recursos do FPGA por meio do balanceamento do *pipeline* formado pelas camadas convolucionais, uso de *buffers* para redução e reutilização de memória para armazenamento dos dados intermediários (gerados e consumidos pelas camadas convolucionais) e uso de precisão numérica de 8 bits para armazenamento dos *kernels* e aumento da vazão de leitura dos mesmos.

Com o método desenvolvido, foi possível executar todas as cinco camadas convolucionais da AlexNet em 3,9 ms, com a frequência máxima de operação de 76,9 MHz. Também foi possível armazenar todos os parâmetros das camadas convolucionais na memória interna do FPGA, eliminando possíveis gargalos de acesso à memória externa.

Palavras-chave: Redes Neurais Convolucionais, AlexNet, Sistema-em-um-chip, FPGA, Reconhecimento Embarcado de Padrões, Reconhecimento de Imagens.

ABSTRACT

Convolutional Neural Networks have been used successfully for pattern recognition in images. However, their high computational cost and the large number of parameters involved make it difficult to perform this type of artificial neural network in real time in embedded applications, where the processing power and the data storage capacity are restricted.

This work studied and developed methods for real-time execution in FPGAs of a trained convolutional neural network, taking advantage of the parallel processing power of this type of device. The focus of this work was the execution of convolutional layers, since these layers can contribute up to 99% of the computational load of the entire network. In the experiments, an FPGA device was used in conjunction with a dual-core ARM processor on the same silicon substrate. The FPGA was used to perform convolutional layers of the AlexNet Convolutional Neural Network.

The methods studied in this work focus on the efficient distribution of the FPGA resources through the balancing of the pipeline formed by the convolutional layers, the use of buffers for the reduction and reuse of memory for the storage of intermediate data (generated and consumed by the convolutional layers) and 8 bits for storage of the kernels and increase of the flow of reading of them.

With the developed methods, it was possible to execute all five AlexNet convolutional layers in 3.9 ms with the maximum operating frequency of 76.9 MHz. It was also possible to store all the parameters of the convolutional layers in the internal memory of the FPGA, eliminating possible external access memory bottlenecks.

Keywords: Convolutional Neural Networks, AlexNet, System-on-chip, FPGA, Embedded Pattern Recognition, Image Recognition.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS E ABREVIATURAS

ANN – *Artificial Neural Networks*.

ASIC – Application-Specific Integrated Circuit.

BRAM – Block-RAM.

CNN – *Convolutional Neural Networks*.

CPU – *Central Processing Unit*.

DRAM – *Dynamic Random-Access Memory.*

FPGA – *Field Programmable Gate Array*.

GPP – *General Purpose Processor*.

GPU – *Graphical Processing Unit*.

GPGPU – *General Purpose Graphics Processing Unit.*

ICANN – *International Conference on Artificial Neural Networks*

ILSVRC –*ImageNet Large-Scale Visual Recognition Challenge*.

LRN – *Local Response Normalization*.

LUT – Lookup Table.

HBM – *High Bandwidth Memory.*

MLP – *Multilayer Perceptron*.

RAM – *Random Access Memory*.

ReLU - *Rectified Linear Unit*.

SoC – *System on Chip*.

LISTA DE SÍMBOLOS

$n_{x,y}^{i}$ – Normalização da ativação na posição x, y do mapa de características i.

$a_{x,y}^{i}$ – Ativação na posição x, y, do mapa de características i.

δ – Constante utilizada na normalização Local Response Normalization (LRN).

n – Quantidade de mapas de características envolvidos na normalização LRN.

α – Coeficiente utilizado na normalização LRN.

β – Expoente utilizado na normalização LRN.

$s(c)$ – Função de ativação sigmoide.

$r(c)$ – Função de ativação *Rectified Linear Unit* (ReLU).

$c[i, j, k]$ – Convolução utilizando o campo receptivo na posição i, j e *kernel* k.

$b[k]$ – Viés (do inglês *bias)* utilizado para o *kernel* k.

S – *Stride*, deslocamento entre os campos receptivos.

m – Mapa de características.

w_k – Coeficientes do kernel k.

SUMÁRIO

1. INTRODUÇÃO

O surgimento das Redes Neurais Convolucionais (RNCs) foi inspirado na organização do córtex visual, onde existem grupos de neurônios organizados de forma a identificar determinados padrões. Esta organização neuronal é recorrente na retina e realiza uma forma de pré-processamento ao disparar mediante determinados padrões de estímulos (BENGIO, 2009; SCHMIDHUBER, 2015).

As RNCs incluem uma estrutura específica, parcialmente inspirada no córtex visual, para realizar a extração de características de imagens. Esta estrutura consiste em camadas convolucionais encadeadas, intercaladas por camadas de *pooling*, para realizar a redução da dimensionalidade das características extraídas. A última camada convolucional é conectada a um classificador, que consiste em uma cadeia de camadas de neurônios completamente conectadas entre si.

Um diferencial apresentado pelas RNCs em relação a outros algoritmos de classificação de imagens é que o treinamento das camadas convolucionais faz parte do treinamento de toda a rede. Isso reduz a necessidade de pré-processamento específico para extração de características, embora ainda seja benéfico realizar outros tipos de pré-processamentos para tratamento das imagens, como o processamento realizado para aumento de dados (do inglês, *data augmentation*). O aumento de dados consiste na geração de novas imagens introduzindo pequenas variações nas imagens originais, aumentando artificialmente o conjunto de dados disponível para treinamento. Este crescimento do conjunto de treinamento contribui para a redução do *overfitting* (SIMARD; STEINKRAUS; PLATT, 2003; CIRE et al., 2011). Neste tipo de pré-processamento, é comum a introdução de ruído, translação da imagem em alguma direção e, eventualmente, espelhamento, caso o espelhamento não descaracterize a imagem.

Esta pesquisa investigou um método para executar uma RNC em *Field Programmable Gate Arrays* (FPGA). Algumas das vantagens do uso de um FPGA incluem: arquitetura reprogramável, baixo consumo energético e capacidade de processamento paralelo

(FARABET et al., 2010). Porém, existem alguns desafios em relação ao uso destes dispositivos, quais sejam memória interna limitada e dificuldade de balancear quais tarefas devem ser executadas em paralelo e quais podem ser executadas sequencialmente.

1.1. OBJETIVO

O objetivo geral deste trabalho foi estudar e desenvolver um método para execução de Redes Neurais Convolucionais em FPGAs. O desenvolvimento deste método considerou os seguintes aspectos:

- Restrições impostas pelos recursos disponíveis nos FPGAs.
- Velocidade de processamento.
- Propriedades da execução das RNCs.

Para tratar os problemas apontados pela literatura, retratados na seção 2, esta pesquisa definiu os seguintes objetivos específicos:

- Estudo e desenvolvimento de um método para possibilitar armazenamento interno dos parâmetros das camadas convolucionais e respectiva leitura com alto *throughput*.
- Desenvolvimento de componentes de *hardware* para FPGAs para gerenciamento dos dados intermediários das camadas convolucionais, reutilizando endereços de memória para minimizar a necessidade de armazenamento.
- Distribuição equilibrada de recursos computacionais do FPGA por balanceamento de *pipeline.*
- Adoção de precisão numérica reduzida em relação ao ponto flutuante de 32 bits, tradicionalmente utilizado em plataforma de alto nível para execução de RNCs, sem comprometer o desempenho de classificação.

1.2. JUSTIFICATIVA

A execução das camadas convolucionais das RNCs demandam um grande volume de multiplicações, sendo estas camadas responsáveis por até 99% de toda a carga computacional da RNC (ADHIANTO et al., 2016; MA et al., 2016; SUDA et al., 2016; SZE et al., 2016, 2017; ALOM et al., 2018). Portanto, o poder de processamento paralelo dos FPGAs é um grande atrativo para execução das camadas convolucionais das RNCs.

Segundo (FARABET et al., 2009; ADHIANTO et al., 2016; SHARMA et al., 2016; WANG et al., 2017), o consumo energético dos FPGAs é entre 4,2 a 10 vezes mais eficiente do que as *General Purpose Graphic Processing Units* (GPGPUs), para a execução da mesma tarefa. Desta forma, torna-se atrativa a utilização deste dispositivo, principalmente em sistemas embarcados, onde a disponibilidade de energia é limitada. Isto ocorre, por exemplo, em veículos aéreos não tripulados, aplicações robóticas, carros autônomos, cidades inteligentes e interface cérebro-máquina. Portanto, decidiu-se investigar a viabilidade de execução de uma Rede Neural Convolucional (RNC) consagrada em reconhecimento de imagens, a AlexNet, em um dispositivo FPGA.

1.3. PRINCIPAIS CONTRIBUIÇÕES

A contribuição principal deste trabalho consiste no estudo e desenvolvimento de um método para execução de Redes Neurais Convolucionais em FPGAs. O método estudado neste trabalho contribui com os seguintes temas:

- ***Buffers* para reuso de memória e armazenamento de dados intermediários das camadas convolucionais**. Estes *buffers* procuram minimizar a demanda por memória sem comprometer o desempenho da execução.
- **Balanceamento do *pipeline* formado pelas camadas convolucionais.** Este balanceamento é essencial para otimização do uso dos recursos do FPGA. O

método estudado mostra como identificar os potencias gargalos das camadas convolucionais e como dedicar o poder de paralelismo do FPGA para prevenção destes gargalos.

- **Alocação dos parâmetros das camadas convolucionais em Block-RAM**. A execução das camadas convolucionais depende da leitura recorrente dos parâmetros referentes aos *kernels* utilizados nas convoluções. Consequentemente, para que o desempenho da execução das camadas convolucionais não seja comprometido, os parâmetros destas camadas foram alocados em múltiplos Block-RAMs (um tipo de memória interna do FPGA), possibilitando o paralelismo na leitura, utilizando precisão numérica de 8 bits.

Além da contribuição metodológica, a pesquisa bibliográfica realizada neste trabalho também permitiu contribuições de outras naturezas. Parte da pesquisa bibliográfica realizada neste trabalho procurou identificar a relevância das Redes Neurais Convolucionais para reconhecimento de padrões em imagens e, como resultado, este trabalho também contribui para um breve **panorama da trajetória das Redes Neurais Convolucionais até os dias atuais**.

Este documento também apresenta a **arquitetura de componentes** desenvolvida para a execução das camadas convolucionais em FPGAs, sendo esta uma das contribuições secundárias deste trabalho.

Apesar das execuções de Redes Neurais Convolucionais em processadores de propósito geral utilizarem representação numérica de ponto flutuante de 32 bits, a literatura mostra que a redução para 8 bits não compromete o desempenho de classificação da rede. Portanto, uma das contribuições secundárias deste trabalho consiste **na pesquisa bibliográfica sobre a viabilidade do uso de precisão numérica de 8 bits para armazenamento dos parâmetros das camadas convolucionais.**

A revisão bibliográfica também teve como objetivo o levantamento dos pontos críticos da execução das Redes Neurais Convolucionais em FPGA. Portanto, uma das contribuições aqui apresentadas é a **compilação dos principais desafios da**

execução das Redes Neurais Convolucionais em FPGAs apontados pela literatura.

Finalmente, também como resultado da revisão bibliográfica, este trabalho apresenta quais são as **principais vantagens da adoção de FPGAs para execução de Redes Neurais Convolucionais**.

2. REVISÃO DA LITERATURA

Neste capítulo, além da revisão bibliográfica, também é feita uma introdução às RNCs, apresentando a inspiração biológica que motivou o surgimento das RNCs. Este capítulo apresenta alguns dos desafios da execução das RNCs em FPGAs e algumas das soluções adotadas.

2.1. INTRODUÇÃO ÀS REDES NEURAIS CONVOLUCIONAIS

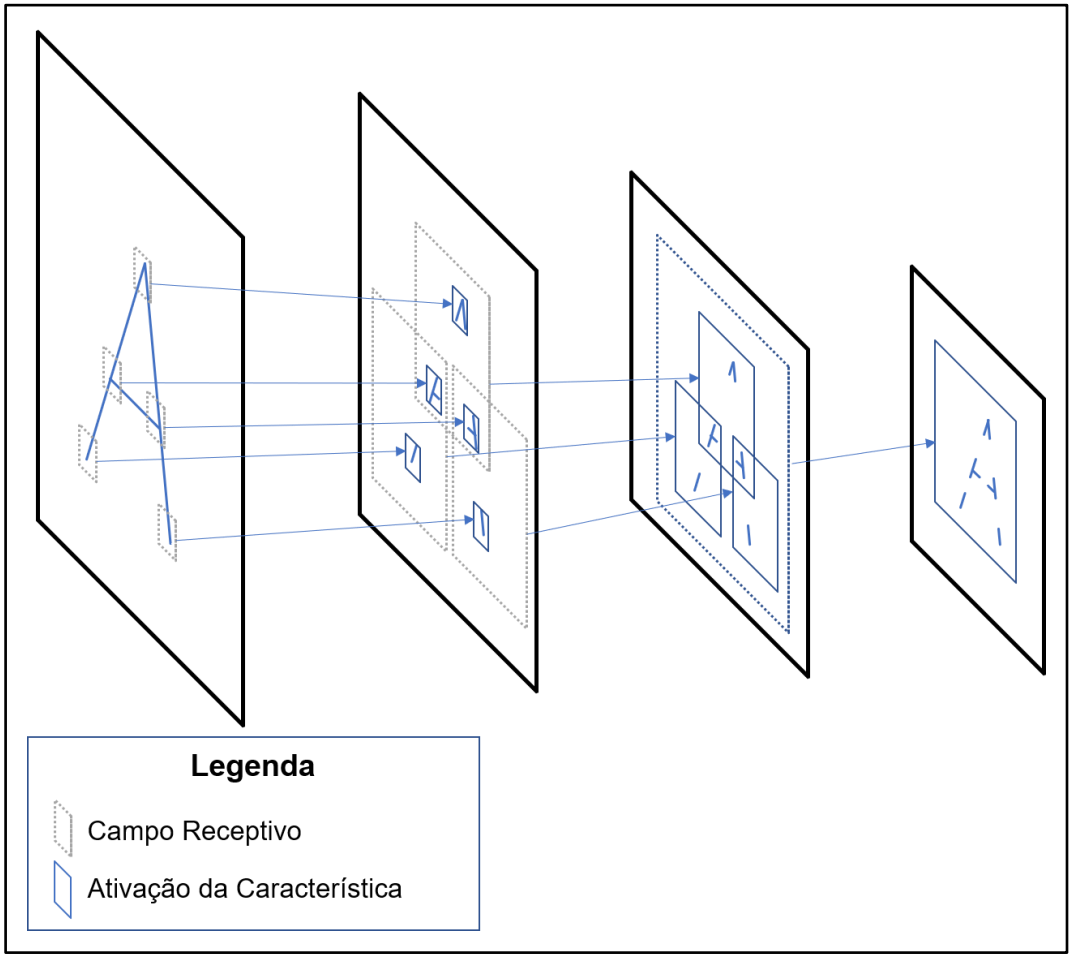
As Redes Neurais Convolucionais (RNCs) surgiram com inspiração na estrutura do córtex visual (BENGIO, 2009; SCHMIDHUBER, 2015), especificamente o modelo proposto por (HUBEL; WIESEL, 1962), o qual apresenta a estrutura do córtex visual do gato. Neste modelo biológico, uma organização recorrente na retina de determinados grupos de neurônios foi apresentada. Estes se excitam sempre que ocorre a apresentação de alguns padrões de estímulos ao seu campo receptivo[1]. Esta organização neuronal, que se repete na retina, serviu de inspiração para os modelos computacionais, que passaram a utilizar compartilhamento de pesos entre neurônios.

O modelo Neocognitron, apresentado por (FUKUSHIMA, 1980) aplica a ideia de neurônios com pesos compartilhados, organizados em camadas hierárquicas, conforme ilustrado na Figura 1. O modelo demonstra que grupos de neurônios com a mesma estrutura e mesmos pesos aumentam a capacidade de invariância à translação, quando conectados a múltiplos campos receptivos das camadas anteriores. Neste modelo, o compartilhamento de pesos entre campos receptivos é semelhante às convoluções utilizadas nas RNCs atuais, embora o termo convolução ainda não tenha sido empregado neste trabalho.

[1] Na retina, o campo receptivo consiste em um conjunto de fotorreceptores conectados a um neurônio que, quando estimulados, estimulam ou inibem a atividade deste neurônio (BEAR; CONNORS; PARADISO, 2016).

Mais tarde, os mesmos princípios de compartilhamento de pesos foram adotados por LeCun (LECUN et al., 1989, 1998), utilizando treinamento baseado em gradiente, inclusive para os pesos das camadas convolucionais, para reconhecimento de caracteres manuscritos. Os autores apontam a importância do compartilhamento de pesos entre neurônios para redução do número de parâmetros livres, aumentado a probabilidade de generalização. O aprendizado dos pesos das camadas convolucionais foi um diferencial valioso introduzido pelas RNCs, pois o aprendizado destes pesos apresentou uma alternativa à tradicional extração não automatizada de características adotada até então.

Figura 1 – Extração de características do modelo Neocognitron.

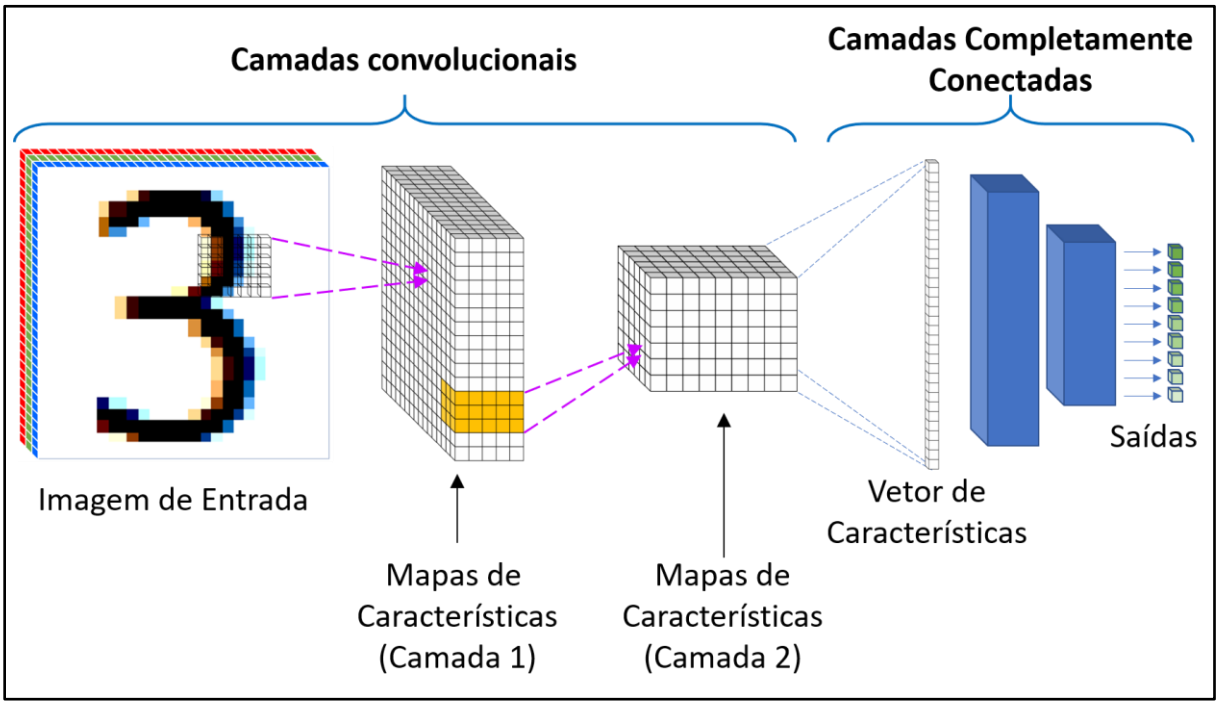


Fonte: Adaptado de (FUKUSHIMA, 1980).

Estes trabalhos foram alguns dos principais precursores das RNCs utilizadas atualmente, mostrando alguns conceitos aplicados nas RNCs modernas. E

contribuindo para a definição das arquiteturas de RNCs utilizadas hoje em dia, com camadas convolucionais para extrair características, intercaladas com camadas de *pooling,* utilizadas para reduzir a dimensionalidade das características extraídas, e camadas completamente conectadas realizando a classificação das características extraídas (SCHMIDHUBER, 2015), conforme ilustrado na Figura 2.

Figura 2 - Exemplificação da estrutura das RNCs utilizando duas camadas convolucionais.

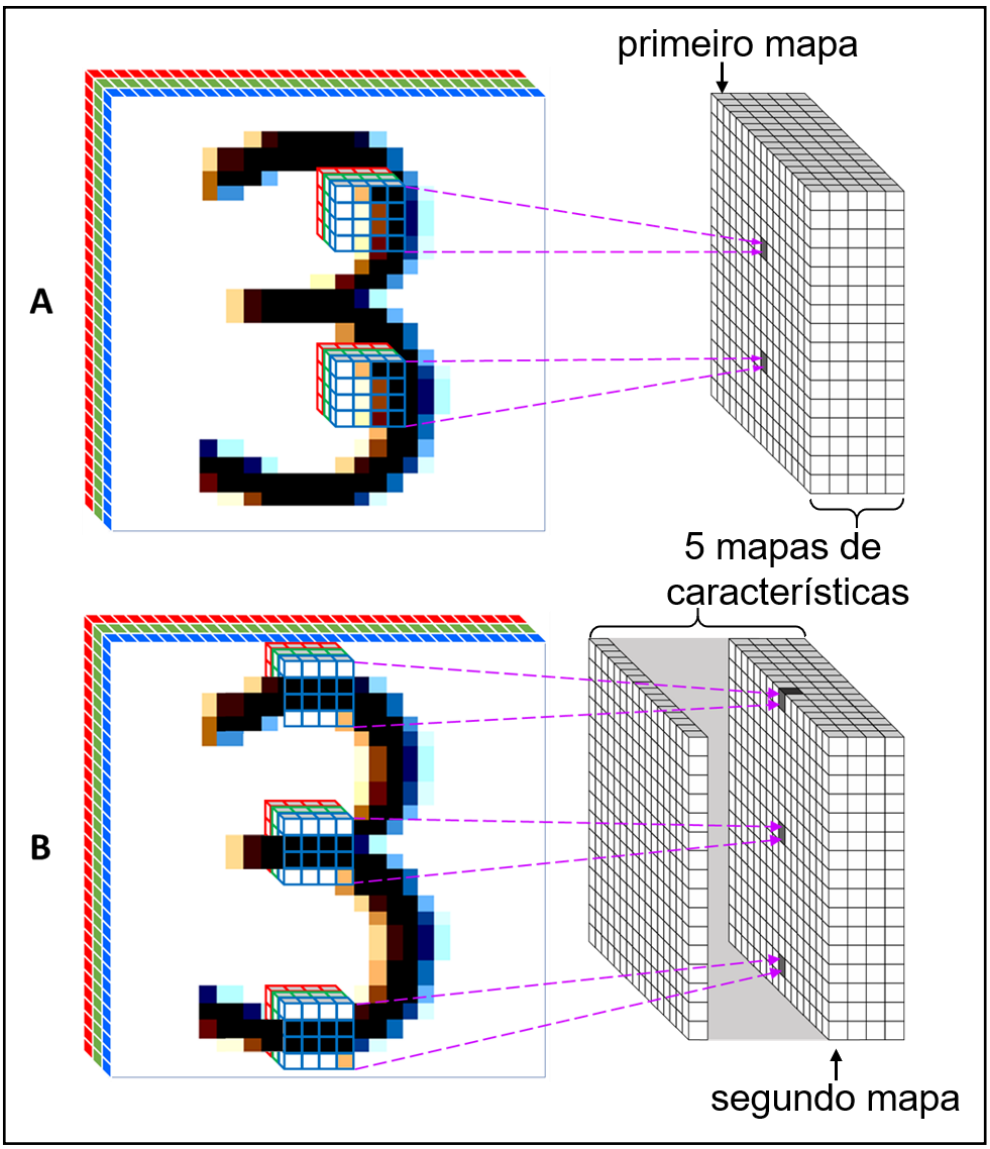


Fonte: Autoria Própria (2019).

Na Figura 2, são ilustrados os mapas de características extraídos por duas camadas convolucionais. Cada **mapa de característica** consiste no conjunto de dados gerados pela extração de um determinado *kernel* em uma camada convolucional. O mapa de características é composto pelos resultados das convoluções após serem submetidos a uma função de ativação, que quantifica a presença da característica em questão no campo receptivo utilizado na convolução. Portanto, pode-se dizer que o mapa de características é formado pelas ativações das convoluções realizadas. A posição de cada ativação em um mapa de característica preserva a equivalência espacial em relação à posição do campo receptivo utilizado na convolução. Consequentemente, dois campos receptivos adjacentes também terão suas respectivas ativações

adjacentes no mapa de características extraído. A preservação desta referência espacial permite o mapeamento das características extraídas em cada camada convolucional, conforme ilustrado pela Figura 3.

Figura 3 – Exemplo de mapas de características com 5 mapas, representando a equivalência espacial de duas ativações de uma determinada característica em **(A)** e de três ativações de outra característica em **(B)**.



Fonte: Autoria própria (2019).

A Figura 3 ilustra uma imagem de entrada, representada por de três mapas de características referentes aos componentes RGB de cada pixel, e também cinco mapas de características extraídos destes mapas de características de entrada;

portanto, são extraídas cinco características, utilizando cinco *kernels* diferentes em uma mesma camada convolucional. Na Figura 3, em (A), são destacadas duas das diversas ativações do primeiro mapa de características para exemplificar o mapeamento das ativações. Note que estas duas ativações, referentes a dois campos receptivos diferentes, possuem um distanciamento entre si que equivale ao distanciamento entre seus respectivos campos receptivos. Em (B), estão destacadas três ativações referentes às ocorrências de outra característica, extraída de outro campo receptivo. Anteriormente, na Figura 2, um vetor de características também está ilustrado, que é resultado da reorganização dos mapas de características da última camada convolucional, ou seja, se trata do rearranjo dos elementos dos mapas da última camada convolucional para formar um vetor. Este vetor é utilizado como entrada do classificador, que consiste em camadas de neurônios completamente conectadas entre si, denominadas camadas completamente conectadas.

Com exceção da primeira camada convolucional, que realiza as convoluções na imagem de entrada, as demais camadas convolucionais realizam convoluções nos mapas de características gerados pela camada anterior. Neste trabalho, o termo mapa de característica também é extrapolado para a imagem de entrada, pois foi interpretado que cada pixel, na representação RGB, carrega as características referentes à quantidade de vermelho, verde e azul daquele ponto na imagem. Da mesma forma, o termo hiperpixel foi adotado para descrever um ponto no mapa de características que contém todas as ativações relativas às convoluções realizadas com um mesmo campo receptivo, ou seja, pode-se interpretar um hiperpixel como um vetor de características extraídas de um mesmo campo receptivo.

Os mapas de características também podem ser submetidos às camadas de *pooling*, que combinam as ativações de uma determinada região que foram obtidas utilizando um mesmo *kernel*. Portanto, o *pooling* combina ativações de uma mesma característica, utilizando normalmente como critério de agrupamento a média dos valores da ativação, valor máximo ou valor mínimo. Além de atuar na redução da dimensionalidade dos dados, as operações de *pooling* também procuram contribuir para invariância à translação, condições de iluminação e ruído (BOUREAU; PONCE; LECUN, 2010).

A RNC AlexNet (KRIZHEVSKY et al., 2012) é uma das RNCs que adota esta estrutura de camadas convolucionais, intercaladas com camadas de *pooling*, e camadas completamente conectadas como classificador em sua arquitetura. Esta RNC foi a solução que obteve o melhor desempenho na edição de 2012 do *benchmark* de reconhecimento de padrões em imagens *ImageNet Large-Scale Visual Recognition Challenge* (ILSVRC) (RUSSAKOVSKY et al., 2015). Esta RNC também obteve uma vantagem de 9,75% em relação ao segundo colocado desta edição, representando ponto de virada em que as RNCs passaram apresentar resultados cada vez melhores, ultrapassando o desempenho humano referenciado por este *benchmark* na edição de 2015. O *benchmark* ILSVRC é realizado em edições anuais desde 2010 e avalia algoritmos de reconhecimento de imagens. Para possibilitar a participação de equipes do setor corporativo que não podem revelar os detalhes das soluções submetidas, além das soluções abertas, este *benchmark* também permite que os participantes submetam soluções fechadas. Os participantes que submetem soluções abertas devem fornecer mais detalhes sobre as soluções adotadas, enquanto os que submetem soluções fechadas oferecem apenas um resumo da solução. Além da divulgação dos resultados do *benchmark*, os participantes são convidados a divulgar suas ideias nos *workshops International Conference on Computer Vision* (ICCV) ou *European Conference on Computer Vision* (ECCV), que ocorrem em anos alternados.

O banco de dados ImageNet, utilizado parcialmente no ILSVRC, conta com mais de 14 milhões de imagens, utilizando aproximadamente 22.000 categorias, sendo atualmente (dezembro de 2018) o maior banco de dados de imagens rotuladas conhecido. O *benchmark* ILSVRC adota um subconjunto das imagens disponíveis no banco de dados ImageNet e utiliza 1,2 milhões de imagens para treinamento, 50.000 para validação e 150.000 para teste.

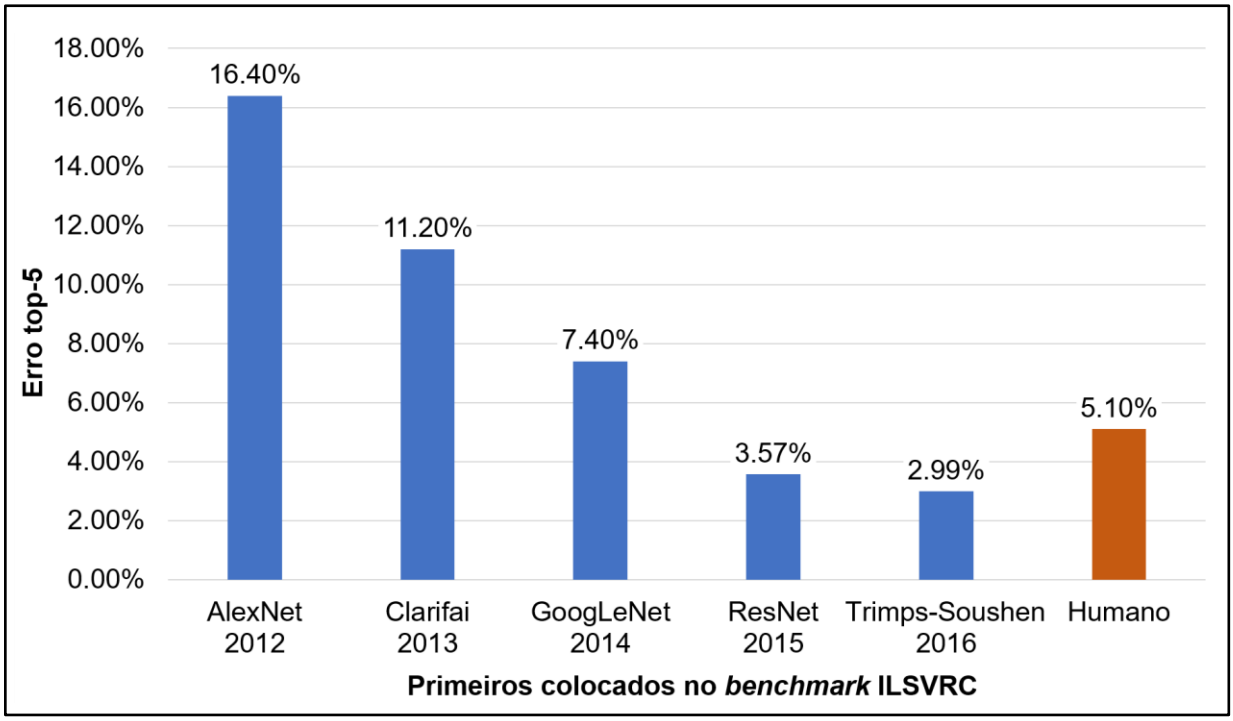
Para medir o desempenho dos algoritmos de classificação de imagens, o *benchmark* ILSVRC utiliza a medida de erro top-5. Esta medida de erro considera cinco respostas indicadas pelo algoritmo de classificação como as classificações mais prováveis para uma determinada imagem. O resultado da classificação é considerado errado caso nenhuma das cinco respostas sejam compatíveis com a categoria rotulada na imagem. Esta medida de erro é justificada pela possibilidade de existirem vários

objetos em muitas das imagens utilizadas, porém, cada imagem recebe apenas um único rótulo referente a um dos objetos contidos.

O *benchmark* ILSVRC mostra a melhora no desempenho das RNCs ao longo das edições. A Figura 4 ilustra o erro top-5 reportado em cada uma das edições do ILSVRC, com as respectivas publicações descritas a seguir:

- A edição de 2012 foi a primeira em que uma RNC obteve o melhor desempenho. A solução que se destacou foi a AlexNet, submetida por dois pesquisadores da universidade de Toronto liderados por Geoffrey Hinton (KRIZHEVSKY et al., 2012).
- Em 2013, a melhor solução foi uma RNC apresentada por Matthew Zeiler, fundador da empresa Clarifai. A solução apresentada foi baseada nas pesquisas desenvolvidas no *Courant Institute of Mathematical Sciences* na *New York University* (ZEILER; TAYLOR; FERGUS, 2011; ZEILER; FERGUS, 2014) e contou com a colaboração de Yann LeCun.
- Em 2014, a melhor solução foi novamente uma RNC, submetida por uma equipe de pesquisadores do Google em parceria com pesquisadores da *University of North Carolina* e *University of Michigan*. A solução submetida utilizou um modelo de RNC denominado *Inception* (SZEGEDY et al., 2015), inspirada no trabalho "*Network In Network*" (NiN) (LIN; CHEN; YAN, 2014).
- Em 2015, o erro top-5 da RNC apresentada foi, pela primeira vez, menor que o erro humano, estimado pelo próprio *benchmark* em 5.1% (RUSSAKOVSKY et al., 2015). A solução foi apresentada por uma equipe de pesquisadores da *Microsoft Research* e utilizou um modelo de RNC denominado *Residual Net* (ResNet), contento 152 camadas (HE et al., 2016).
- Em 2016, a solução que obteve o melhor resultado utilizou uma combinação de diferentes modelos de RNCs, dentre eles, *Inception* e *ResNet*, utilizando a acurácia de cada modelo como fator de ponderação (SZEGEDY et al., 2016).

Figura 4 – Erros top-5 reportados pelas RNCs no *benchmark* ILSVRC.



Fonte: http://www.image-net.org/challenges/LSVRC/

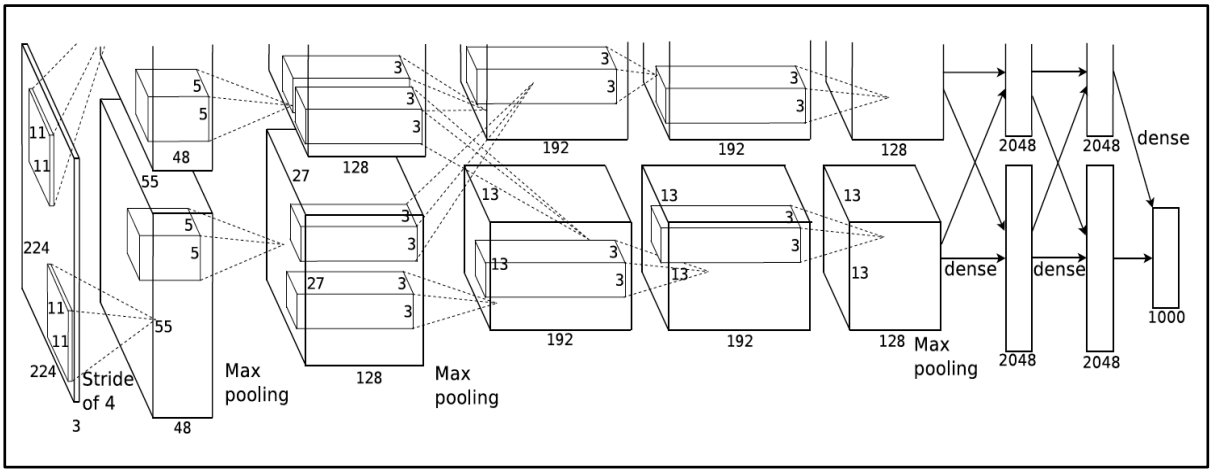
A AlexNet, além de ter sido a primeira RNC a se destacar no *benchmark*, também chamou atenção por apresentar uma diferença de 9,75% no desempenho de classificação em relação ao segundo colocado do *benchmark* ILSVRC de 2012. Desde então, esta arquitetura tem sido replicada em muitos estudos, tendo até o momento (dezembro de 2018), mais de 30 mil citações de acordo com a base bibliográfica *Google Scholar.*

2.2. ARQUITETURA ALEXNET

A arquitetura da AlexNet, ilustrada na Figura 5, consiste em uma sequência de cinco camadas convolucionais, sendo as camadas um, dois e cinco seguidas por uma camada de *pooling*. Os hiperparâmetros das camadas convolucionais e de *pooling* estão descritos na Tabela 1. Após a camada de *pooling* da quinta camada convolucional, existem três camadas completamente conectadas.

O treinamento desta RNC foi executado pelo autor em duas GPUs (KRIZHEVSKY et al., 2012), dividindo o modelo em duas partes iguais, superior e inferior, conforme ilustrado na Figura 5. Aproximadamente 2,3 milhões de parâmetros constituem os kernels das cinco camadas convolucionais, contribuindo com 3,83% do total de parâmetros da RNC, enquanto aproximadamente 58,6 milhões de pesos são utilizados nos neurônios das camadas completamente conectadas, contribuindo com 96,17% do total de parâmetros. Em contrapartida, o custo computacional das camadas convolucionais é de 666 milhões de MACs, cerca de 92% do total, enquanto o custo computacional das camadas completamente conectadas é de 58 milhões de MACs, cerca de 8% do total.

Figura 5 – Arquitetura AlexNet.



Fonte: (KRIZHEVSKY et al., 2012).

Apesar das outras RNCs ilustradas na Figura 4 apresentarem melhor desempenho de classificação que a AlexNet, elas também apresentam maior quantidade de parâmetros, o que torna o desafio da execução em FPGAs ainda maior. Dentre as RNCs da Figura 4, a AlexNet é que apresenta a menor quantidade de parâmetros nas camadas convolucionais, sendo possível armazená-los nos FPGAs disponíveis atualmente. Em segundo lugar está a GoogLeNet, com 6 milhões de parâmetros, 3,7 milhões de parâmetros a mais que a AlexNet. Todas as demais RNCs da Figura 4 apresentam mais de 14 milhões de parâmetros nas camadas convolucionais.

Tabela 1 – Hiperparâmetros das camadas convolucionais da AlexNet.

Camada	Kernels	Campo Receptivo	Mapa de Características	*Stride*
Convolucional 1	96	11 x 11 x 3	(2x) 55 x 55 x48	4
Pooling 1	–	3 x 3 x 1	(2x) 27 x 27 x 48	2
Convolucional 2	256	5 x 5 x 48	(1x) 27 x 27 x 256	1
Pooling 2	–	3 x 3 x 1	(1x) 13 x 13 x 256	2
Convolucional 3	384	3 x 3 x 256	(2x) 13 x 13 x 192	1
Convolucional 4	384	3 x 3 x 192	(2x) 13 x 13 x 192	1
Convolucional 5	256	3 x 3 x 192	(2x) 13 x 13 x 128	1
Pooling 5	–	3 x 3 x 1	(2x) 6 x 6 x 128	2

As camadas convolucionais 1 e 2 da AlexNet utilizam *Local Response Normalization* (LRN) logo após a aplicação da função de ativação, antes das camadas de *pooling*. Esta normalização é realizada em cada grupo de cinco características adjacentes dos hiperpixels dos mapas de características destas camadas. Ou seja, é uma normalização que atua entre características adjacentes em um mesmo hiperpixel, modulando a ativação de uma determinada característica de acordo com a atividade das características adjacentes. Se esta normalização fosse aplicada na imagem de entrada, ela modularia os valores RGB de cada pixel. Portanto, esta normalização não altera o volume de dados do mapa de características. No artigo original (KRIZHEVSKY et al., 2012), esta normalização é apresentada como uma forma de inibição lateral inspirada em neurônios reais e, de acordo com os autores, contribui com 1,4% e 1,2% na diminuição do erro top-1 e top-5, respectivamente. A LRN é dada pela expressão

$$n_{x,y}^{i} = a_{x,y}^{i} \Big/ \left(\delta + \alpha \sum_{j=\max\left(0, i-\frac{n}{2}\right)}^{\min(n-1, i+n-2)} \left(a_{x,y}^{j}\right)^{2} \right)^{\beta}, \qquad (1)$$

onde $n_{x,y}^{i}$ é a ativação normalizada, localizada na posição x, y do mapa de características i, $a_{x,y}^{i}$ é a ativação não normalizada na posição x, y, do mapa de características i e j é o índice que percorre os mapas de características envolvidos na

normalização de $a^i_{x,y}$. Os quatro hiperparâmetros $\delta, n, \alpha\ e\ \beta$ são constantes fornecidas pelo trabalho original como $\delta = 2, n = 5, \alpha = 10^{-4}\ e\ \beta = 0{,}75$.

Embora a LRN tenha contribuído com alguma melhora do desempenho de classificação da AlexNet, atualmente ela não tem sido utiliza em RNCs (CHI et al., 2016; SHAFIEE et al., 2016; SZE et al., 2017). Em alguns casos, a utilização da LRN ainda pode degradar o desempenho de classificação (SIMONYAN; ZISSERMAN, 2015).

2.3. FPGA

O FPGA, acrônimo para *Field Programmable Gate Arrays*, que é traduzido para português como “Arranjo de Portas Programáveis em Campo”, consiste em um conjunto de blocos lógicos que podem ser conectados para formar praticamente qualquer circuito digital ou sistema. Os FPGAs, portanto, não são fabricados para executar alguma arquitetura de processamento específica, sendo necessário descrever esta arquitetura, atendendo às necessidades de alguma aplicação desejada, e então implantar esta arquitetura reprogramando o FPGA. Esta arquitetura é projetada de maneira um pouco semelhante aos projetos de Circuitos Integrados de Aplicação Específica (ASIC), do inglês “*Application Specific Integrated Circuits”*.

Uma das principais vantagens dos FPGAs, em relação aos ASICs, é que eles sempre podem ser reconfigurados para formar diferentes circuitos, enquanto que os ASICs não podem ser alterados após serem fabricados, sendo que este processo de fabricação pode chegar a custar milhares de dólares. Em contrapartida, esta flexibilidade implica maior demanda em área de *chip*, maior *delay* e maior consumo de energia em relação aos ASICs. Estas desvantagens estão relacionadas à malha de roteamento reprogramável dos FPGAs, sendo que, de modo geral, os FPGAs demandam aproximadamente 20 a 35 vezes mais área que um *Application-Specific Integrated Circuit* (ASIC) padrão, com velocidade 4 vezes menor e consumo de energia 10 vezes maior (KUON; TESSIER; ROSE, 2007).

2.4. RNCS E FPGAS

Convencionalmente, as RNCs têm sido executadas em *General Purpose Processors* (GPPs), como CPUs ou *General Purpose Graphics Processing Unit* (GPGPUs). Porém, a execução de RNCs em FPGAs pode oferecer algumas vantagens em relação aos GPPs ou GPGPUs. Uma destas vantagens é a possibilidade de aplicação em sistemas embarcados, onde a disponibilidade de energia é limitada, como ocorre em veículos aéreos não tripulados, aplicações robóticas, carros autônomos, cidades inteligentes e interface cérebro-máquina. Segundo (FARABET et al., 2009; ADHIANTO et al., 2016; SHARMA et al., 2016; WANG et al., 2017), os FPGAs são entre 4,2 a 10 vezes mais eficientes em termos de consumo de energia quando comparados com a execução equivalente da mesma tarefa em GPGPUs.

Além da eficiência energética, o poder de processamento paralelo dos FPGAs também é um atrativo para a execução das RNCs. Segundo (FARABET et al., 2010), tanto os FPGAs quando os ASICs podem ser utilizados para realizar o reconhecimento de imagens em tempo real (pelo menos 30 quadros por segundo). A mesma taxa mínima não pode ser atingida com a execução equivalente em um *laptop Core* 2 *Duo* 2.4 GHz *Apple MacBook PRO*. Em uma outra comparação (ZHAO et al., 2016), o desempenho em FPGA e GPGPU foram respectivamente 4,3 e 4,9 vezes melhores que a execução na plataforma Intel Xeon E5-2680 v3 de 12 *cores* de 2,5 GHz. Neste caso, apesar da execução GPGPU ter obtido um desempenho melhor, o consumo de energia foi 8,6 vezes maior.

Além da conveniência dos FPGAs para sistemas embarcados, eles também foram utilizados com sucesso para aceleração de serviços de *software* em grande escala no centro de processamento de dados do serviço de buscas Bing (PUTNAM et al., 2015). A aceleração das buscas utilizando FPGAs apresentou um aumento de 95% na vazão em comparação à solução de software, enquanto o aumento no consumo de energia foi de 10%. Os autores consideraram inicialmente tanto o uso de FPGAs quanto de

GPGPUS, porém, os requisitos energéticos das GPGPUs foram considerados muito altos para os padrões dos servidores.

Apesar das vantagens citadas, a execução das RNCs em FPGAs também apresenta alguns desafios. Um destes desafios é definir um bom equilíbrio entre processamento paralelo e processamento sequencial durante o projeto dos elementos de *hardware*. Mesmo que o paralelismo contribuia para o bom desempenho das RNCs, quanto maior o nível de paralelismo, maior é a quantidade de recursos de *hardware* utilizados. Por outro lado, o processamento sequencial permite o reaproveitamento de alguns elementos de *hardware* para execução de determinadas tarefas recorrentes, sacrificando parte do desempenho em troca da economia de recursos do FPGA. O bom equilíbrio entre estes dois tipos de processamento deve considerar disponibilidade de recursos do dispositivo FPGA e garantir o desempenho mínimo exigido. No caso de aplicações que demandem o reconhecimento de imagens em tempo real, este desempenho mínimo pode ser 30 quadros por segundo, por exemplo. A literatura indica algumas abordagens para viabilizar a execução das RNCs com o desempenho mínimo exigido pela aplicação em questão, como:

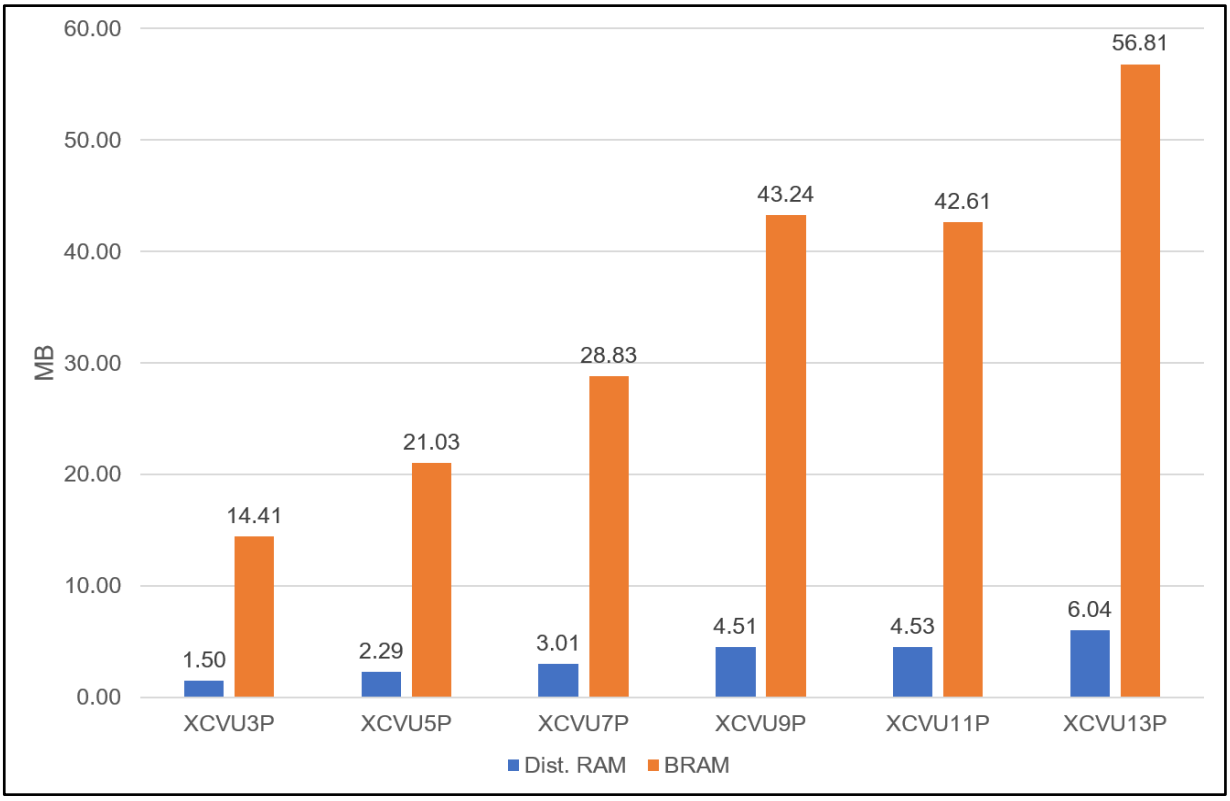
- Uso da reconfiguração dinâmica do FPGA (CHAKRADHAR et al., 2010).
- Conjugação de múltiplos dispositivos FPGA (ZHANG et al., 2016).
- Utilização conjunta de uma CPU e um FPGA para aceleração de parte de uma computação no dispositivo FPGA (SANKARADAS et al., 2009; ADHIANTO et al., 2016; DI CECCO et al., 2017).

Vale notar que as abordagens elencadas não são excludentes, sendo possível adotar alguma combinação entre elas. Em (ZHAO et al., 2016) a conjugação de múltiplos dispositivos FPGAs complementou o uso da reconfiguração dinâmica. Neste caso, o autor aplicou a reconfiguração dinâmica em revezamento, alternando a reconfiguração de alguns dispositivos com a execução de outros.

Outro desafio da execução de RNCs em FPGAs é quantidade de memória interna disponível. Mesmo os dispositivos FPGA de alto desempenho, não possuem memória interna suficiente para armazenamento dos parâmetros das RNCs. Em geral, os FPGAs disponibilizam dois tipos de memórias internas, *distributed* RAM e Block-RAM

(BRAM). A *distributed* RAM leva esse nome por ser o resultado da conjunção das diversas *Lookup Tables* (LUTs) distribuídas ao longo do dispositivo FPGA, enquanto a BRAM consiste em blocos de memória RAM localizados em determinadas regiões do FPGA. As duas principais diferenças entre elas são a sincronicidade de leitura e a capacidade de armazenamento. A *distributed* RAM permite que sejam realizadas leituras assíncronas, enquanto a memória BRAM exige que as leituras sejam realizadas em sincronia com o ciclo de *clock*. Ambas exigem sincronismo com o *clock* durante a escrita. A diferença de capacidade de armazenamento entre as duas memórias interna é de quase 10 vezes, sendo a memória BRAM mais abundante que a *distributed* RAM, conforme ilustrado pela Figura 6, que apresenta a quantidade de memória BRAM e *distributed* RAM de alguns dispositivos da família Virtex UltraScale+ do fabricante Xilinx. Os modelos de prefixo XCVU13P, da família Virtex UltraScale+, contam com 56,81 MB de memória BRAM (XILINX, 2018), com custo estimado entre 39.000 e 88.000 dólares (DIGIKEY, 2018), dependendo da temperatura de funcionamento, número de pinos, frequência de operação, entre outros fatores. Este dispositivo, apesar de ser um dos que disponibilizam a maior quantidade de memória interna atualmente, ainda não é capaz de armazenar todos os parâmetros das RNCs do estado da arte em reconhecimento de imagens.

Figura 6 – Distributed RAM e Block-RAM disponíveis nos FPGAs da linha ultrascale+.



Fonte: Adaptado de (XILINX, 2018).

Para tratar o problema da insuficiência de memória dos FPGAs para execução das RNCs, alguns autores recorrem ao uso de memória externa para o armazenamento dos parâmetros. (FARABET et al., 2009; ZHANG et al., 2015; ESSER et al., 2016; SUDA et al., 2016; WANG et al., 2016). Porém, a grande quantidade de parâmetros transmitidos entre a memória e o FPGA pode representar um desafio, tornando o acesso à memória um potencial gargalo para a execução das RNCs (SUDA et al., 2016; TAPIADOR, 2016; AYDONAT et al., 2017). Uma alternativa para solucionar o problema da insuficiência de memória é a substituição da representação numérica tradicionalmente utilizada em plataforma de alto nível, de ponto flutuante de 32 bits, por uma representação com menor precisão numérica que utiliza menos bits para o armazenamento dos parâmetros. Foi possível encontrar na literatura trabalhos que realizaram o treinamento de RNAs utilizando diferentes representações numéricas, variando entre 8 e 20 bits (SAVICH; MOUSSA; AREIBI, 2007; COURBARIAUX; BENGIO; DAVID, 2014; DETTMERS, 2015; GUPTA et al., 2015; HAN et al., 2015). O treinamento exige uma precisão numérica maior do que a inferência, pois durante o treinamento é necessário acumular pequenas mudanças realizadas nos pesos dos

neurônios (COURBARIAUX; BENGIO; DAVID, 2014). Os autores (FARABET et al., 2010; GOKHALE et al., 2014) utilizaram representação em ponto fixo de 16 bits para realizar a inferência das RNCs, a partir de RNCs treinadas em ponto flutuante de 32 bits, sem degradar o desempenho da classificação. Em (SUDA et al., 2016), foi utilizada representação em ponto fixo de 8 bits da inferência da RNCs, degradando o desempenho da classificação em menos de 1%. Em (VANHOUCKE; SENIOR; MAO, 2011), a representação em 8 bits foi utilizada sem degradação de desempenho de classificação. Neste caso, foi utilizada quantização linear, redimensionando a magnitude máxima dos pesos para o intervalo de -128 e 127.

2.5. FUNÇÃO DE ATIVAÇÃO RELU E O MLP

O teorema da aproximação universal de Cybenko (CYBENKO, 1989) afirma ser possível aproximar arbitrariamente qualquer função contínua por meio da superposição finita de funções sigmoides. Essa superposição de funções sigmoides pode ser realizada por uma rede neural do tipo *Multilayer Perceptron* (MLP) com uma camada oculta.

Uma das principais técnicas para treinamento das redes MLP é o *backpropagation* (HAYKIN, 2008), que atualiza os pesos dos neurônios em função da diferença entre a saída esperada da rede e a saída obtida da rede, sendo, portanto, uma técnica de aprendizado supervisionado. Essa atualização de pesos se inicia na última camada, e é propagada através das camadas anteriores até a camada de entrada. Para o ajuste dos pesos dos neurônios que executam a função sigmoide como função de ativação, é utilizada a derivada parcial da função de calcula o erro da rede em relação ao peso a ser ajustado.

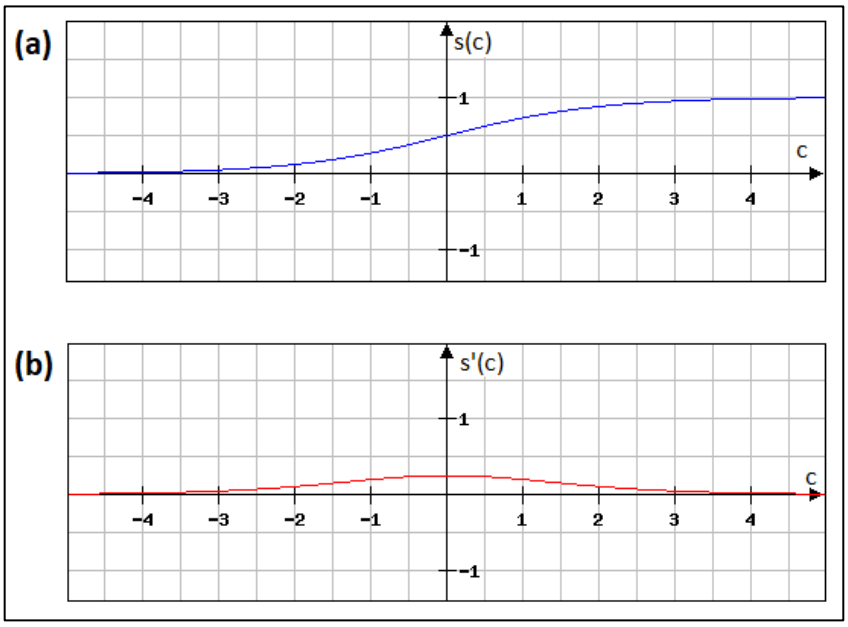
Os gráficos da função sigmoide e sua derivada podem ser consultados na Figura 7 e a função é descrita pela equação

$$s(c) = \frac{1}{1 + e^{-c}} \tag{2}$$

sendo que c é a entrada do neurônio, podendo ser o resultado da extração de uma característica pela convolução de um *kernel* e um campo receptivo, quando a função de ativação é utilizada nas camadas convolucionais, ou a combinação linear das saídas dos neurônios da camada anterior, quando utilizada no classificador. A derivada desta função é dada pela Equação 3.

$$s'(c) = s(c) \cdot \big(1 - s(c)\big)\,. \tag{3}$$

Figura 7 – Função Sigmoide **(a)** e sua respectiva derivada **(b)**.



Fonte: Autoria própria (2019).

A função de ativação *Rectified Linear Unit* (ReLU) tem se mostrado uma alternativa muito eficiente (GLOROT; BORDES; BENGIO, 2011) para a função sigmoide. A função de ativação ReLU pode ser descrita pela Equação 4.

$$r(c) = \max(0, c)\,, \tag{4}$$

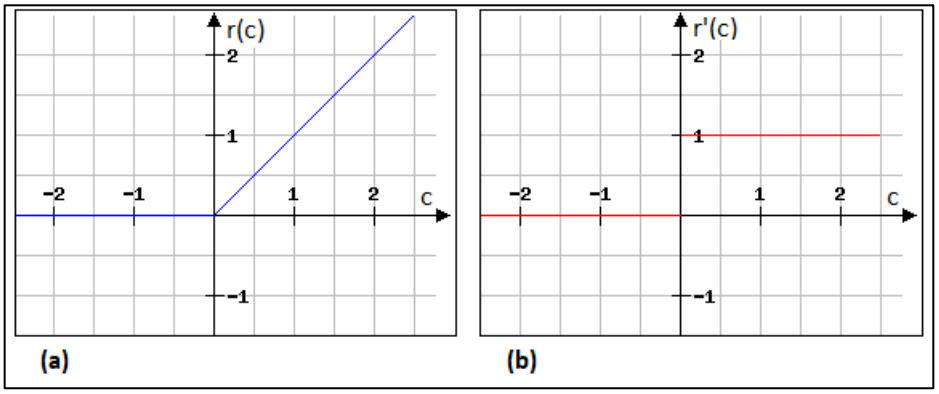
onde $r(c)$ é a ativação do neurônio em questão e c é a entrada do neurônio, podendo ser o resultado da extração de uma característica pela convolução de um *kernel* e um campo receptivo, quando a função de ativação é utilizada nas camadas convolucionais, ou a combinação linear das saídas dos neurônios da camada anterior, quando utilizada no classificador. A função ReLU é consideravelmente mais simples

de ser executada em hardware, demandando menos recursos, e o cálculo de sua derivada, utilizado durante o treinamento para atualização dos pesos da rede, também é relativamente simples, podendo ser representado pela Equação 5.

$$r'(c) = \begin{cases} 0, se\ c < 0 \\ 1, se\ c \geq 0 \end{cases}. \quad (5)$$

Os respectivos gráficos da função ReLU e sua derivada são apresentados pela Figura 8.

Figura 8 – **(a)** Função ReLu e **(b)** derivada da função ReLu.



Fonte: Autoria própria (2019).

Note que a execução da função ReLU em hardware consiste basicamente em verificar se o valor de x é negativo ou positivo, retornando 0 quando for negativo ou, caso contrário, retornando o próprio valor de c. Na execução da derivada retorna 0 para valores negativos, ou 1 caso contrário. A verificação de valores negativos em hardware pode ser feita por meio da verificação da presença do dígito referente ao sinal negativo na representação binária de c.

A AlexNet, assim como outras RNC modernas, utiliza ReLU como função de ativação, o que é muito conveniente para a execução em *hardware*, por ser uma função que demanda menos recursos computacionais que a função sigmoide ou tangente hiperbólica. Esta função de ativação é utilizada tanto nas camadas convolucionais quanto nas camadas do classificador. Este classificador, na comunidade científica de RNCs, é conhecido como **Camadas Completamente Conectadas** (do inglês, *Fully Connected Layers*), pois todos os neurônios de cada camada são conectados aos neurônios das camadas anteriores, assim como nos MLPs.

3. DESENVOLVIMENTO DO MÉTODO PROPOSTO

As primeiras seções deste descrevem o método desenvolvido neste trabalho e as seções posteriores deste capítulo descrevem a arquitetura de componentes desenvolvida para a execução das camadas convolucionais.

Apesar da *Local Response Normalization* (LRN) ter sido utilizada na versão original da AlexNet na primeira e segunda camada convolucional, conforme descrito na seção 2.2, nos experimentos realizados neste trabalho ela não foi utilizada, pois esta normalização aumenta significativamente o custo computacional ao calcular o quadrado de cada ativação dos mapas de características. Portanto, optou-se por sacrificar o ganho de 1,4% e 1,2% do erro top-1 e top-5 (KRIZHEVSKY et al., 2012) em troca da diminuição do uso de recursos de *hardware* do FPGA.

3.1. PARALELISMO EM REDES NEURAIS CONVOLUCIONAIS

Existem algumas possibilidades para exploração do paralelismo nas RNCs, principalmente nas camadas convolucionais, onde está concentrada a maior parte do custo computacional. Aproximadamente 95% do custo computacional da AlexNet está associado à execução das camadas convolucionais (KRIZHEVSKY, 2014), sendo estas camadas responsáveis por apenas 5% do total de parâmetros (FARABET et al., 2010) de toda RNC. Este custo computacional das camadas convolucionais está associado à execução das convoluções, pois esta operação é realizada repetidamente para extrair características para cada um dos *kernels* utilizados, formando os mapas de características. Além da recorrência da operação de convolução, esta operação também apresenta um custo computacional inerentemente alto, pois depende da execução de inúmeras multiplicações. Esta operação pode ser descrita pela Equação 6.

$$c[i,j,k] = b[k] + \sum_{\mathrm{x}=0}^{D-1} \sum_{\mathrm{y}=0}^{D-1} \sum_{z=0}^{F-1} m[S \cdot i + \mathrm{x}, S \cdot j + \mathrm{y}, \mathrm{z}] \cdot w_k[x, \mathrm{y}, \mathrm{z}] \qquad (6)$$

sendo $c[i,j,k]$ uma palavra computada, localizada na posição i,j no plano referente à característica k do mapa de características, $b[k]$ o viés (do inglês *bias*) referente à característica k, D as dimensões do *kernel*, F a quantidade de características no mapa de características de entrada, m o mapa de características de entrada de onde os campos receptivos são obtidos, S o *stride*, $m[S \cdot i + x, S \cdot j + y, z]$ o campo receptivo e w_k os elementos do *kernel* utilizado para extrair a característica k. O mapa de características de uma determinada camada convolucional é formado pelo agrupamento destas características extraídas pelas convoluções e submetidas a uma função de ativação, que no caso da AlexNet é a função ReLu (4), apresentada na seção 2.5. Este agrupamento preserva a localização relativa do campo receptivo utilizado para a extração das características, por isso utiliza-se o termo “mapa de características”.

Sendo a convolução uma operação altamente recorrente e computacionalmente custosa, é de se esperar que a execução paralela das multiplicações e somas desta operação tenha um impacto significativo no desempenho da execução das camadas convolucionais. Portanto, é relevante analisar os resultados da aplicação do paralelismo do FPGA para execução das operações realizadas nas convoluções.

Além da possibilidade de paralelizar as operações realizadas em cada convolução, também é possível paralelizar as convoluções executadas nas diferentes camadas. Isso é possível porque, a partir do momento em que uma determinada camada tenha extraído características suficientes para formar um campo receptivo, é possível utilizar este campo receptivo para extração de características da camada convolucional consecutiva. Portanto, é formado um *pipeline* ao longo das camadas convolucionais, com cada camada consumindo dados da camada anterior e gerando dados para a camada seguinte.

Neste trabalho, estas duas possibilidades de paralelismo são exploradas, executando as diversas multiplicações e somas de cada convolução e as convoluções das cinco camadas convolucionais em paralelo. Considerando que cada convolução executa uma quantidade significativa de multiplicações e que os dispositivos FPGAs possuem capacidade de paralelismo finita, devemos aplicar este poder de processamento paralelo eficientemente entre as camadas. Para isto, foi realizado o balanceamento

do *pipeline* para evitar desperdício de recursos do FPGA e minimizar o tempo de execução das camadas convolucionais.

3.2. BALANCEAMENTO DE PIPELINE - PLANO DE DISTRIBUIÇÃO DE RECURSOS DE HARDWARE

Esta seção mostra como foi realizado o balanceamento do *pipeline* formado pelas camadas convolucionais e como este balanceamento influencia na distribuição de recursos do FPGA. Esta seção foi o foco principal do trabalho apresentado no congresso 27th *International Conference on Artificial Neural Networks* (ICANN) (SOUSA; MIGUEL ANGELO DE ABREU DE SOUSA; EMÍLIO DEL-MORAL-HERNANDEZ, 2018), com o resumo apresentado no Apêndice B – Resumo do Artigo Publicado.

Para acelerar a execução da RNC, este trabalho focou na extração de características executada nas camadas convolucionais, pois, dependendo da arquitetura de RNC utilizada, esta operação concentra entre 90% e 99% do custo computacional das RNCs (FARABET et al., 2010; SZE et al., 2016, 2017). A Tabela 2 mostra a quantidade de convoluções necessárias para a extração de característica em cada camada convolucional da AlexNet. A quantidade de convoluções de cada camada equivale ao produto da quantidade de campos receptivos e da quantidade de *kernels* utilizados em cada campo receptivo. A primeira camada convolucional é a que executa a maior quantidade de convoluções. A última linha da Tabela 2 compara a quantidade de convoluções executadas na primeira camada com a quantidade de convoluções executadas nas demais camadas. Para manter a consistência da comparação, foi considerado que a primeira camada convolucional executa 100% das convoluções dela mesma. Esta comparação mostra que a quantidade de convoluções executadas na segunda camada equivale a 64,26% das convoluções realizadas na primeira camada, enquanto a terceira e quarta camada executam 22,35% das convoluções executadas na primeira camada. Já a quinta camada, executa apenas 14,9% das convoluções realizadas na primeira camada. É importante ressaltar que cada camada

convolucional depende dos dados originados pelas convoluções executas na camada anterior. Portanto, para aumentar o *throughput* das camadas convolucionais é importante acelerar principalmente a execução das convoluções das primeiras camadas, que executam maior número de convoluções e geram dados para as camadas seguintes.

Tabela 2 – Convoluções executadas em cada camada convolucional da AlexNet.

Camada Convolucional	1	2	3	4	5
Campos Receptivos	3.025	729	169	169	169
Kernels	96	256	384	384	256
Convoluções	290.400	186.624	64.896	64.896	43.264
Comparação das Convoluções	100,00%	64,26%	22,35%	22,35%	14,90%

A Tabela 2 também apresenta uma perspectiva do *pipeline* formado pelas cinco camadas convolucionais, sendo este *pipeline* iniciado com a primeira convolução da primeira camada e finalizado com a última convolução da última camada. O fluxo de dados deste *pipeline* corresponde aos resultados obtidos pela execução de cada convolução, gerando dados que são agregados para formar o mapa de características utilizado pela camada convolucional seguinte. Considerando que a geração e o consumo de dados em uma camada convolucional é proporcional à quantidade de convoluções executadas nesta camada, o *pipeline* é balanceado controlando a quantidade de ciclos de *clock* necessários para executar cada convolução. A quantidade de ciclos de *clock* de cada convolução é consequência da quantidade de multiplicações executadas em cada ciclo de *clock*. Se todas as multiplicações da convolução forem executadas em paralelo, será necessário apenas 1 ciclo de *clock* para o cálculo da convolução. A redução da quantidade de multiplicações em paralelo implica maior quantidade de ciclos de *clock* para execução da convolução; porém, menos recursos de *hardware* são utilizados, pois os mesmos multiplicadores são reaproveitados nos demais ciclos de *clock*. Portanto, o nível de paralelismo de cada convolução é determinado durante o planejamento do componente que executa as convoluções.

Para acelerar a computação da convolução, foi utilizada a técnica de desenrolamento de laço (do inglês, *loop unrolling*) nas três somatórias descritas na equação (6). Esta técnica consiste na transformação de um laço para que mais de uma iteração seja executada simultaneamente. Esta técnica é aplicável em laços cujas iterações não sejam dependentes das iterações anteriores, como ocorre no cálculo da convolução (ZHANG et al., 2015; MA et al., 2016, 2017). O fator de desenrolamento do loop determina a quantidade de iterações executadas simultaneamente. Um fator de desenrolamento dois implica uma transformação do laço para que duas iterações sejam executadas simultaneamente, reduzindo a quantidade de iterações pela metade. O maior fator de desenrolamento possível é a própria quantidade de iterações do laço, o que implicaria a execução simultânea de todas as iterações do laço, portanto, o laço seria totalmente desenrolado.

Para a execução da primeira camada convolucional, adotou-se arbitrariamente 1 ciclo de *clock* para a execução de cada convolução desta camada. Consequentemente, todo os três laços da equação (6) devem ser completamente desenrolados para que todas as multiplicações descritas na equação sejam executadas em paralelo. Portanto, para executar as 290.400 convoluções na primeira camada, são necessários pelo menos 290.400 ciclos de *clock*. Além dos ciclos de *clock* necessários para as operações de convolução, também são necessários outros ciclos para carregamento dos dados necessários para as convoluções.

Para validar a escolha arbitrária de um ciclo de *clock* por convolução, foi utilizada a ferramenta de síntese de *hardware* para analisar a quantidade de recursos necessária para este nível de paralelismo. A ferramenta mostrou que, para o dispositivo FPGA utilizado, menos de 20% dos recursos foram comprometidos. Isso mostra que não é necessário reduzir o paralelismo, pois além dos recursos dedicados para a primeira camada, ainda resta uma quantidade de recursos que viabiliza a continuidade do projeto.

Considerando que a primeira camada convolucional é a camada que executa o maior número de convoluções, a aceleração das demais camadas deve levar em consideração a quantidade de ciclos de *clock* desta camada para determinar o fator de desenrolamento dos laços. Qualquer camada que excedesse 290.400 ciclos de

clock da primeira camada, se tornaria o gargalo de todo o *pipeline* formado pelas camadas convolucionais, comprometendo o desempenho da execução da rede. Por outro lado, qualquer camada que execute todas as convoluções em uma quantidade muito inferior de ciclos de *clock* estaria desperdiçando recursos do FPGA realizando operações rapidamente e depois aguardando a geração de dados das camadas anteriores. Portanto, o bom balanceamento do *pipeline* deve evitar que existam camadas degradando o *throughput* geral da rede e evitar o consumo de recursos de *hardware* que seriam subutilizados aguardando a geração de dados das camadas anteriores.

Apesar da operação de adição também ser amplamente utilizada nas convoluções, as considerações são feitas em relação às multiplicações porque os multiplicadores demandam uma quantidade significativamente maior de recursos para sua execução em FPGA (ARIF; LAL, 2015). Além disso, a quantidade de adições tem uma relação direta com a quantidade de multiplicações, sendo que, ao reduzir a quantidade de multiplicações executas por ciclo de *clock*, também se reduz a quantidade de adições.

Para executar as convoluções da segunda camada convolucional, também foi adotado um ciclo de *clock* por convolução, demandando 186.624 ciclos de *clock* para executar todas as convoluções da segunda camada. Apesar desta quantidade de ciclos de *clock* ser relativamente pequena em comparação com os ciclos necessários para a execução da primeira camada, a adoção de mais de um ciclo compromete o desempenho das camadas convolucionais. Mesmo a segunda camada executando menos convoluções que a primeira, ela ainda executa mais de 50% das convoluções executadas na primeira; portanto, ainda não é possível adotar mais de um ciclo de *clock* por convolução, pois seriam necessários 373.284 ciclos de *clock* para executar todas as 186.624 convoluções desta camada. Este valor excede os ciclos de *clock* necessários para a execução da primeira camada em 28,6%, tornando esta camada o novo gargalo do *pipeline* e comprometendo o *throughput* das camadas convolucionais.

Para a execução da terceira e quarta camada convolucional, foram feitas considerações semelhantes às da segunda camada convolucional. Tanto a terceira quanto a quarta camada convolucional executam um total de 64.896 convoluções

cada, equivalente a 22,35% das convoluções da primeira camada. Para estas camadas, a adoção de quatro ciclos de *clock* por convolução resulta em 259.584 ciclos de *clock* necessários para a execução de todas as convoluções. Caso tivessem sido adotados cinco ciclos de *clock* por convolução, seriam necessários 324.480 para a execução de todas as convoluções destas camadas, o que excederia em 11% a quantidade de *clocks* necessários para executar a primeira camada convolucional. Só seria possível a adoção de cinco ciclos se estas camadas executassem menos de 20% das convoluções da primeira camada.

Para a última camada, aplicando o mesmo raciocínio das camadas anteriores, pode-se concluir que utilizando seis ciclos de *clock* por convolução, todas as 43.264 convoluções desta camada podem ser executas em 259.584 *clocks*. Não seria possível adotar sete ciclos de *clock* por convolução nesta camada sem exceder a quantidade de *clocks* necessária para a execução da primeira.

A Tabela 3 sumariza, de acordo com a camada, a quantidade de ciclos *clocks* necessários para execução de cada convolução, assim como a quantidade total de *clocks* necessários para a execução de todas as convoluções da camada. Na Tabela 3, a quantidade de convoluções e a comparação das convoluções foram obtidas da Tabela 2. Na Tabela 3, o tamanho do *kernel* discrimina a quantidade de coeficientes que compõem os *kernels* de cada camada; portanto, equivale às multiplicações realizadas entre cada coeficiente e cada elemento do campo receptivo para o cálculo da convolução. A linha com os ciclos de *clock* por convolução é o resultado do balanceamento do *pipeline*, descrito anteriormente nesta seção. A quantidade de multiplicações por ciclo de *clock* consiste na divisão do total de multiplicações executadas por convolução pela quantidade de ciclos de *clock* necessários para cada convolução. A última linha, ciclos de *clock* por camada, descreve a quantidade de ciclos de *clock* necessários para executar todas as convoluções da camada, sendo este valor o produto entre o total de convoluções da camada e a quantidade de *clocks* por convolução.

Tabela 3 – Ciclos de *Clock* para execução de cada convolução.

Camada Convolucional	1	2	3	4	5
Convoluções	290.400	186.624	64.896	64.896	43.264
Comparação das Convoluções	100,00%	64,26%	22,35%	22,35%	14,90%
Tamanho do Kernel (multiplicações)	363	1.200	2.304	1.728	1.728
Ciclos de *Clock* por Convolução	1	1	4	4	6
Multiplicações por Ciclo de *Clock*	363	1.200	576	432	246
Ciclos de *Clock* por Camada	290.400	186.624	259.584	259.584	259.584

A quantidade de multiplicações por ciclo de *clock* está diretamente ligada ao consumo de recursos do FPGA, pois quanto mais multiplicações são executadas em paralelo, mais multiplicadores são necessários, consequentemente, mais recursos do FPGA são utilizados. Com isso, a quebra da execução da convolução em vários ciclos de *clock* é capaz de poupar recursos do FPGA. Na terceira e quarta camada convolucional, a utilização de 4 ciclos de *clock* por convolução reduz a quantidade de multiplicadores em 75%, pois utiliza ¼ dos multiplicadores necessários em comparação com a adoção de 1 ciclo de *clock* por convolução. Com 4 ciclos de *clock* por convolução, apenas ¼ das multiplicações são executadas em cada ciclo de *clock*, reaproveitando os multiplicadores em cada ciclo de *clock*. Da mesma forma, na terceira camada, os multiplicadores são reduzidos em 83,3% em comparação com a adoção de 1 ciclo de *clock* por convolução, pois os multiplicadores são reaproveitados nos 6 ciclos de *clock* utilizados para convolução nesta camada.

É importante notar que, apesar das três últimas camadas executarem menos convoluções, cada convolução entre um *kernel* e um campo receptivo destas camadas demanda mais multiplicações, pois utilizam *kernels* maiores, portanto, com mais coeficientes. Em cada convolução, a terceira camada convolucional realiza 6,35 vezes mais multiplicações que a primeira e 1,92 vezes mais multiplicações que a segunda. A quarta e quinta camadas convolucionais realizam, cada uma, 4,76 vezes mais multiplicações que a primeira camada e 1,44 vezes mais multiplicações que a segunda em cada convolução. Consequentemente, o balanceamento do *pipeline* tem papel fundamental na distribuição de recursos de *hardware* entre as camadas

convolucionais, podendo ser o fator diferencial que viabiliza a execução das mesmas em *hardware*.

3.3. PRECISÃO NUMÉRICA ADOTADA

Aproximadamente 60 milhões de parâmetros são utilizados na arquitetura AlexNet (KRIZHEVSKY et al., 2012), sendo a maior parte destes parâmetros, aproximadamente 58,6 milhões, correspondente às camadas completamente conectadas, enquanto os 2,3 milhões restantes pertencem às camadas convolucionais. O armazenamento destes parâmetros das camadas convolucionais utilizando representação de 32 bits demandaria 9,2 MB de memória. Porém, reduzindo a representação numérica para 8 bits, a necessidade de memória para armazenamento dos parâmetros das camadas convolucionais é reduzida para 2,3 MB. Esta redução de 9,2 para 2,3 MB de dados viabiliza o armazenamento de todos os parâmetros das camadas convolucionais na memória interna do FPGA, evitando o acesso a memórias externas. Embora a quantidade de dados seja relativamente pequena para o armazenamento em memórias externas, a alta recorrência de leitura dos *kernels* para o cômputo das convoluções faz com que o acesso à memória externa se torne um fator limitante para a execução da RNC (SUDA et al., 2016; TAPIADOR, 2016; AYDONAT et al., 2017).

Alguns autores foram capazes de realizar o treinamento de RNCs utilizando diferentes representações numéricas, variando entre 8 e 20 bits (SAVICH; MOUSSA; AREIBI, 2007; DETTMERS, 2015; GUPTA et al., 2015; HAN et al., 2015). Porém, redes que já estejam treinadas, permitem maior redução da precisão numérica. Isso porque, durante o treinamento, existe a necessidade do acúmulo de pequenas alterações realizadas nos pesos da rede (COURBARIAUX; BENGIO; DAVID, 2014). Em (FARABET et al., 2010; GOKHALE et al., 2014), o treinamento da RNC foi realizado utilizando representação numérica de 32 bits com ponto flutuante, enquanto a execução da rede treinada utilizou 16 bits com ponto fixo. Em outro trabalho (SUDA et al., 2016), a rede treinada utilizando ponto flutuante de 32 bits foi executa com ponto fixo de 8 bits, comprometendo o desempenho da classificação em 1,41% na medida

top-1 e em menos de 1% na medida top-5. Neste trabalho, assim como em (VANHOUCKE; SENIOR; MAO, 2011), foi utilizada representação numérica de 8 bits em ponto fixo e a magnitude máxima foi normalizada para variar no intervalo -128 e 127. Além do trabalho de Vanhoucke, esta mesma representação numérica e quantização também é utilizada pelo *framework* TensorFlow em sua versão Lite, voltada para soluções *mobile*, para reduzir a necessidade de armazenamento e acesso à memória após o treinamento (“Post-training quantization,” 2018).

Além da redução da necessidade de armazenamento, a precisão numérica de 8 bits também possibilita que os componentes de *hardware* projetados demandem menos recursos do FPGA em comparação com precisões numéricas com mais de 8 bits. Como consequência, o consumo energético também é reduzido em comparação aos componentes de *hardware* projetados para precisões numéricas com mais de 8 bits.

3.4. ARMAZENAMENTO DOS *KERNELS* EM BRAMS

Considerando a grande quantidade de parâmetros envolvidos nas camadas convolucionais e a diferença entre a capacidade de armazenamento das memórias internas BRAM e *distributed* RAM apontadas na seção 2.2, a memória BRAM é utilizada para armazenamento dos parâmetros por apresentar maior capacidade de armazenamento, aproximadamente 10 vezes maior que a *distributed* RAM. Para armazenamento dos 2,3 milhões de parâmetros das camadas convolucionais da AlexNet, utilizando representação numérica de 8 bits, são necessários 2,3 MB de memória para armazenamento de todos os *kernels*. Atualmente, existem alguns dispositivos FPGAs que apresentam esta capacidade de armazenamento utilizando apenas da memória interna. Um destes dispositivos é o modelo XC7Z100, do fabricante Xilinx, que possui 755 blocos de memória BRAMs com capacidade de 36 kilobits (Kb) cada um, totalizando 3,31 MB de memória interna BRAM.

O armazenamento dos *kernels* em BRAM é conveniente por permitir a leitura dos *kernels* com alto *throughput*, visto que, como reportado por alguns autores (SUDA et al., 2016; TAPIADOR, 2016; AYDONAT et al., 2017), a sobrecarga do acesso

recorrente às memórias externas, apesar de aumentar a capacidade de armazenamento, pode limitar o desempenho da execução das RNCs. A recorrência do acesso à memória se deve à quantidade de *kernels* utilizados nas convoluções com cada campo receptivo. Ainda que a quantidade de parâmetros das camadas convolucionais represente apenas 3.83% do total de parâmetros da RNC, estes parâmetros são lidos diversas vezes para execução das camadas convolucionais, pois, para cada campo receptivo de uma camada convolucional, todos os *kernels* daquela camada devem ser carregados da memória para o cômputo das convoluções. Portanto, os *kernels* devem ser armazenados na memória BRAM considerando o *throughput* adequado para a taxa de execução das convoluções de cada camada convolucional.

O armazenamento dos *kernels* nas memórias BRAM é feito com a alocação dos dados nos diversos blocos de memória. As memórias internas BRAM são compostas por uma série de blocos individuais com capacidade de armazenamento de 36 Kb cada, contanto os bits de paridade, que são constituídos de duas memórias RAM independentes de 18 Kb, também incluindo os bits de paridade. Isso faz com que seja possível configurar um bloco BRAM como uma memória de 36 Kb ou duas memórias de 18 Kb independentes. Também é possível configurar a porta de acesso para manipulação de diferentes profundidades e larguras de memória, fazendo com que a memória tenha mais endereços armazenando menos bits em cada endereço ou menos endereços armazenando mais bits. Caso o bloco de memória esteja configurado como uma memória de 36 Kb, é possível configurar a porta de memória para as seguintes configurações de quantidades de endereços e bits armazenados em cada endereço:

- 32K x 1 bit.
- 16K x 2 bits.
- 8K x 4 bits.
- 4K x 8 bits, com 1 bit de paridade adicional.
- 2K x 16 bits, com 2 bits de paridade adicionais.
- 1K x 32 bits, com 4 bits de paridade adicionais.
- 512 x 64 bits, com 8 bits de paridade adicionais.

Caso o bloco esteja configurado como duas memórias de 18 Kb independentes, é possível configurar a porta de memória para as seguintes configurações de quantidades de endereços e bits armazenados em cada endereço:

- 16K x 1 bit.
- 8K x 2 bits.
- 4K x 4 bits.
- 2K x 8 bits, com 1 bit de paridade adicional.
- 1K x 16 bits, com 2 bits de paridade adicionais.
- 512 x 32 bits, com 4 bits de paridade adicionais.

Vale notar que apesar do bloco de memória BRAM comportar o armazenamento de 36 Kb, parte destes bits podem ser utilizados como bits de paridade, reduzindo o armazenamento efetivo para 32 Kb, ou podem ser utilizados como bits de dados adicionais.

A Tabela 4 discrimina a taxa de leitura de dados para a execução das camadas convolucionais, de acordo com a quantidade de ciclos de *clock* necessários para a execução de cada convolução definida na Tabela 3. A coluna que discrimina o tamanho dos *kernels* em bits leva em consideração a quantidade de coeficientes dos mesmos, utilizando representação numérica de 8 bits. A quantidade de dados lidos para cada convolução é proporcional à quantidade de coeficientes de cada *kernel*. Considerado que cada convolução demanda a leitura de um *kernel* da memória, a partir da quantidade de dados de *kernel* lidos por convolução e a quantidade de ciclos de *clock* por convolução, descritos na Tabela 3, determina-se uma taxa mínima de leitura de dados por ciclos de *clock* necessária. De acordo com os ciclos de *clock* necessários para a execução das convoluções em cada camada convolucional, definidos na Tabela 3, a primeira camada convolucional deve executar cada convolução em 1 ciclo de *clock*. Para que isso seja possível, em cada ciclo de *clock,* um *kernel* desta camada deve ser lido da memória BRAM; logo, 2.904 bits devem ser lidos a cada ciclo de *clock,* conforme descrito na Tabela 4*,* considerando a representação numérica de 8 bits e os 363 coeficientes de cada *kernel* desta camada.

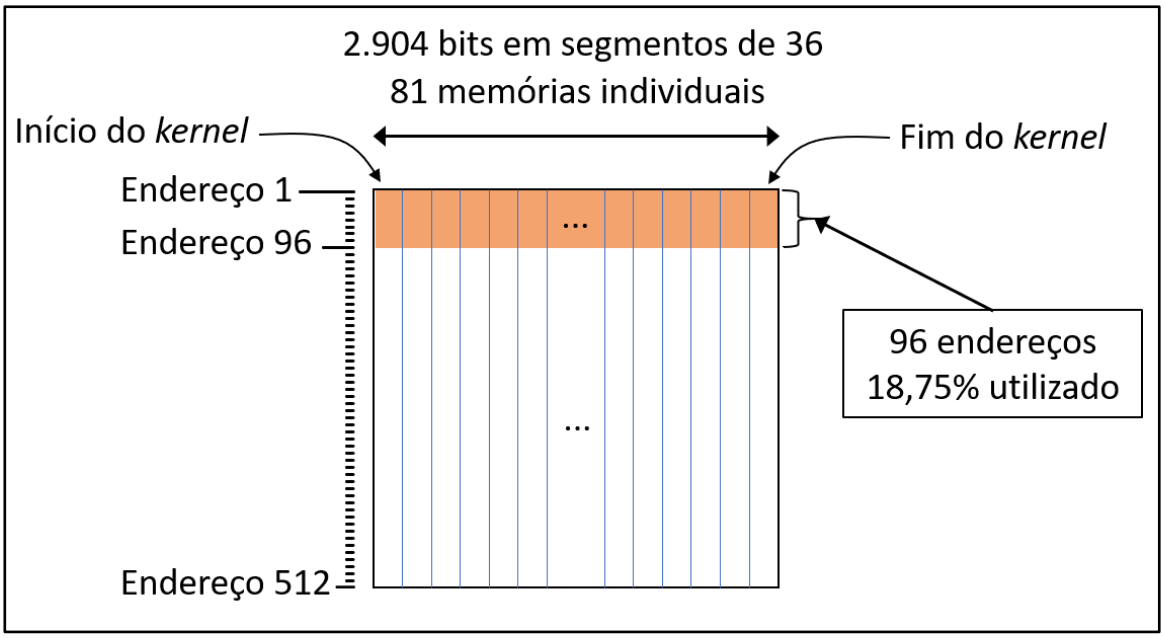
Dado que um único bloco de memória BRAM suporta a leitura de até 72 bits por ciclo de *clock* ou 36 bits por ciclo de *clock*, caso ele seja utilizado como duas memórias independentes, deve-se determinar uma estratégia para armazenamento dos *kernels* em BRAM para atingir o *throughput* de leitura necessário.

Tabela 4 – Taxa mínima de leitura dos *kernels* para execução das camadas convolucionais.

Camada Convolucional	Dimensões e Quantidade de Características dos *Kernels*	Tamanho do *Kernel* (Coeficientes)	Tamanho do *Kernel* (bits)	*Clocks* por Passo Convolucional	Taxa Mínima de Leitura (bits / *clock*)
1	11x11x3	363	2.904	1	2.904
2	5x5x48	1.200	9.600	1	9.600
3	3x3x256	2.304	18.432	4	4.608
4	3x3x192	1.728	13.824	4	3.456
5	3x3x192	1.728	13.824	6	2.304

Para aumentar o *throughput* de leitura dos *kernels*, vários blocos de memória são utilizados como memórias individuais para armazenar segmentos de 36 bits dos *kernels*, atingindo a taxa de leitura necessária com a leitura simultânea destes segmentos. Vários segmentos de 36 bits são lidos em cada ciclo de *clock* e concatenados para que o *kernel* seja formado. No caso da primeira camada convolucional, a segmentação dos 2.904 bits de cada *kernel* em pedaços de 36 bits resulta em 80 segmentos de 36 bits e um segmento de 24. Portanto, são necessárias 81 memórias individuais para armazenamento e leitura de um único *kernel* em um ciclo de *clock*, conforme ilustrado pela Figura 9, onde as memórias individuais estão representadas pelas barras alinhadas horizontalmente. Cada memória individual armazena 96 segmentos de 36 bits, referentes a cada um dos 96 *kernels* da primeira camada, ocupando um total de 96 endereços. Na Figura 9, o armazenamento dos 81 segmentos do primeiro *kernel* está representado no primeiro endereço de cada uma das 81 memórias individuais, os 81 segmentos último kernel estão armazenados no endereço 96 de cada umas das 81 memórias, os demais 416 endereços, de 97 até 512, não são utilizados.

Figura 9 – Alocação dos *kernels* da primeira camada em memórias BRAM, com as memórias individuais representadas pelas barras alinhadas horizontalmente, com seus respectivos 512 endereços representados verticalmente.



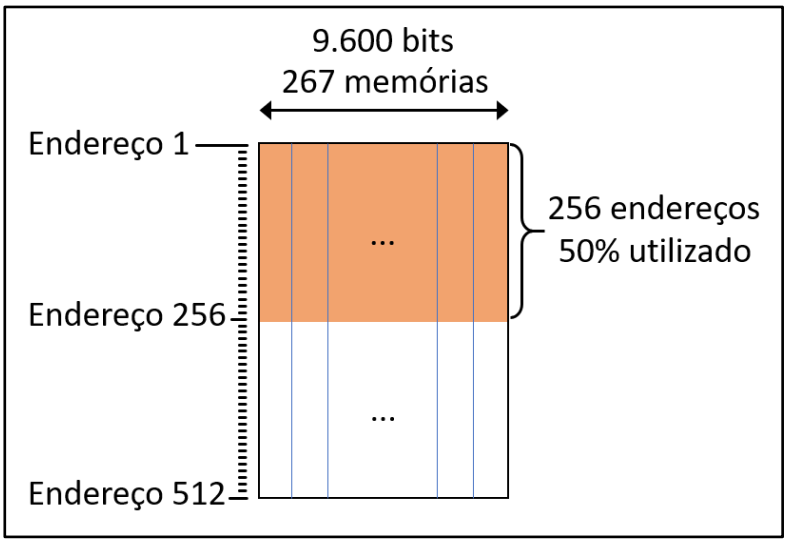
Fonte: Autoria própria (2018)

Conforme mencionado anteriormente, cada BRAM é composta por duas memórias individuais de 18 Kb, sendo que cada uma destas duas memórias possui uma porta para leitura de até 36 bits e outra porta para escrita de até 36 bits. Por isso a capacidade máxima de leitura de um BRAM é de 72 bits por ciclo de *clock*, pois é o resultado da união das duas portas de leitura de 36 bits (XILINX, 2017). O mesmo é válido para a capacidade de escrita dos BRAMs. Como consequência, os 416 endereços restantes, que não são utilizados para armazenar os *kernels* da primeira camada, não podem ser reaproveitados de outra forma, pois as portas disponíveis nas memórias individuais já estão comprometidas com a leitura dos 96 primeiros endereços. Só seria possível realizar leituras nos 416 endereços restantes nos ciclos de *clock* em que não são realizadas leituras nos 96 primeiros endereços. Em termos práticos, isso inviabiliza o acesso aos 416 endereços restantes, pois a alta taxa de leitura dos 96 endereços iniciais comprometeria o desempenho da leitura dos 416 endereços restantes.

Como a primeira camada convolucional utiliza poucos *kernels* em comparação com as demais camadas convolucionais, poucos endereços das memórias individuais são utilizados, tendo um aproveitamento de apenas 18,75% dos endereços. Neste caso, o uso de *distributed* RAM deve ser considerado, mesmo que este tipo de memória seja mais escasso. Tendo como base o dispositivo XC7Z100 da Xilinx, utilizado nos experimentos, as 81 memórias individuais necessárias para armazenamento dos *kernels* da primeira camada representariam 5,36% dos BRAMs do dispositivo, enquanto o armazenamento dos *kernel* em *distributed* RAM resultaria em 2,09% deste recurso, mesmo ele sendo mais escasso. Portanto, considerando os recursos deste dispositivo em questão, é mais vantajoso armazenar os *kernels* da primeira camada em *distributed* RAM.

A mesma estratégia de segmentação é adotada para o armazenamento dos kernels da segunda camada convolucional. Os *kernels* da segunda camada contêm 1.200 coeficientes, resultando em 9.600 bits de dados em numérica de 8 bits. O armazenamento destes dados em memórias de 36 bits de largura exige 267 memórias, conforme ilustrado pela Figura 10. São utilizados os 256 endereços iniciais das memórias individuais, conforme ilustrado na Figura 10, pois cada *kernel* ocupa um endereço de memória e a segunda camada contém 256 *kernels*, utilizando 50% dos 512 endereços disponíveis.

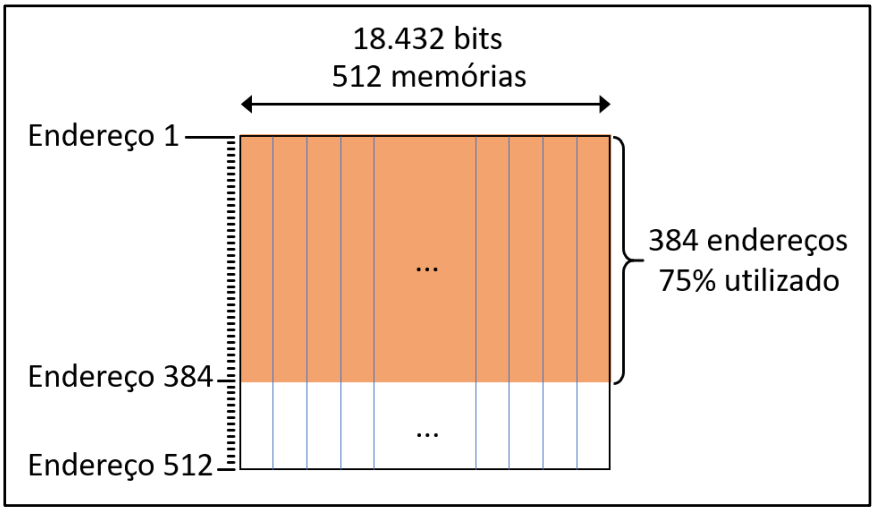
Figura 10 – Alocação de 9.600 bits de cada *kernel* da segunda camada em BRAMs.



Fonte: Autoria própria (2018)

Para a terceira camada, aplicando novamente a estratégia de segmentação utilizada nas camadas anteriores, o armazenamento dos 384 *kernels* desta camada convolucional demanda 384 endereços de memória de cada uma das 512 memórias individuais necessárias para o armazenamento dos 18.432 bits de cada *kernel*, conforme ilustrado pela Figura 11.

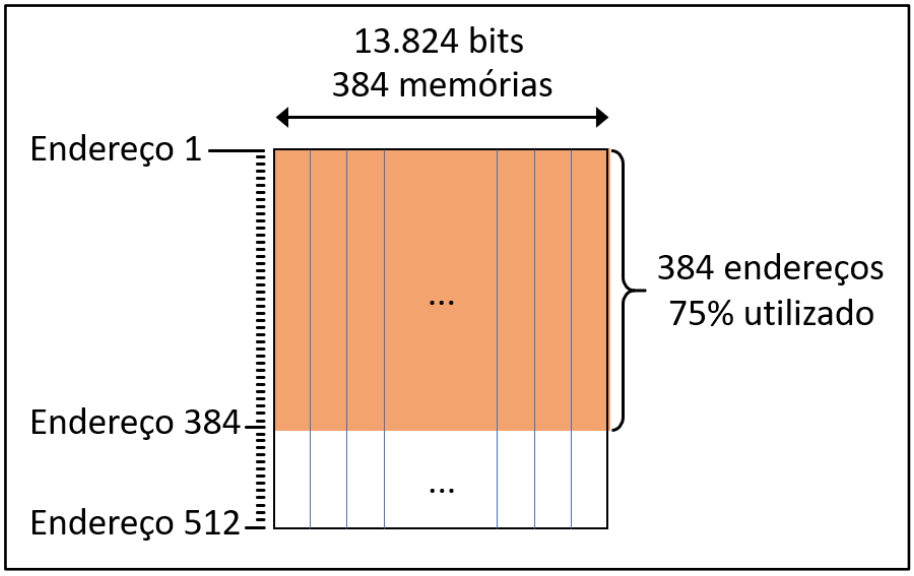
Figura 11 – Alocação dos 18.432 bits de cada *kernel* da terceira camada em memória BRAM.



Fonte: Autoria própria (2018)

O armazenamento dos kernels da quarta camada convolucional utiliza 384 memórias individuais para armazenar os 13.824 bits de cada um dos *kernels*. Coincidentemente, esta camada utiliza 384 *kernels*, portanto, 384 endereços das 384 memórias são utilizados, sendo o primeiro *kernel* armazenado no primeiro endereço e o último *kernel* no endereço 384, ilustrados na Figura 12.

Figura 12 – Alocação dos 13.824 bits de cada *kernel* da quarta camada em memória BRAM.

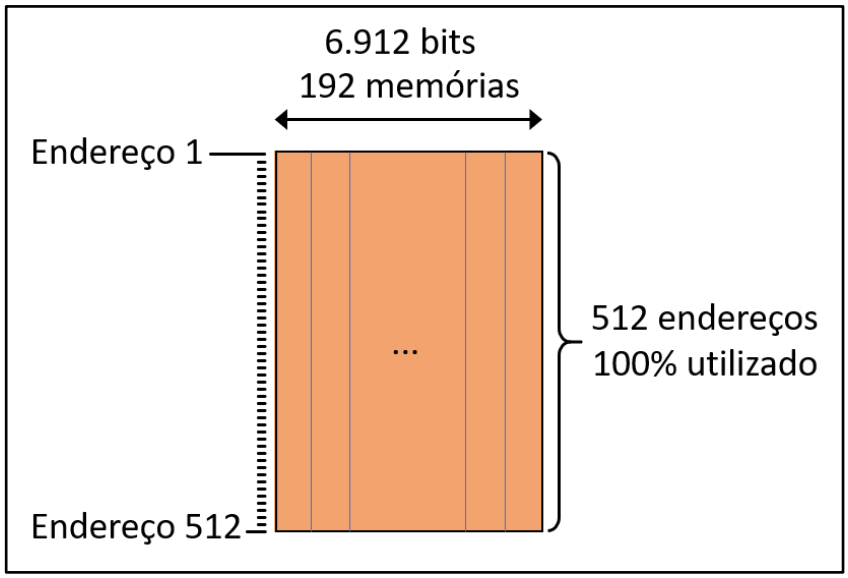


Fonte: Autoria própria (2018)

A quinta camada convolucional utiliza *kernels* com 1.728 coeficientes, cujo armazenamento demandaria 384 memórias com 36 bits de largura e utilizaria 256 endereços, ou seja, 50% dos 512 disponíveis em cada uma destas memórias, pois esta camada utiliza 256 *kernels*. Como cada memória individual permite a leitura de um endereço por ciclo de *clock*; isso possibilitaria que, na quinta camada, também fosse lido um *kernel* a cada ciclo de *clock*. Porém, conforme descrito na Tabela 4 e originalmente na Tabela 3, que descreve o balanceamento do *pipeline*, a quinta camada convolucional utiliza seis ciclos de *clock* por convolução. Portanto, não é necessário que cada *kernel* seja lido em um ciclo de *clock*, pois um *kernel* é utilizado a cada seis ciclos de *clock*. Neste caso, pode-se utilizar mais de um endereço para armazenamento de cada *kernel*, armazenando um segmento do *kernel* em cada endereço. Como esta camada utiliza 256 *kernels*, pode-se utilizar dois endereços por *kernel*, com cada endereço armazenando metade de um *kernel*, consequentemente, utilizando 512 endereços de 192 memórias individuais, conforme ilustrado pela Figura 13. A utilização de dois endereços de memória para cada *kernel* demanda dois ciclos de *clock* para o carregamento de um kernel, sendo possível atender a taxa de leitura de um kernel a cada seis ciclos de *clock* necessários para o cômputo de cada convolução desta camada. A utilização de dois endereços por *kernel* possibilitou que 100% dos endereços fossem utilizados, evitando ociosidade dos endereços, e reduziu

a quantidade de memórias utilizadas, pois utiliza 192 memórias ao invés das 384 memórias que seriam necessárias para armazenar cada *kernel* em um único endereço.

Figura 13 – Alocação dos 6.912 bits de cada *kernel* da quinta camada convolucional em memória BRAM.



Fonte: Autoria própria (2018)

Apesar da segunda camada convolucional também utilizar apenas 50% dos endereços de memória, nesta camada não é possível adotar dois endereços para armazenamento de cada *kernel* sem degradar o desempenho da rede, porque isso resultaria em dois ciclos de *clock* para leitura de cada *kernel*. Consequentemente, a segunda camada demandaria dois ciclos de *clock* para o cômputo da convolução devido à taxa de leitura dos *kernels* da memória, o que tornaria a segunda camada convolução o gargalo do *pipeline.*

Redução semelhante também poderia ser feita na terceira e quarta camadas convolucionais utilizando quatro endereços para armazenar cada *kernel*, consequentemente, reduzindo em 25% a quantidade de memórias individuais utilizadas. Porém, essa redução exigiria um mecanismo para tratamento de uma quantidade de endereços que não é uma potência de dois, pois para 384 *kernels* destas camadas, ao utilizar quatro endereços por *kernel*, seriam necessários 1.536

endereços. Para esta quantidade de endereços, o mecanismo discutido no Apêndice A – Endereçamento Conjugado de BRAMs seria necessário para combinar um endereçamento de 1.024 endereços com um de 512 endereços. Para o dispositivo utilizado nos experimentos, não foi necessário realizar esta redução por existirem BRAMs suficientes.

Ao todo, 1.355 memórias individuais foram utilizadas para armazenar os *kernels* das camadas 2, 3, 4 e 5, contabilizando 678 BRAMs utilizadas. Caso as 81 memórias individuais fossem utilizadas para armazenas os *kernels* da primeira camada convolucional, esse total seria de 1.436 memórias individuais utilizadas, portanto, demandando 718 BRAMs.

3.5. *BUFFERS* TUBULARES PARA REUSO DE MEMÓRIA

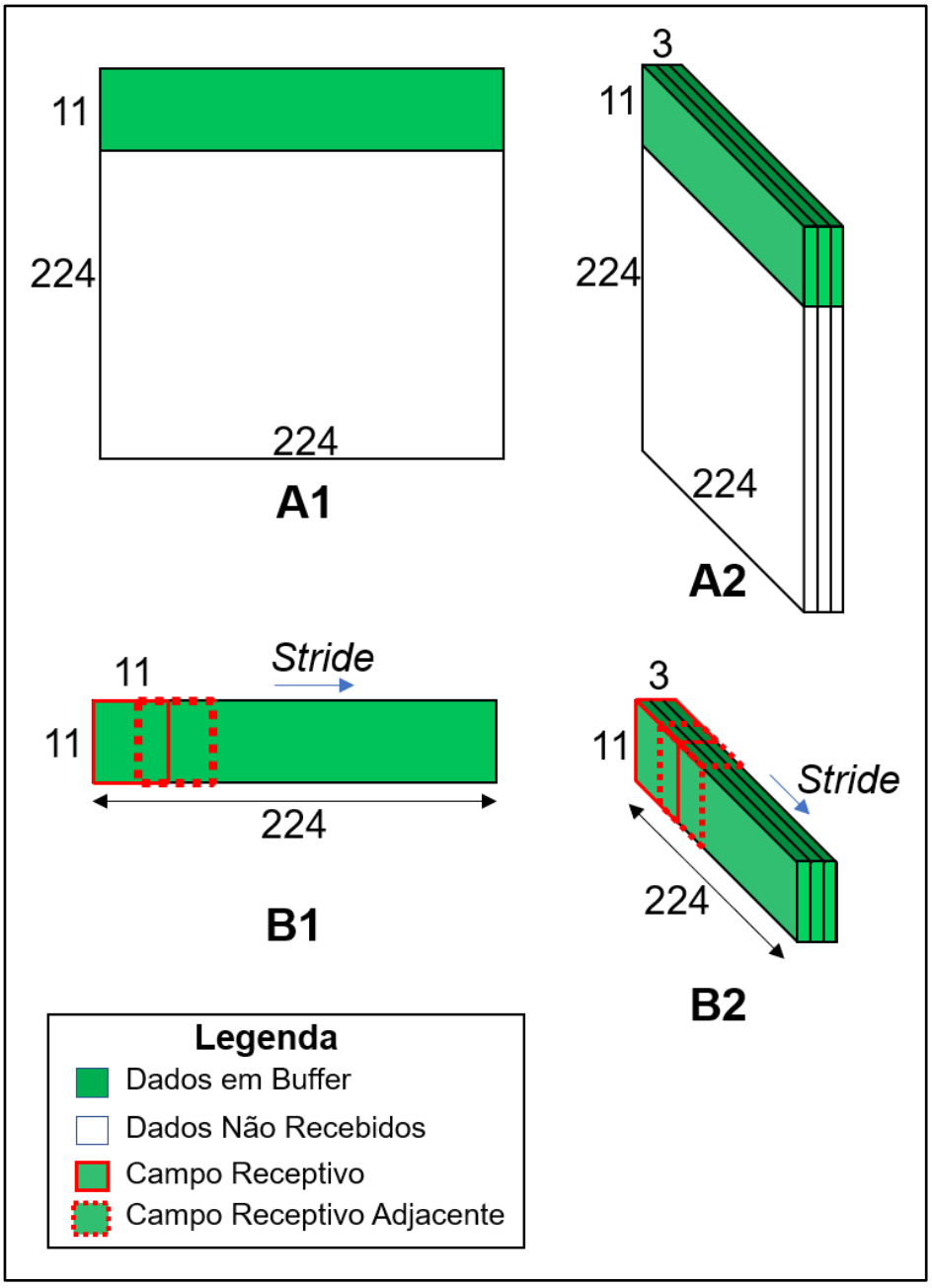
Para gerenciamento dos dados que constituem os mapas de características, foram utilizados *buffers* que reaproveitam endereços de memória e, consequentemente, reduzem a demanda por memória. Estes *buffers* permitem a reutilização dos endereços de memória que contenham dados obsoletos para recepção de novos dados, o que faz com que menos memória seja necessária. Cada instância de *buffer* é conectada a dois componentes, um deles é responsável por fornecer dados que devem ser armazenados no *buffer* e o outro é responsável por consumir os dados armazenados. Estes componentes que consomem dados realizam o processamento dos mesmos e fornecem o resultado do processamento para ser armazenado em outro *buffer.*

Os *buffers* são utilizados em todas as camadas convolucionais para armazenar dados das operações de convolução ou das operações de *pooling;* isso significa que tanto os componentes fornecedores quanto os consumidores podem ser os componentes que realizam as convoluções ou operações de *pooling*. Para que isso seja possível, o *buffer* deve ser suficientemente versátil para suportar o armazenamento de dados de mapas de características variados. Portanto, o componente de hardware referente ao *buffer* foi descrito de maneira parametrizável, permitindo que os *buffers* possam ser

instanciados com diferentes configurações, de acordo com as propriedades do mapa de características em questão. Além das dimensões dos mapas de características, os *buffers* também recebem parâmetros referentes às dimensões do campo receptivo e *stride*. Estes parâmetros são utilizados para verificar se foram recebidos dados suficientes para formar um novo campo receptivo. Quando existem dados suficientes, o *buffer* emite um sinal para seu respectivo consumidor de dados para que ele realize a leitura do campo receptivo formado.

Para realizar o armazenamento de dados no *buffer*, é necessário que o componente fornecedor informe o dado a ser armazenado e também notifique o envio de um novo dado. Portanto, para tal notificação, existe uma porta de entrada no *buffer* responsável por receber um sinal binário vindo do componente fornecedor de dados, solicitando a escrita de um novo dado no *buffer*. O evento da borda de subida do sinal de *clock* durante a sinalização do envio de um novo dado é o gatilho para que o *buffer* realize o recebimento dos dados submetidos na sua entrada e os armazene em uma estrutura de dados tridimensional, equivalente a um segmento do mapa de características, conforme ilustrado pela Figura 14. Na figura, são ilustradas duas perspectivas (A1 e A2) de um mapa de características de 224 linhas, 224 colunas e três características (224 x 224 x 3), equivalente às dimensões da entrada da primeira camada convolucional. Também são ilustradas nesta figura, duas perspectivas (B1 e B2) do *buffer* utilizado para armazenar um segmento deste mapa de características.

Figura 14 – Utilização de Buffers para armazenamento de mapas de características. **A1** e **A2** são duas perspectivas diferentes de um mapa de características de dimensões 224 x 224 x 3. **B1** e **B2** são duas perspectivas diferentes do buffer utilizado para armazenar o mapa de características.



Fonte: Autoria própria (2018)

A posição do armazenamento dos dados no *buffer* leva em consideração que os dados transmitidos para o *buffer* são as características que formam cada hiperpixel. O

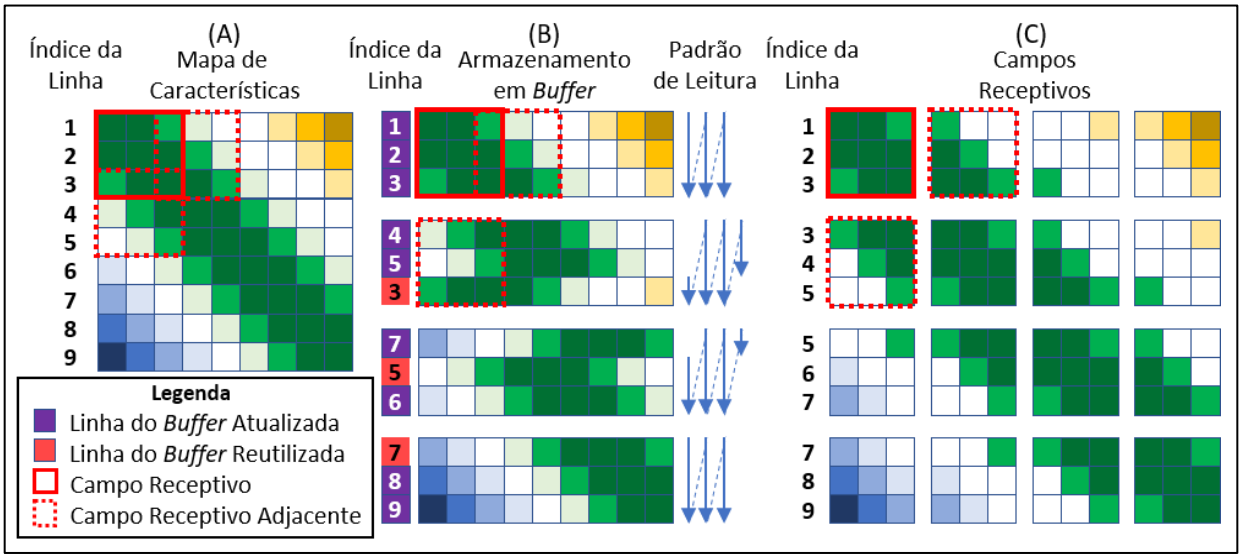
hiperpixel é o vetor formado pelo conjunto de características extraídas pelas convoluções de múltiplos *kernels* com um campo receptivo. Portanto, cada hiperpixel carrega um conjunto de informações referentes a um determinado campo receptivo do mapa de características anterior, da mesma forma que um *pixel*, na representação RGB, carrega as características referentes às intensidades de vermelho, verde e azul. Quando todas as características de um hiperpixel são transmitidas para o *buffer*, inicia-se a transmissão das características do próximo hiperpixel, pertencente à próxima coluna e à mesma linha. Quando todos os hiperpixels de uma linha são transmitidos, inicia-se novamente o mesmo processo com a transmissão do primeiro hiperpixel da linha seguinte, até que todas as linhas sejam recebidas. Esta ordenação é consequência da execução das convoluções, sem a necessidade de processamento adicional A convolução de um campo receptivo com os respectivos *kernels,* já resulta na formação do vetor de características que compões um hiperpixel do mapa de características consecutivo. Basta que a ordenação dos *kernels* utilizados seja mantida para todos os campos receptivos. A ordenação dos hiperpixels no mapa de características consecutivo também é consequência do posicionamento original dos campos receptivos utilizados nas convoluções.

Uma consequência desta ordenação natural dos dados é a possibilidade do armazenamento parcial do mapa de características. Na Figura 15, é ilustrado um mapa de características de tamanho 9x9 (A), quatro estados diferentes do *buffer* (B), cada estado representando os dados armazenados no *buffer* após a recepção completa das linhas 3, 5, 7 e 9 do mapa de características e os campos receptivos lidos do *buffer* (C). Na Figura 15, o primeiro e o segundo estado em (B), mostram que a terceira linha do campo receptivo foi mantida no *buffer*, pois ela forma tanto os campos receptivos contidos nas linhas 1, 2 e 3, quanto os campos receptivos formados pelas linhas 3, 4 e 5. Portanto, no segundo estado, a região de memória onde armazenava as linhas 1 e 2 foi reutilizada para armazenar as linhas 4 e 5.

O reaproveitamento dos endereços de memória para armazenamento de novos dados faz com que a escrita e a leitura no *buffer* sejam feitas de maneira análoga a um *buffer* circular, neste caso, um *buffer* circular tridimensional. Da mesma maneira que existe a continuidade do último endereço para o primeiro endereço no *buffer* circular, no buffer tubular, existe a continuidade da última para a primeira linha, por isso o nome

“tubular”. Essa continuidade faz com que a primeira e a última linha do *buffer* se tornem adjacentes. Portanto, na Figura 15 (B), após a utilização da terceira linha do *buffer* para o armazenamento da terceira linha de mapa de características, a primeira linha do *buffer* é utilizada novamente para a escrita da quarta linha do mapa de características, sobrescrevendo os dados antigos da primeira linha do mapa de características. A leitura do *buffer* também deve levar em consideração esta continuidade para que o campo receptivo mantenha a mesma estrutura apresentada no mapa de características, como ilustrado pela coluna “padrão de leitura” da Figura 15 (B).

Figura 15 – Ilustração do *buffer* tubular para um mapa de características 9x9, campo receptivo 3x3 e *stride* 2.



Fonte: Autoria própria (2018)

A adoção do *stride* de dois, utilizado no exemplo da Figura 15, para campos receptivos de tamanho 3 x 3, resulta em uma sobreposição de uma coluna entre os campos receptivos compostos pelas mesmas linhas e a sobreposição de uma linha entre os campos receptivos compostos pelas mesmas colunas. Esta sobreposição entre as linhas do mapa de características permite o reuso destes dados armazenados no *buffer*, pois essa sobreposição resulta no compartilhamento de dados entre campos receptivos adjacentes. Quando todas as convoluções com um campo receptivo são concluídas, parte da região do *buffer* que armazena este campo receptivo pode ser

reutilizada para o armazenamento de novos dados, preservando apenas a região de memória que contenha dados compartilhados com campos receptivos que ainda serão usados.

Para que o *buffer* acumule dados suficientes para realizar as convoluções, é necessário que ele tenha capacidade de armazenar uma quantidade de linhas do mapa de características suficiente para compor um campo receptivo. Uma vez que os dados armazenados no *buffer* permitem a formação de um campo receptivo, as convoluções podem ser realizadas, liberando o reaproveitamento de parte da memória que armazena o campo receptivo. Na Figura 15, essa parte que pode ser liberada é a região formada pelos quatro hiperpixels das duas primeiras linhas e colunas de cada campo receptivo, pois não são compartilhados com o campo receptivo seguinte. Portanto, um *buffer* que comporte uma quantidade de linhas do mapa de características equivalente à quantidade de linhas do campo receptivo já é suficiente para armazenar os dados dos mapas de características utilizados nas convoluções. Isso implica uma redução significativa de memória em relação ao armazenamento completo de um mapa de características.

3.6. ARQUITETURA

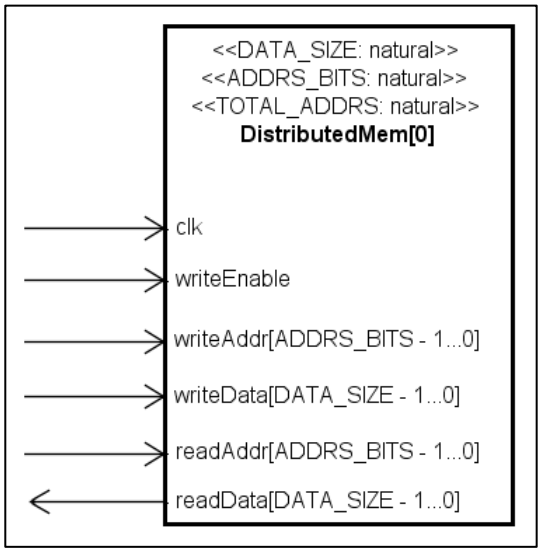
Esta seção descreve os componentes projetados para execução das camadas convolucionais em FPGA. A apresentação da arquitetura segue uma abordagem *bottom- up*, descrevendo inicialmente os componentes mais simples, seguindo para os componentes mais complexos, composto pelos componentes simples. Os componentes projetados foram escritos utilizando a linguagem de descrição de *hardware VHDL* (*VHSIC Hardware Description Language*, onde VHSIC é a sigla de *Very High Speed Integrated Circuits*). O código em VHDL está disponível para ser acessado no repositório *online GitHub*[2].

[2] Endereço do repositório *online*: https://github.com/markfsousa/convnet_fpga

3.6.1. Componente do Buffer Tubular

O primeiro componente a ser descrito é o componente de manipulação da *distributed* RAM, ilustrado na Figura 16, denominado ***DistributedMem***. Ou seja, este componente é responsável por ler e escrever dados da memória formada pela aglomeração das diversas *Lookup Tables* (LUTs) presentes nas células nos FPGAs. Este componente segue um padrão fornecido pelo fabricante do FPGA para que a ferramenta de síntese de *hardware* seja capaz de inferir que a *distributed* RAM deve ser utilizada para armazenar os dados manipulados por este componente.

Figura 16 – Componente para armazenamento e leitura de dados em *distributed* RAM, **DistributedMem***.*



Fonte: Autoria própria (2019)

Uma característica relevante deste componente é ser parametrizável, sendo possível determinar quantos bits são utilizados para dados e quantos bits são utilizados para endereçamento. No momento em que o componente é instanciado, devem ser informados os parâmetros:

- *DATA_SIZE*, para determinar quantos bits serão utilizados para dados;
- *ADDRS_BITS*, para determinar quantos bits serão utilizados nos endereços;
- *TOTAL_ADDRS*, para informar quantos endereços serão utilizados.

O componente *DistributedMem* possui cinco portas de entrada e uma de saída, sendo as quantidades de bits nas portas de dados e portas de endereço determinadas pela parametrização do componente.

Para realizar a escrita de dados, deve-se sinalizar na porta *writeEnable* durante uma borda de subida de *clock* que uma escrita deve ser realizada, informando o endereço da escrita na porta *writeAddr* e o dado a ser escrito na porta *writeData*. Estes dados são escritos no índice de memória referente ao endereço informado. A ferramenta de síntese de *hardware* fornecida pela Xilinx se encarrega de alocar estes índices nas LUTs do FPGA.

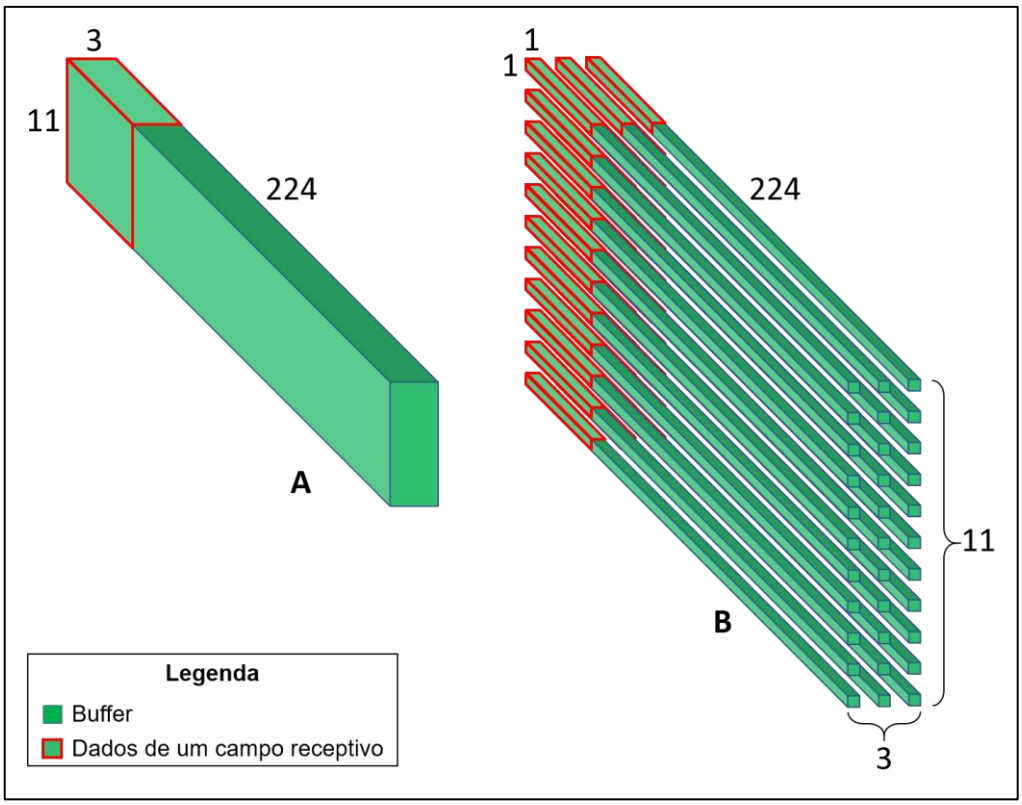
Apesar do componente *DistributedMem* suportar leitura assíncrona, os demais componentes da arquitetura, que utilizam este componente, funcionam sincronamente. Ou seja, eles solicitam a leitura em função da mudança do ciclo de *clock* e utilizam os dados lidos no próximo ciclo de *clock.* A porta *readAddr* é utilizada para informar o endereço de leitura e o dado lido é submetido na porta de saída *readData*.

O componente *DistributedMem* é capaz de ler e escrever um dado em um ciclo de *clock*. Porém, para a leitura de um campo receptivo do *buffer* com a leitura de um dado por ciclo de *clock*, seriam necessários tantos ciclos de *clock* quanto elementos em cada campo receptivo. Portanto, seriam necessários 363, 1.200, 2.304, 1.728 e 1.728 ciclos de *clock* para leitura de cada campo receptivo da primeira, segunda, terceira, quarta e quinta camada convolucional, respectivamente. Essa quantidade de ciclos de *clock* ultrapassaria a quantidade de ciclos utilizados para realizar as convoluções do campo receptivo com todos os *kernels* das respectivas camadas convolucionais. Ou seja, o *buffer* tubular demandaria mais tempo para ler um campo receptivo da memória do que o tempo necessário para realizar as convoluções do campo receptivo com todos os *kernels* da camada.

Para diminuir o tempo necessário para a leitura dos campos receptivos, foi utilizada uma técnica parecida com a utilizada para armazenamento dos *kernels* em múltiplos BRAMs. Esta técnica consiste na segmentação das memórias, utilizando *distributed* RAMs menores e realizando a leitura de várias memórias em paralelo.

A Figura 17 mostra o *buffer* da primeira camada convolucional utilizando múltiplas memórias para aumentar o *throughput* de leitura com a leitura simultânea destas memórias. Na Figura 17, em (A), o *buffer* tubular está ilustrado conforme apresentado na seção 3.5, enquanto em (B), é ilustrado como é feita a segmentação dos dados do *buffer* para armazenamento em memórias separadas. Este *buffer* contém 11 linhas, 224 colunas e 3 características, referentes aos canais RGB da entrada. Os campos receptivos desta camada têm dimensão 11 x 11 x 3, totalizando 363 elementos em cada campo receptivo. Para a primeira camada convolucional, para cada linha e cada característica do *buffer* é utilizada uma *distributed* RAM independente. Portanto, são necessários 33 componentes *DistributedMem*, cada um sendo capaz de armazenar 224 dados, equivalente às 224 colunas de cada uma das 3 características.

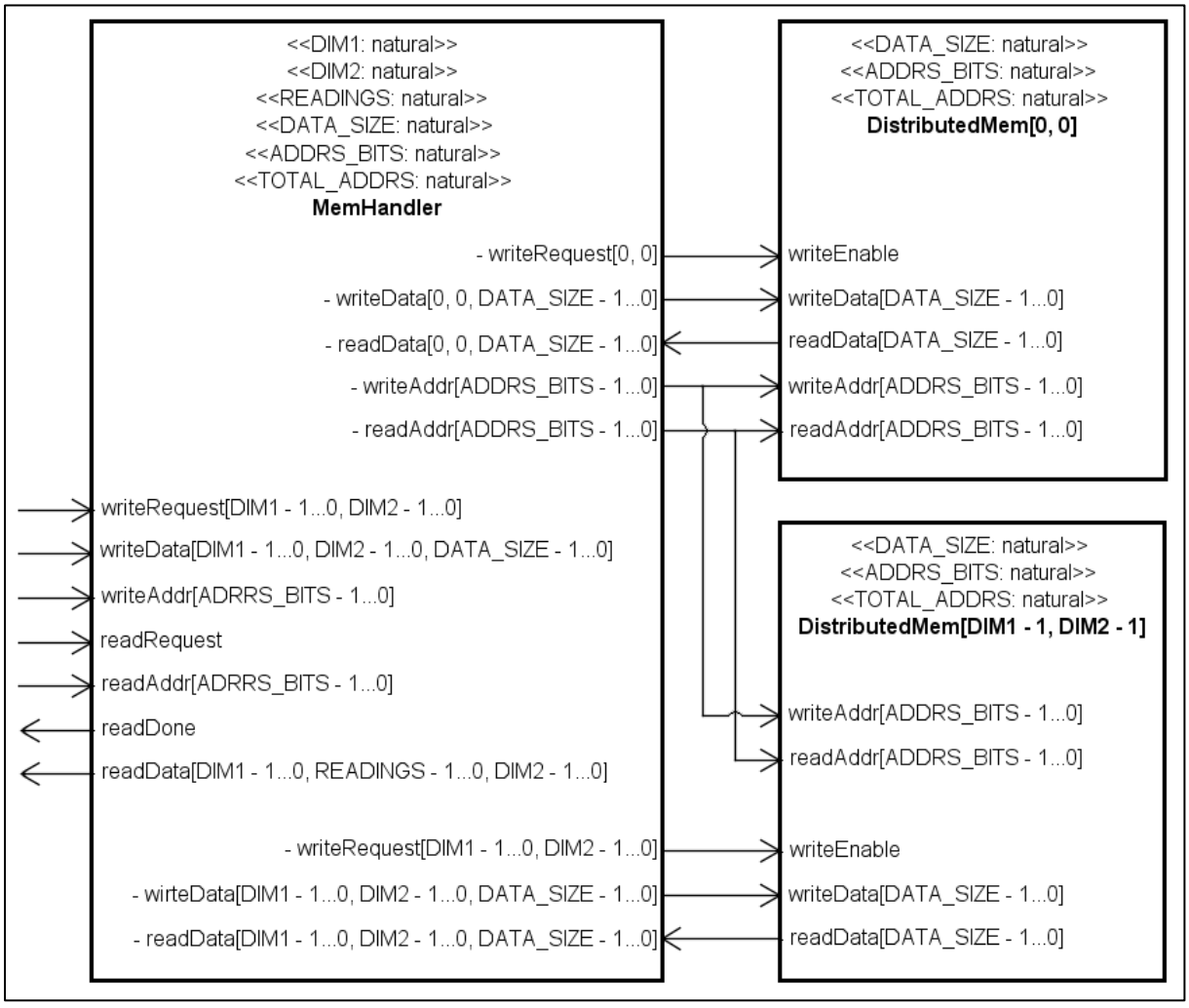
Figura 17 – *Buffer* tubular da primeira camada convolucional utilizando paralelismo de 33 memórias para aumento de *throughput* de leitura, cada memória contendo 224 endereços. **(A)** representação do *buffer* e do campo receptivo. **(B)** alocação dos dados do *buffer* em 33 memórias.



Fonte: Autoria própria (2019)

O funcionamento do *buffer* tubular permanece o mesmo elucidado na seção 3.5, pois a organização dos dados nas *distributed* RAMs é abstraída por um componente de *hardware* responsável por manipular os índices de memória equivalentes nos diferentes *DistributedMem*. Este componente manipulador de memória, denominado ***MemHandler***, está ilustrado na Figura 18. O sinal de *clock* foi omitido na figura para melhorar a legibilidade.

Figura 18 – Componente manipulador de memória, ***MemHandler***.



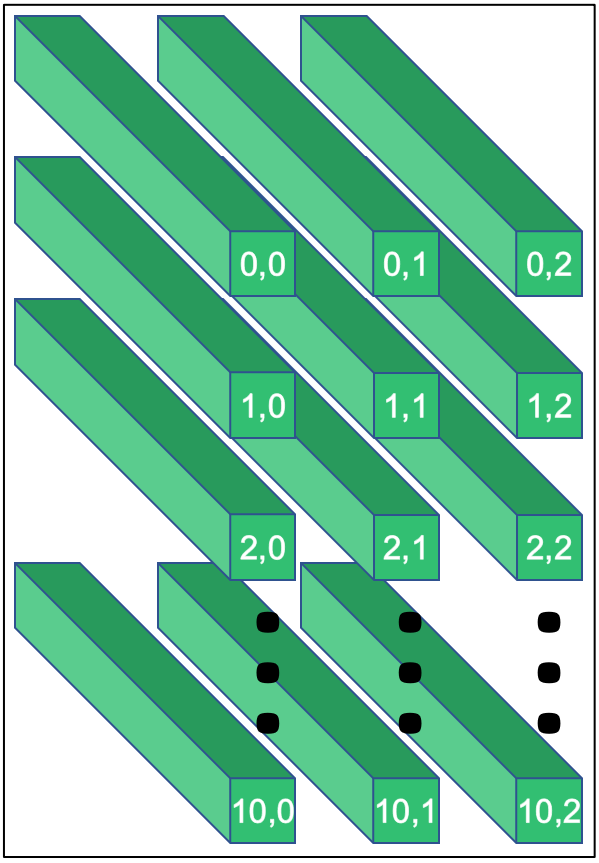
Fonte: Autoria própria (2019).

Na Figura 18, apenas duas instâncias do componente *DistributedMem* são ilustradas, embora sejam utilizadas 33 instâncias deste componente para o *buffer* tubular da

primeira camada convolucional. O componente *MemHandler* também é parametrizável e ao ser instanciado deve receber os parâmetros:

- *DIM1 e DIM2*, indicando quantas *distributed* RAMs serão utilizadas, ou seja, quantos componentes *DistributedMem* deverão ser instanciados. No exemplo da primeira camada convolucional, estes parâmetros devem ser 11 e 3, conforme ilustrado pela Figura 17. Como resultado, é criada uma matriz de componentes de memória com 33 componentes. Portanto, o endereçamento nestas memórias deve ser feito indicando dois índices entre 0 e 10 e entre 0 e 2, para identificar qual das memórias deverá ser acessada e o endereço a ser acessado nesta memória. A identificação das instâncias de memória deve ser feita conforme os índices ilustrados da Figura 19;
- *READINGS*, indicando quantas leituras sequenciais serão realizadas nas *distributed* RAM para recuperar um campo receptivo, ou seja, quantos endereços diferentes serão acessados para ler um campo receptivo. No exemplo da primeira camada convolucional, este parâmetro deve ser 11, pois 11 endereços diferentes deverão ser lidos das 33 memórias para recuperar um campo receptivo contendo 363 elementos;
- *DATA_SIZE*, *ADDRS_BITS* e *TOTAL_ADDRS*, que são os parâmetros utilizados para criar as 33 instâncias do componente *DistributedMem*.

Figura 19 – Índices de identificação das instâncias de *DistributedMem* utilizadas pelo componente *MemHandler*.



Fonte: Autoria própria (2019).

O componente *MemHandler* contém sete portas, além da porta do sinal de *clock*, sendo cinco portas de entrada e duas de saída, descritas a seguir, considerando o exemplo do *buffer* tubular da primeira camada convolucional:

- *writeRequest* (entrada), contendo tantos bits quanto memórias paralelas. Cada bit indica uma solicitação de escrita em uma memória paralela, portanto 33 bits. Note que, no diagrama da Figura 18, cada bit é destinado a uma instância *DistributedMem* diferente, sendo equivalente ao sinal *writeEnable* desta instância, seguindo o mapeamento de memórias apresentado na Figura 19;
- *writeData* (entrada), contendo tantos dados quanto memórias paralelas, portanto, 33 para o exemplo da primeira camada. Cada dado é escrito na *distributed* RAM correspondente apenas quando o bit equivalente na porta *writeRequest* sinaliza uma solicitação de escrita, possibilitando a escrita de 1

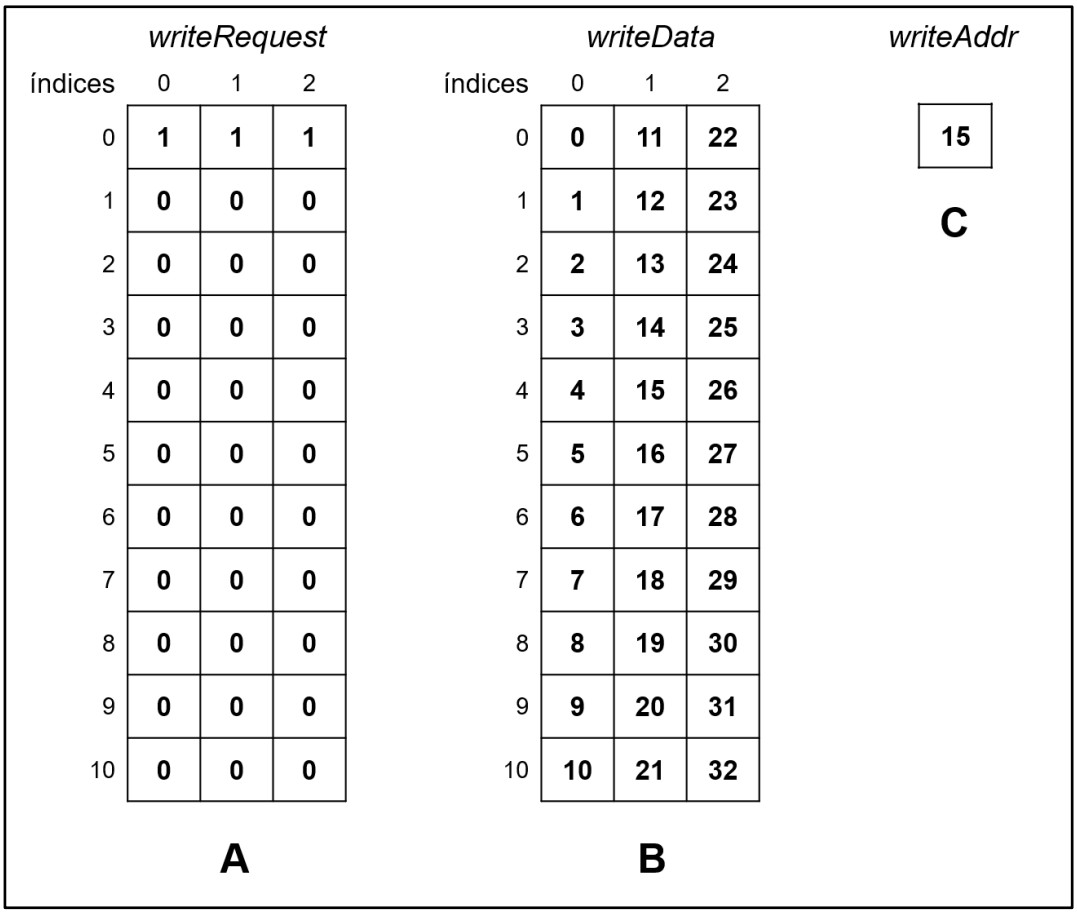
até 33 dados simultaneamente. Assim como a porta *writeRequest*, esta porta também segue o mapeamento de memórias apresentado na Figura 19;

- *writeAddr* (entrada), indicando o endereço utilizado para escrita dos dados nas *distributed* RAMs. Portanto, os dados são escritos nos mesmos endereços em todas as memórias;
- *readRequest* (entrada), indicando quando uma leitura deve ser realizada;
- *readAddr* (entrada), indicando o endereço onde a leitura deve ser realizada. São realizadas múltiplas leituras consecutivas nos endereços adjacentes, até atingir uma quantidade de leituras equivalente ao parâmetro *READINGS*, completando a leitura de um campo receptivo. Para o exemplo da primeira camada, são realizadas 11 leituras consecutivas nas 33 memórias;
- *readDone* (saída), quando todas as leituras consecutivas são realizadas, esta porta de saída indica a conclusão da leitura;
- *readData* (saída), retorna todos os dados lidos em todas as leituras consecutivas.

Uma característica importante deste componente é que apesar de ele ser capaz de ler e escrever dados em várias memórias simultaneamente, ele utiliza apenas dois endereços, um para leitura e outro para escrita. Portanto, ele escreve dados diferentes em memórias diferentes simultaneamente, mas desde que seja no mesmo endereço em todas as memórias, pois o endereço é compartilhado entre as memórias. O endereço de leitura também é compartilhado entre as memórias, portanto, ele é capaz de ler o mesmo endereço de várias memórias simultaneamente.

A escrita neste componente acontece quando existe pelo menos um bit na porta *writeRequest* indicando que a escrita deve ser realizada. Os índices destes bits de requisição de leitura são utilizados para identificar quais das instâncias dos componentes *DistributedMem* devem ser utilizados para realizar a escrita e também recuperar a estrutura do campo receptivo lido. Para os valores exemplificados na Figura 20, os dados 0, 11 e 22 serão escritos no endereço 15 das memórias relacionadas aos componentes *DistributedMem* de índices (0, 0), (0, 1) e (0, 2) respectivamente.

Figura 20 – Exemplo de entradas para as portas *writeRequest, writeData* e *writeAddr*. **(A)** solicitação de escrita, **(B)** dados para escrita, **(C)** endereço para escrita.

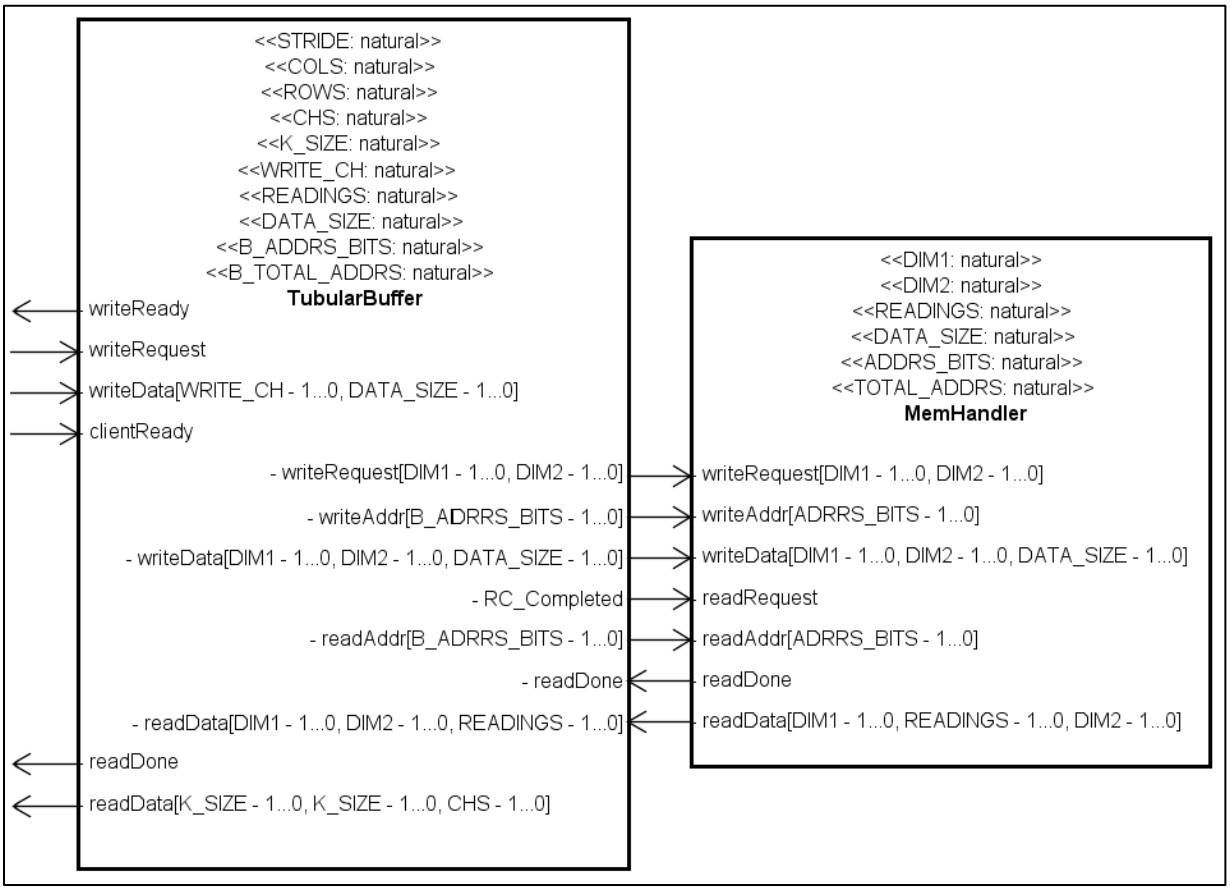


A leitura é realizada pelo componente *MemHandler* quanto o bit da porta de entrada *readRequest* equivale a um. Neste caso, é realizada uma série de leituras simultâneas em todos os componentes de memória distribuída *DistributedMem*. Em cada ciclo de *clock*, um mesmo endereço é lido de todas as memórias simultaneamente. A primeira leitura é realizada no endereço indicado na porta de entrada *readAddr*, seguida pelas leituras nos endereços consecutivos, até completar a leitura de todos os dados do campo receptivo em questão. Terminada a leitura, a porta de saída *readDone* sinaliza a conclusão e os dados são submetidos na porta de saída *readData*.

O componente ***TubularBuffer***, ilustrado na Figura 21, é o componente que executa o *buffer* tubular. Ele é o componente responsável pelo controle dos dados lidos e

escritos no *buffer*. Portanto, ele é o componente que calcula os índices dos dados em relação a sua posição no *buffer* e determina se o *buffer* comporta a recepção de novos dados e também determina se os dados armazenados são suficientes para a leitura de um novo campo receptivo.

Figura 21 – Componente do *buffer* tubular, ***TubularBuffer*.**



Fonte: Autoria própria (2019).

O cálculo dos índices dos dados no *buffer* tubular deve considerar o reaproveitamento das linhas, apresentado na seção 3.5. Portanto, é necessário que este componente tenha as informações referentes às dimensões do *buffer* e do campo receptivo, inclusive do *stride*. Para isso, o componente *TubularBuffer* permite a configuração de alguns parâmetros, descritos a seguir:

- *STRIDE*: define o distanciamento entre os campos receptivos. Por padrão, este parâmetro assume valor um;

- *COLS*: define quantas colunas são utilizadas no *buffer*. Este valor equivale à quantidade de colunas no mapa de características que está sendo armazenado no *buffer*;
- *ROWS*: determina quantas linhas serão utilizadas para armazenar os dados dos campos receptivos. Este valor deve ser, no mínimo, igual à quantidade de linhas do campo receptivo. Embora o mínimo de linhas necessárias para a execução do *buffer* tubular seja igual à quantidade de linhas do campo receptivo, o componente também permite que o *buffer* tenha mais linhas, portanto, utilizando mais memória. Esta opção pode ser utilizada em casos com pouca restrição de memória para evitar que o *buffer* fique cheio. Quando o *buffer* está cheio, ele impede o recebimento de novos dados até que sejam realizadas novas leituras para liberar espaço;
- *CHS*: define a quantidade de características armazenadas no *buffer*. Este valor é equivalente à quantidade de características do mapa de característica armazenado no *buffer*;
- *K_SIZE*: determina qual o tamanho de *kernel* utilizado para convoluções com os campos receptivos, consequentemente, o tamanho do campo receptivo, pois ambos têm as mesmas dimensões. Este parâmetro considera que os *kernels* de uma mesma camada tem a mesma quantidade de linhas e colunas, dado que este é o caso da AlexNet;
- *WRITE_CH*: indica quantas características são armazenadas em cada escrita no *buffer*. Portanto, este parâmetro é utilizado para configurar a porta de entrada de dados *buffer*;
- *READINGS, DATA_SIZE, B_ADDRS_BITS, B_TOTAL_ADDRS*: são os parâmetros utilizados para instanciar o componente *MemHandler*. Os parâmetros *DIM1* e *DIM2* do *MemHandler* são equivalentes à quantidade de linhas do *buffer* e quantidade de características, respectivamente. Portanto, ao instanciar o componente *MemHandler*, os parâmetros *DIM1* e *DIM2* recebem os valores definidos em *ROWS* e *CHS*.

O componente *TubularBuffer* contém seis portas, três de entrada e três de saída, além do sinal de *clock,* que foi omitido na Figura 21. Estas portas estão detalhadas a seguir:

- *writeReady* (saída): indicando se o *buffer* está pronto para escrita de novos dados. O único momento em que o *buffer* não pode receber novos dados é quando ele está cheio. Isto é verificado comparando os índices onde o novo dado deve ser inserido com os índices do último dado lido. Caso seja identificado que o recebimento de novos dados vai sobrescrever dados antigos de algum campo receptivo que ainda não tenha sido utilizado em convoluções, o *buffer* sinaliza nesta porta que ainda não está pronto para novas escritas;
- *writeRequest* (entrada): esta porta indica que um novo dado está sendo informado para armazenamento;
- *writeData* (entrada): esta porta recebe os dados que devem ser armazenados. Note que ela é configurada de acordo com o parâmetro WRITE_CH, que determina a quantidade de características que é recebida em cada escrita;
- *clientReady* (entrada): o componente que utiliza este *buffer* indica através desta porta se ele está disponível para o recebimento da leitura de um campo receptivo;
- *readDone* (saída): esta porta sinaliza que uma leitura de um campo receptivo foi realizada;
- *readData* (saída): transmite os dados referentes ao campo receptivo lido. Esta porta também é definida de acordo com a parametrização do componente, pois as informações referentes às dimensões

A escrita no *buffer* ocorre quando é realizada uma solicitação de leitura na porta *writeRequest*. Caso o *buffer* não esteja sinalizando indisponibilidade de escrita na porta *writeReady*, os dados informados na porta *writeData* são armazenados. Caso o *buffer* esteja sinalizando indisponibilidade de escrita, os dados informados na porta *writeData* não são armazenados, mesmo que a escrita seja devidamente notificada na porta *writeRequest*.

Para armazenar os dados, o componente *TubularBuffer* utiliza índices pré-calculados para determinar o posicionamento dos dados no *buffer*. Sempre que ocorre uma nova escrita, estes índices são atualizados. Estes índices são utilizados para definir os bits da matriz de solicitação de leituras, informada na porta *writeRequest* do componente *MemHandler* e determinar o endereço onde os dados devem ser armazenados nas

respectivas memórias. Enquanto é feita a solicitação de escrita para o componente MemHandler, o componente TubularBuffer atualiza os índices que serão utilizados na próxima escrita.

Para realizar a leitura do campo receptivo, o *TubularBuffer* realiza a comparação de índices de leitura com os índices de escrita para identificar se existem dados suficientes recebidos. Caso existam, o *TubularBuffer* utiliza os índices de leitura, que também são pré-calculados, para determinar o endereço de onde os dados devem ser lidos. Este endereço é informado na porta *readAddr* do componente *MemHandler*, junto com uma solicitação de leitura na porta *readRequest*. Simultaneamente, o *TubularBuffer* atualiza os índices de leitura, levando em consideração o *stride* do campo receptivo.

Após a solicitação de leitura ter sido realizada no componente *MemHandler*, este componente realiza uma sequência de leituras em endereços consecutivos de suas memórias, até que se complete a leitura do campo receptivo. Concluída a leitura, o componente *TubularBuffer* recebe uma notificação de conclusão de leitura. Ao receber esta notificação, o *buffer* verifica se a porta de entrada *clientReady* está indicando que o componente utilizador do *buffer* está apto para receber o campo receptivo lido; caso ele esteja, os dados do campo receptivo são transmitidos na porta *readData*. Caso contrário, o *buffer* aguarda até que o componente possa receber o campo receptivo. Enquanto o campo receptivo lido não for transmitido, não são realizadas novas leituras. Quando o *buffer* consegue transmitir o campo receptivo lido, ele volta a realizar as verificações se existem dados suficientes para a leitura de um novo campo receptivo.

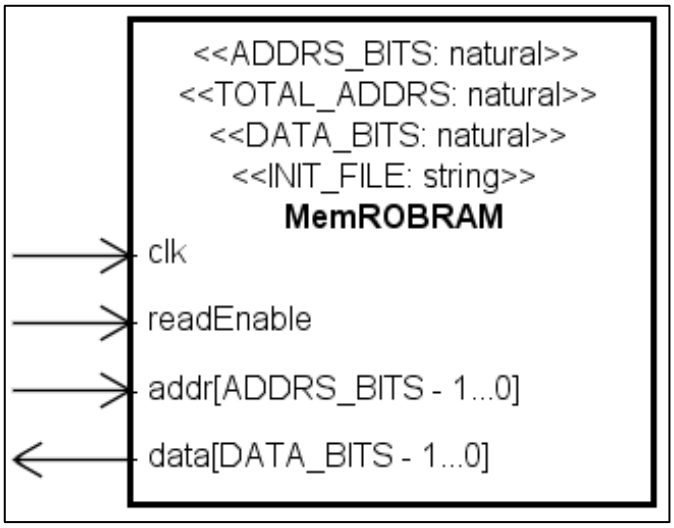
3.6.2. Componente de execução das convoluções

Os componentes que executam as convoluções são os componentes que realizam a várias multiplicações e somas entre os elementos dos *kernels* e dos campos receptivos. Para isto, um dos componentes necessários é o componente que executa o *buffer* tubular, já apresentado anteriormente. Outro componente necessário é o

componente responsável pela leitura dos *kernels* da memória, de acordo com a taxa de utilização de *kernels* em convoluções. Finalmente, um terceiro componente é responsável por realizar as multiplicações e somas, retornando o resultado das extrações de características de uma camada convolucional, sem incluir a operação de *pooling*. Estes dois últimos componentes serão apresentados a seguir.

A leitura dos *kernel* da memória é realizada pelo componente ***MemROBRAM***., ilustrado na Figura 22. Este componente é responsável por realizar as leituras dos dados dos *kernels* das memórias BRAM. Como os *kernels* nunca são atualizados, são apenas lidos, estes componentes são projetados para trabalharem em modo *read-only*.

Figura 22 – Componente para leitura de *kernels* da memória BRAM.



Fonte: Autoria própria (2019).

Conforme ilustrado pela Figura 22, o componente *MemROBRAM* aceita quatro parâmetros para configuração do componente no momento em que ele é instanciado. Estes parâmetros são:

- *TOTAL_ADDRS*: indica quantos endereços devem ser utilizados em cada memória. Este valor deve ser múltiplo da quantidade de *kernels* utilizados na camada em que este componente é utilizado. No caso da primeira camada convolucional, são utilizados 96 *kernels* e todos eles são lidos em um único ciclo de *clock*; portanto, cada *kernel* ocupa um endereço. Consequentemente,

a quantidade de endereços deve ser igual a quantidade de *kernels*. Para o caso das camadas que realizam a leitura de cada *kernel* em quatro ciclos de *clock,* a quantidade de endereços deve ser quatro vezes a quantidade de *kernels*, pois cada *kernel* ocupa quatro endereços diferentes. Em cada ciclo de *clock*, um endereço é lido, resultando na leitura de ¼ de *kernel* por ciclo de *clock*;

- *ADDRS_BITS*: indica a quantidade de bits utilizados em cada endereço;
- *DATA_BITS*: indica a quantidade de bits de dados lidos em cada ciclo de *clock*. Nos casos em que o *kernel* inteiro é lido em um único ciclo de *clock*, este valor deve ser igual a quantidade de bits de cada *kernel.* Nos casos em que o *kernel* é lido em mais de um ciclo de *clock*, este valor é inversamente proporcional à quantidade de ciclos de *clock* necessários para ler cada *kernel*;
- *INIT_FILE*: indica o arquivo contendo os dados dos *kernels* para inicialização da memória. Por se tratar de um componente que realiza apenas leituras, a memória deve ser inicializada com os valores desejados. Portanto, os dados do arquivo são utilizados para realizar a definição inicial do FPGA, inicializando as BRAM com estes valores.

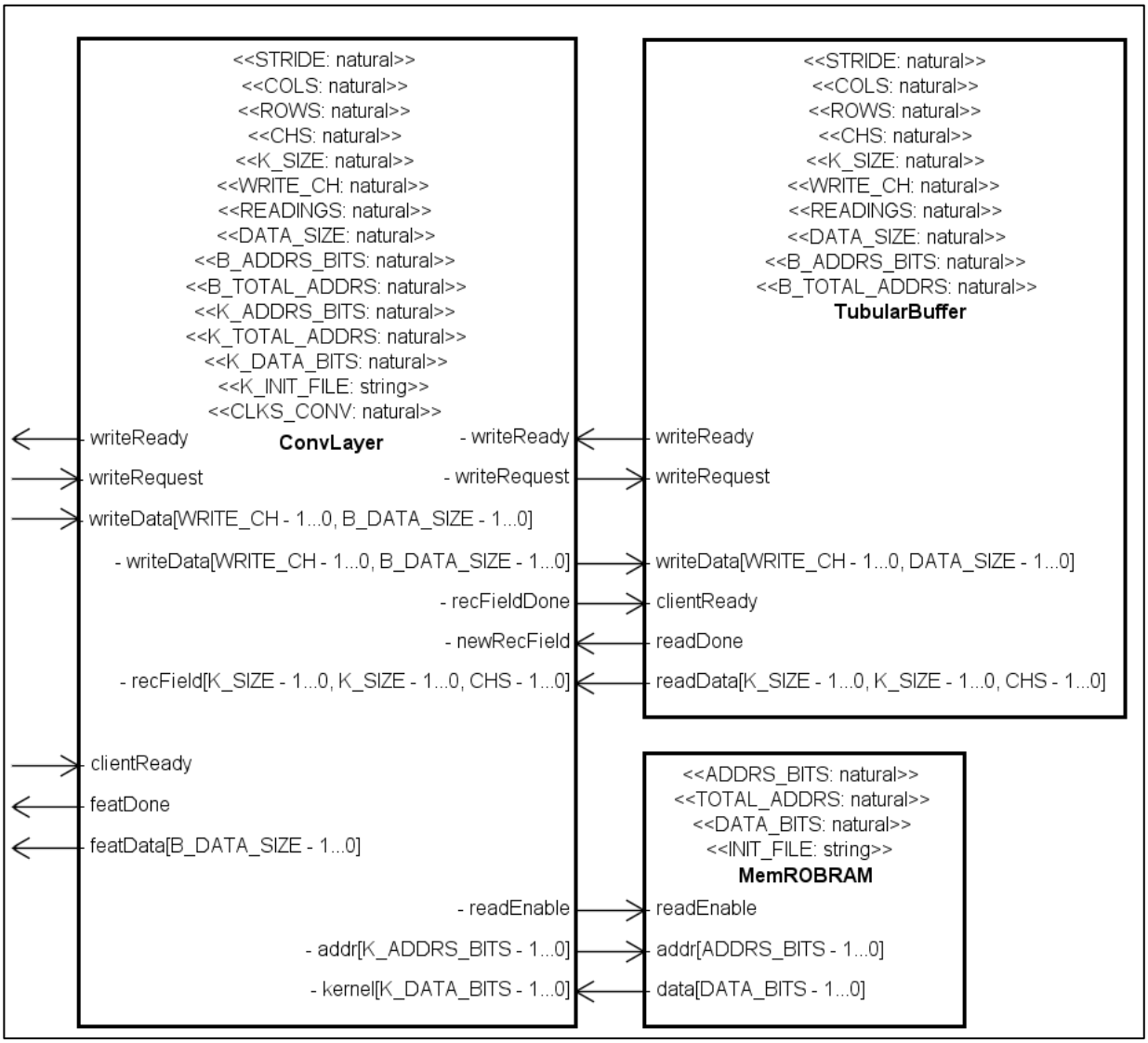
O componente *MemROBRAM* contém quatro portas, sendo três portas de entrada e uma de saída, descritas a seguir:

- *clk* (entrada): porta que recebe o sinal de *clock;*
- *readEnable* (entrada): indica se uma leitura deve ser realizada;
- *addr* (entrada): indica o endereço da leitura que deve ser realizada.
- *data* (saída): porta para transmissão dos dados lidos da memória.

Para realizar a leitura de um *kernel* completo, o componente *MemROBRAM* acessa várias memórias BRAM, pois a largura máxima de leitura suportada em cada memória é de 36 bits. Portanto, conforme descrito na seção 3.4, cada *kernel* é segmentado em várias BRAMs e o componente *MemROBRAM* é responsável por realizar a leitura destas BRAMs simultaneamente para compor um *kernel*.

O componente ***ConvLayer*** é o componente que utiliza os dados do *buffer* tubular e dos *kernels* para executar as convoluções. Este componente está ilustrado na Figura 23.

Figura 23 – Componente de execução de camada convolucional, ***ConvLayer***.



Fonte: Autoria própria (2019).

O componente *ConvLayer* aceita alguns parâmetros, sendo a maioria deles utilizados para instanciar os componentes *MemROBRAM* e *TubularBuffer;* portanto, já foram explicados anteriormente. Para evitar o conflito entre os nomes de parâmetros, alguns deles sofreram alguma alteração. No componente *ConvLayer*, foi acrescentado o prefixo K_ nos parâmetros referentes ao componente *MemROBRAM*. O parâmetro *CLKS_CONV*, indica a quantidade de ciclos de *clock* utilizados em cada convolução.

O componente *ConvLayer* possui seis portas, sendo três delas de entrada e três de saída. Algumas destas portas são conectadas diretamente no componente do *buffer*

pois são utilizadas para a recepção dos dados do mesmo. A porta do sinal de *clock* foi omitida para facilitar a organização do diagrama, portanto, considere que todos os componentes da Figura 23 compartilham o mesmo sinal de *clock*. As portas do componente *ConvLayer* são:

- *writeReady* (saída): esta porta indica que o *buffer* tubular está disponível para recepção de dados.
- *writeRequest* (entrada): sinaliza uma solicitação de escrita no *buffer*.
- *writeData* (entrada): esta porta recebe os dados que devem ser escritos no *buffer*.
- *clienteRead* (entrada): o sinal recebido nesta porta indica se o componente cliente, que recebe os resultados das convoluções executadas neste componente, está apto para a recepção destes resultados.
- *featDone* (saída): o sinal enviado nesta porta indica que uma nova característica foi extraída, ou seja, foram executadas todas as multiplicações e somas entre os elementos do campo receptivo e do *kernel*.
- *featData* (saída): esta porta transmite a característica extraída, portanto, o resultado da convolução.

Para leitura dos *kernels*, o componente *ConvLayer* indica uma solicitação de leitura no sinal *readEnable* e indica o endereço em que o *kernel* deve ser lido. Este endereço equivale a uma contagem de leituras, pois o primeiro *kernel* é armazenado no primeiro endereço e os demais *kernels* ocupam os endereços consecutivos. Nos casos em que as convoluções são executadas em mais de um ciclo de *clock*, o *kernel* é lido parcialmente da memória, pois, sendo segmentado em vários endereços, cada leitura obtém apenas o segmento do *kernel* que será utilizado em multiplicações com campos receptivos.

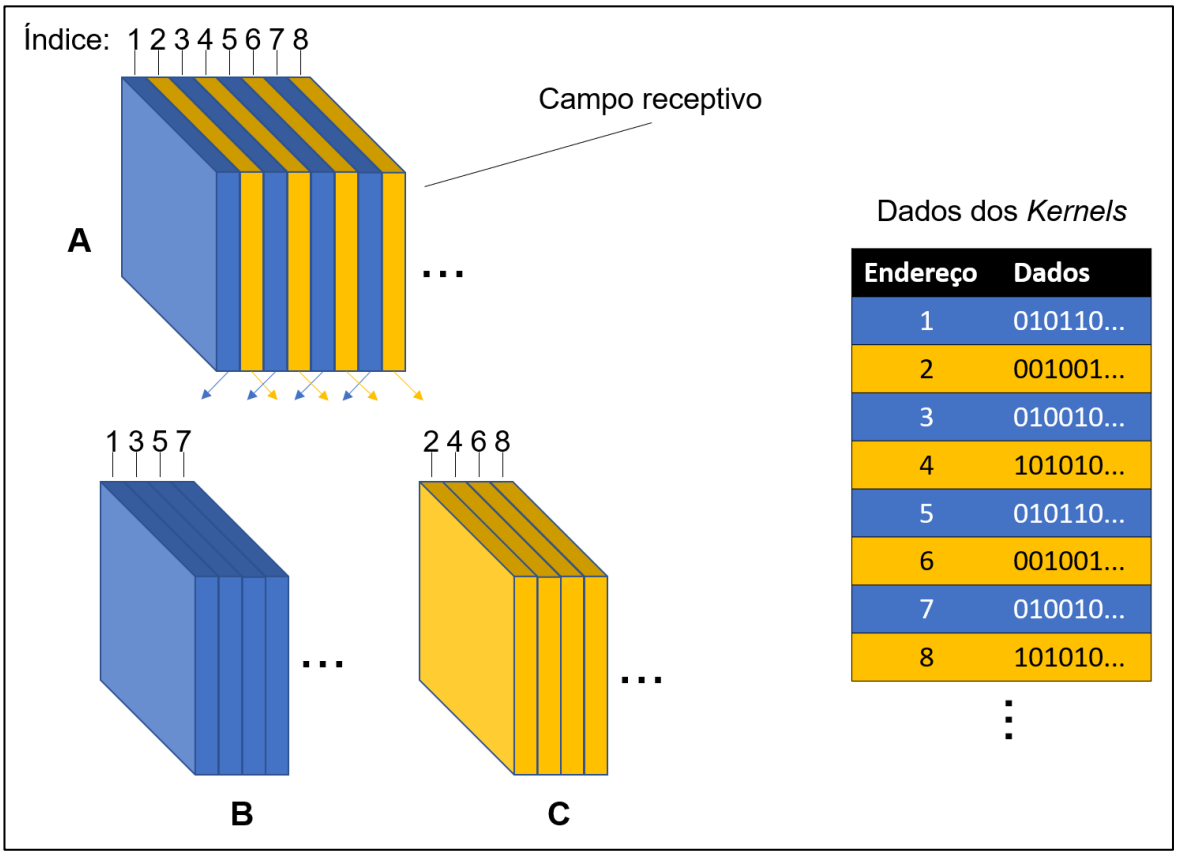
As convoluções dos *kernels* com um campo receptivo são iniciadas no componente *ConvLayer* sempre que um novo campo receptivo é recebido do componente *TubularBuffer*. Este evento é manifestado pela alteração do sinal *newRecField*, equivalente ao sinal *readDone* do componente *TubularBuffer*. Ao iniciar as convoluções, o componente *ConvLayer* fica indisponível para recepção de novos

campos receptivos, indicando a indisponibilidade por meio do sinal *recFieldDone*, que só se altera após todas as convoluções dos *kernels* com o campo receptivo recebido.

Nos casos em que o parâmetro *CLKS_CONV* é maior que um, portanto, cada convolução sendo executada em mais de um ciclo de *clock*, as multiplicações e somas pertinentes ao primeiro ciclo de *clock* da convolução são executadas e o resultado parcial é armazenado para ser somado às demais multiplicações e somas executadas nos ciclos seguintes, até a conclusão da convolução de um *kernel* com o campo receptivo. Após a conclusão, caso a porta *clientReady* indique que o resultado pode ser submetido, é sinalizada na porta *featDone* que existe um novo resultado disponível e o resultado é submetido na porta *featData*. Caso contrário, o componente aguarda pela disponibilidade do componente cliente para envio do resultado.

O componente *ConvLayer* ainda comporta mais um parâmetro que não foi ilustrado nos diagramas. Este parâmetro define a quantidade de segmentos em que o mapa de características deve ser dividido. A arquitetura da AlexNet, ilustrada na Figura 5 (página 29), separa os mapas de características de algumas camadas em duas partes, metade ilustrada na parte superior e a outra metade na parte inferior da Figura 5. Nas camadas que separam o mapa de características em duas partes, este parâmetro deve assumir valor dois, caso contrário, assume valor um. O componente de *buffer* junta os mapas de características das duas partes, intercalando os mapas de características do modelo superior com os mapas do modelo inferior. Portanto, a recuperação dos dados do campo receptivo é obtida pela separação das características, conforme ilustrado na Figura 24.

Figura 24 – Separação das características dos campos receptivos para formação de dois novos campos receptivos. **(A)** campo receptivo original. **(B)** campo receptivo formado pelas características ímpares do campo receptivo A. **(C)** campo receptivo formado pelas características pares do campo receptivo A.



Fonte: Autoria própria (2019).

Na Figura 24, a separação em duas partes é realizada selecionando os índices correspondentes à cada característica que pertencente à cada campo receptivo. Como são dois modelos, superior e inferior, os dados são separados em dois campos receptivos, um deles correspondendo aos mapas do modelo superior e o outro aos do modelo inferior. Na Figura 24, os índices das características do campo receptivo **(A)** são utilizados para selecionar as características referentes aos índices ímpares para formação de um campo receptivo independente **(B)**, enquanto as características dos índices pares são utilizadas para formação de outro campo receptivo **(C)**.

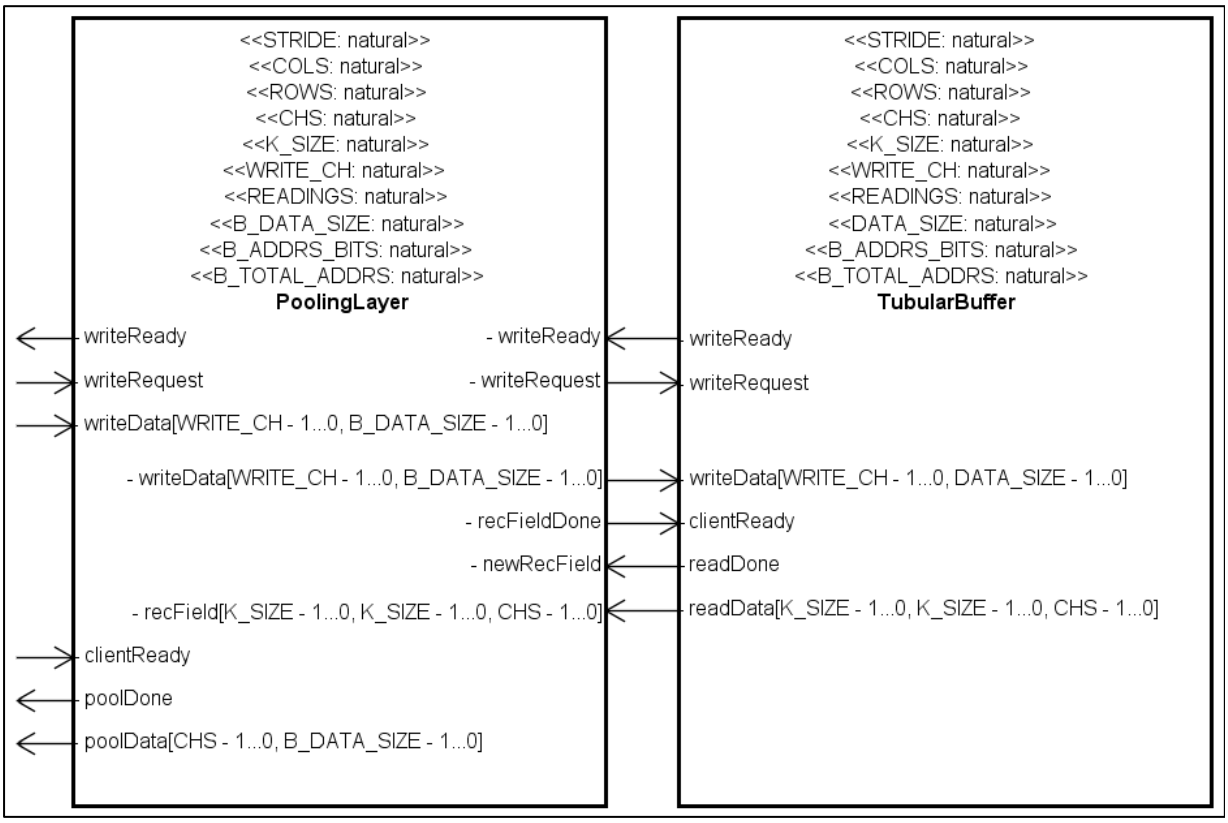
Os *kernels* também devem ser armazenados na memória de maneira intercalada, o *kernel* armazenado no primeiro endereço é utilizado nos campos receptivos formados

por características de índices ímpares, o *kernel* armazenado no endereço seguinte é utilizado nos campos receptivos formados por características de índices pares e assim sucessivamente. Porém, nas camadas que utilizam mais de um ciclo de *clock* por convolução, os *kernels* já são segmentados em mais de um endereço. Portanto, os segmentos de um mesmo *kernel* deverão continuar sendo armazenados em endereços contíguos, pois estes segmentos serão lidos sequencialmente para serem utilizados em convoluções com um mesmo campo receptivo. Após o último segmento de um *kernel* utilizado em um campo receptivo formado por características de índices pares, os segmentos de um *kernel* utilizado em um campo receptivo formado por características de índices ímpares devem ser armazenados.

3.6.3. Componente de Pooling

O componente ***PoolingLayer*** realiza a operação de *pooling*. Ele é muito semelhante ao componente que executa as convoluções em termos de portas, sinais e parâmetros, com a diferença que o componente de *pooling* não necessita de um componente para leitura de *kernels* da memória. Este componente está descrito na Figura 25.

Figura 25 – Componente para execução da operação de *pooling*, *PoolingLayer*.



Fonte: Autoria própria (2019).

O componente *PoolingLayer* se comunica com o componente *TubularBuffer* da mesma maneira que o componente *ConvLayer* para recepção, armazenamento e leitura de lados de campos receptivos. O diferencial deste componente é que, quando um novo campo receptivo é recebido do *buffer*, a operação de *pooling* é realizada em um único ciclo de *clock*. Caso o componente cliente esteja sinalizando disponibilidade através da porta *clientReady,* o resultado já é retornado no mesmo ciclo de *clock* em que a operação é realizada, utilizando a porta *poolData* para dados e a porta *poolDone* para sinalizar a transmissão dos dados.

Uma outra característica do componente *PoolingLayer* é que a porta *poolData*, utilizada para envio do resultado do *pooling*, comporta várias características. Isto é uma consequência da própria natureza da operação de *pooling*, pois esta operação não agrupa dados de mapas de características diferentes, preservando assim a quantidade de características, embora todas as ativações de uma mesma

característica em um campo receptivo tenham sido substituídas por um único valor. Portanto, enquanto a porta *featData*, do componente *ConvLayer*, transmite apenas uma característica, a porta *poolData* transmite um vetor de características, configurando um *hiperpixel* completo.

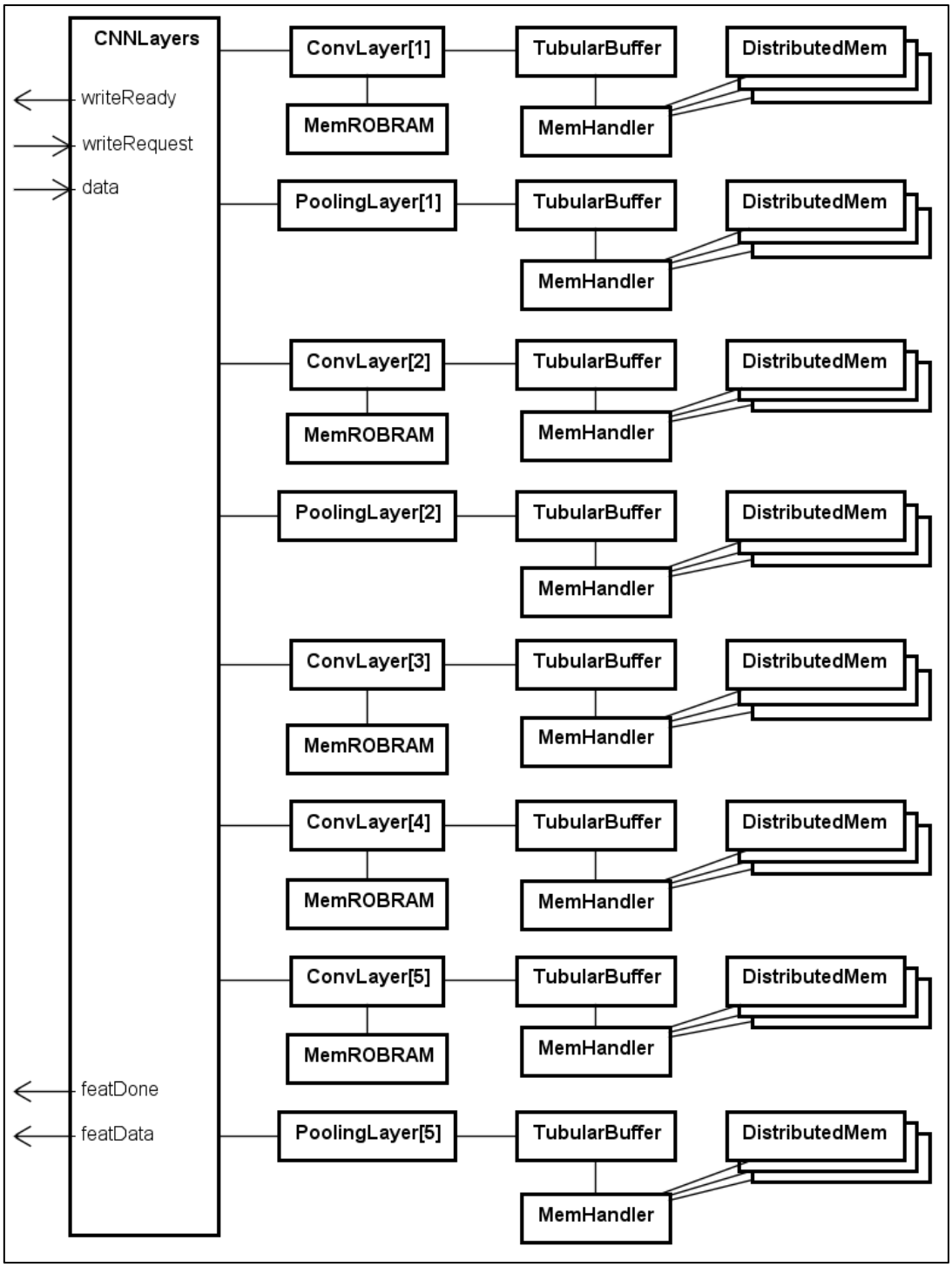
3.6.4. Componente CNN

Finalmente, o componente ***CNNLayers*** é responsável por conectar todos os componentes das camadas convolucionais e de *pooling*. Este componente está ilustrado na Figura 26. Para facilitar a organização, o diagrama da figura omitiu todos os sinais transmitidos entre os componentes, pois a figura também oferece uma visão geral de todos os componentes.

O componente *CNNLayers* é responsável pela recepção e transmissão dos sinais entre as camadas. Ele também é responsável pela recepção dos dados de entrada e encaminhamento dos mesmos para a primeira camada convolucional, assim como a recepção e transmissão dos dados da última camada de *pooling*.

Este componente, por instanciar todos os componentes das camadas convolucionais e de *pooling*, é responsável pela definição de todos os parâmetros de todas as camadas. Portanto, para facilitar a gestão destes parâmetros, foram criados alguns tipos de dados específicos para abstrair as constantes utilizadas na definição destes parâmetros. Estes tipos de dados consistem em conjuntos de propriedades, de acordo com os parâmetros exigidos por cada componente. Desta forma, a definição dos parâmetros de cada componente foi simplificada, pois bastou criar algumas variáveis destes tipos de dados, definindo o valor de cada propriedade, e referenciá-las ao instanciar cada componente.

Figura 26 – Componentes das camadas convolucionais.



Fonte: Autoria própria (2019).

O componente CNNLayers possui cinco portas, sendo duas de entrada e três de saída, além do sinal de *clock* omitido. Estas portas são descritas a seguir:

- *writeReady* (saída): indica se o componente está pronto para o recebimento de dados.
- *writeRequest* (entrada): indica se está sendo feita uma solicitação de escrita.
- *data* (entrada): recebe os dados que devem ser escritos.
- *featDone* (saída): sinaliza que existem dados para serem transmitidos.
- *featData* (saída): porta para transmitir os dados.

As portas *writeReady*, *writeRequest* e *data* são conectadas ao *buffer* tubular da primeira camada convolucional. Portanto, os sinais transmitidos nestas portas servem para comunicação direta com o *buffer*, tanto para o recebimento dos dados utilizados na primeira camada convolucional, quanto para verificar a disponibilidade do *buffer* para recebimento de dados. De maneira semelhante, os sinais featDone e featData, são sinais ligados à saída de dados do componente de *pooling* da última camada de *pooling.*

4. RESULTADOS E DISCUSSÕES

As subseções deste capítulo4.1 apresentas os resultado em relação ao tempo de execução dos componentes projetados para o caso da AlexNet, em seguida apresenta o aspecto da área consumida do disposto FPGA e por último, discorre sobre como estes resultados podem ser afetados com a generalização dos componentes de *hardware* projetado para a execução de outras arquitetura de RNCs.

Os resultados apresentados a seguir foram obtidos por meio de simulações temporais disponibilizadas pela ferramenta Vivado 2018.1 da Xilinx. Estas simulações consideram os tempos de propagação dos sinais entre os elementos que constituem o FPGA; portanto, a simulação temporal é capaz de prever o funcionamento real do sistema projetado.

4.1. TEMPO DE EXECUÇÃO

O tempo para a execução das cinco camadas convolucionais, incluindo as camadas de *pooling*, foi de 3,94 ms, conforme ilustrado na Tabela 5, que também contrasta os resultados com outros três trabalhos. A frequência máxima de operação suportada é de 76,9 MHz.

Tabela 5 – Tempos de execução de CNNs AlexNet em FPGAs.

	A	B	C	Este Trabalho
Tempo em convoluções	21,61 ms	19,86 ms	9,92 ms	3,94 ms
Frequência de operação	100 MHz	193,6 MHz	100 MHz	76,9 MHz
FPGA	Virtex-7 VX485T	Stratix-V GXA7	Stratix-V GXA7	Zynq XCZ100

Fonte: A – (ZHANG et al., 2015); B – (SUDA et al., 2016); C – (MA et al., 2016).

O tempo de execução medido neste trabalho considera o tempo decorrido entre a recepção do primeiro dado de uma imagem, na entrada da primeira camada convolucional, até a transmissão da última característica obtida na última camada de *pooling*. Portanto, este é o tempo total necessário para a execução de todas as convoluções em uma imagem. Este tempo ainda pode ser reduzido ao explorar o paralelismo existente entre as camadas para processar uma sequência de imagens.

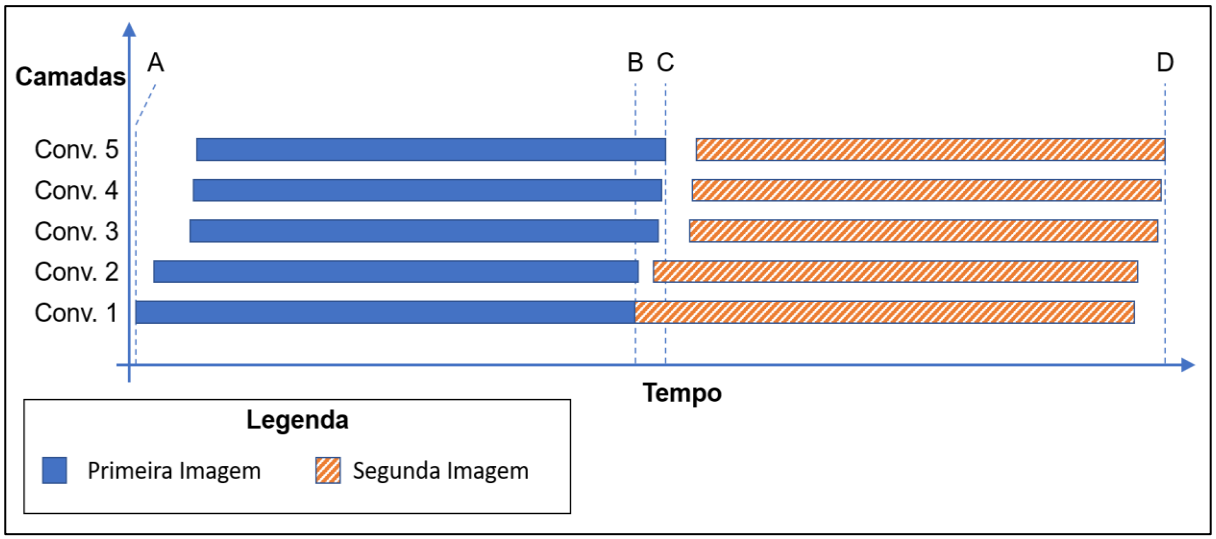
Embora não seja possível diminuir o tempo total decorrido entre o início e o fim da computação das convoluções em uma imagem, é possível diminuir o intervalo entre a conclusão do processamento de cada imagem. Para isso, as convoluções de uma imagem na primeira camada são iniciadas assim que as convoluções da imagem anterior forem concluídas nesta camada. Portanto, não é necessário aguardar os 3,94 ms para conclusão de todas as convoluções em todas as camadas.

A Figura 27 ilustra os instantes de início e término das execuções em cada camada convolucional para duas imagens seguidas. Os instantes da primeira imagem correspondem ao tempo de execução obtido dos experimentos. Os instantes da segunda imagem foram projetados levando em consideração os tempos medidos na primeira imagem. Na Figura 27, (A) representa o início da execução das convoluções na primeira camada convolucional, enquanto (C) representa o término de todas as

convoluções da primeira imagem. Portanto, a diferença de tempo entre o instante (A) até o instante (C) é de 3,94 ms, pois este é o tempo necessário para a execução de todas as convoluções em uma única imagem. Note que, na Figura 27, as convoluções da segunda imagem iniciam no instante (B), pois não é necessário aguardar o instante (C) para iniciar as convoluções, dado que as camadas são executas paralelamente no FPGA.

Iniciar as convoluções da segunda imagem antes do término das convoluções da primeira faz com que durante um período de tempo, de aproximadamente 0,22 ms, exista uma sobreposição de duas imagens sendo processadas. Consequentemente, é possível concluir as convoluções de uma imagem a cada 3,72 ms, resultando em uma taxa máxima de processamento de 268,82 imagens por segundo.

Figura 27 – Tempo de execução, em escala, de cada camada convolucional. (A) indica o início das convoluções da primeira imagem na primeira camada convolucional. (B) indica o término das convoluções da primeira imagem e início das convoluções da segunda imagem na primeira camada convolucional. (C) indica o término das convoluções da primeira imagem na quinta camada convolucional. (D) indica o término das convoluções da segunda imagem na última camada convolucional.



Fonte: Autoria própria (2019)

A Figura 27 também apresenta o primeiro indício de que o balanceamento do *pipeline* está adequado, pois indica que o tempo de execução das convoluções em cada camada é relativamente semelhante. A Tabela 6 mostra o tempo necessário para a execução de todas as convoluções em cada camada. Portanto, o comprimento das barras da Figura 27 equivale aos valores da última coluna da Tabela 6.

Tabela 6 – Instantes de início e fim da execução das convoluções em cada camada. Tempo necessário para conclusão das convoluções.

Camada	Instante de início da primeira convolução (ms)	Instante de término da última convolução (ms)	Tempo de execução (ms)
Conv. 1	0,019	3,742	3,723
Conv. 2	0,158	3,765	3,606
Conv. 3	0,439	3,929	3,490
Conv. 4	0,457	3,947	3,490
Conv. 5	0,476	3,966	3,490

Na Tabela 6, é possível notar que a primeira convolução da primeira camada ocorre somente no instante 0,019 ms, pois existe um tempo necessário para o acúmulo dos dados no *buffer* até que se forme o primeiro campo receptivo. Após a formação deste primeiro campo receptivo, a sequência de convoluções na primeira camada é iniciada. Uma análise parecida também pode ser feita para o início das convoluções nas demais camadas convolucionais, pois estas camadas também dependem do acúmulo de dados nos seus respectivos *buffers* para o início das suas convoluções.

Analisando o instante em que ocorrem as primeiras convoluções em cada camada, é possível notar que as camadas dois e três demandam uma quantidade de tempo relativamente maior que as demais camadas para iniciarem a primeira convolução A primeira convolução na segunda camada ocorre 0,12 ms após a primeira convolução na primeira camada, esta diferença de tempo entre a terceira e segunda camada é de 0,237 ms. Nas demais camadas esta diferença não passa de 0,015 ms. Esta demora para início das convoluções na segunda e terceira camada está relacionada à quantidade de convoluções necessárias para o acúmulo de dados que formam o

primeiro campo receptivo destas camadas. A Tabela 7 mostra as dimensões dos mapas de características gerados em cada camada e a quantidade de ativações contidas nestes mapas. A quantidade de ativações é equivalente à quantidade de convoluções, pois cada ativação é gerada por uma convolução.

Tabela 7 – Dimensões dos mapas de características extraídos pelas camadas convolucionais.

Camada	Dimensões do mapa de característica extraído (altura x largura x características)	Quantidade de ativações no mapa de características extraído
Conv. 1	55 x 55 x 96	290.400
Conv. 2	27 x 27 x 256	186.624
Conv. 3	13 x 13 x 384	64.896
Conv. 4	13 x 13 x 384	64.896
Conv. 5	13 x 13 x 256	43.264

Os mapas de características extraídos pelas camadas um e dois são os que contém maior número de ativações, portanto, demandam maior quantidade de convoluções. Consequentemente, para acumular as primeiras linhas dos mapas de características que formam o primeiro campo receptivo, utilizado pela camada dois e três, é necessário que seja realizado um número significativo de convoluções.

A Tabela 7 também destaca a heterogeneidade existente entre as camadas convolucionais. As primeiras camadas convolucionais extraem mapas de características maiores e menos profundos, enquanto as últimas extraem mapas de características menores e mais profundos, com exceção da última camada, que tem mapas com a mesma profundidade dos mapas da segunda camada. A parametrização dos componentes permite que eles sejam utilizados para todas as camadas, mesmo que elas tenham atributos diferentes, como quantidade de

características utilizadas, dimensões dos mapas extraídos ou quantidade de *kernels* diferentes.

4.2. ÁREA UTILIZADA

A síntese dos componentes projetados para execução das camadas convolucionais da AlexNet resultou no consumo de quase 100% dos recursos do FPGA. Ao todo, foi utilizado 97% das LUTs, 93% das BRAMs e 87% dos DSPs disponíveis no dispositivo FPGA utilizado.

A reutilização de endereços de memória realizada pelo *buffer* tubular permitiu a execução das camadas convolucionais sem a necessidade do armazenamento completo dos mapas de características. Portando, este armazenamento parcial dos mapas de características das camadas convolucionais e de *pooling* reduziu a necessidade de armazenamento interno do dispositivo em torno de 88,31% em relação ao armazenamento completo de todos os mapas de características, conforme ilustrado pela Tabela 8.

Tabela 8 – Redução de memória devido ao reuso de dados do buffer.

Camadas	Tamanho do Mapa de Características	Campo Receptivo	Palavras no Mapa de Características	Palavras Armazenadas	Redução do Armazenamento
Conv. 1	224 x 224 x 3	11 x 11 x 3	150.528	7.392	95,09%
Pooling 1	55 x 55 x 96	3 x 3 x 1	290.400	15.840	94,55%
Conv. 2	27 x 27 x 96	5 x 5 x 48	69.984	12.960	81,48%
Pooling 2	27 x 27 x 256	3 x 3 x 1	186.624	20.736	88,89%
Conv. 3	13 x 13 x 256	3 x 3 x 256	43.264	9.984	76,92%
Conv. 4	13 x 13 x 384	3 x 3 x 192	64.896	14.976	76,92%
Conv. 5	13 x 13 x 384	3 x 3 x 192	64.896	14.976	76,92%
Pooling 5	13 x 13 x 256	3 x 3 x 1	43.264	9.984	76,92%
Total	-	-	913.856	106.848	88,31%

Na Tabela 8, a coluna que descreve a quantidade de palavras armazenadas é equivalente à capacidade de armazenamento total do *buffer*. Este valor leva em consideração que os *buffers* armazenam uma quantidade de linhas do mapa de características equivalente à quantidade de linhas do campo receptivo. Portanto, para o caso da primeira camada convolucional, o *buffer* é capaz de armazenar 11 linhas, 224 colunas e 3 características, possibilitando o armazenamento de até 7.392 palavras. Estas 7.392 palavras equivalem a aproximadamente 5% das 150.528 palavras que formam o mapa de características de 224 colunas, 224 linhas e 3 características.

4.1. GENERALIZAÇÕES PARA OUTRAS ARQUITETURAS E DISPOSITIVOS FPGA

A parametrização dos componentes projetados, além de facilitar o reaproveitamento dos mesmos componentes para a execução das diferentes camadas da AlexNet, também possibilita a reutilização destes componentes para execução de outras arquiteturas de RNCs que também sejam formadas por encadeamento de camadas convolucionais e de *pooling*.

Os parâmetros dos componentes também podem ser utilizados para modificar a quantidade de computação executada em paralelo. É possível reduzir a quantidade de operações executadas em paralelo, aumentando a quantidade de ciclos de *clock* utilizados por camada e, consequentemente exigindo menos recursos do FPGA. Isto pode ser feito para utilização de dispositivos FPGAs com menos recursos ou liberação de recursos do FPGA para execução de outros sistemas. Também é possível aumentar a quantidade de convoluções realizadas em paralelo, caso o projeto utilize um dispositivo FPGA com mais recursos.

Dispositivos FPGA com menos recursos muitas vezes implicam também menor capacidade de armazenamento. Portanto, a utilização de dispositivos FPGA com menos recursos pode ser limitada pela insuficiência de armazenamento destes

dispositivos, dada a demanda por armazenamento das RNCs. Alternativamente, estes dispositivos podem ser utilizados para executar outras arquiteturas menores de RNCs, que tenham sido formuladas para problemas particulares, que tratem problemas menos complexos que o proposto pelo *benchmark* ILSVRC.

Em contrapartida, a utilização de dispositivos FPGAs com maior capacidade pode possibilitar a execução de RNCs maiores, com mais parâmetros e melhor desempenho de classificação que a AlexNet, ou possibilitar a execução da AlexNet com maior taxa de processamento de imagens por segundo, aumentando a quantidade de convoluções com ciclo de *clock*.

5. CONCLUSÃO

Este trabalho desenvolveu um método para execução de redes neurais convolucionais em FPGAs. A RNC AlexNet foi utilizada como exemplo para executar os experimentos.

Com base na pesquisa bibliográfica, foi concluído que um dos desafios da execução de RNCs em FPGAs é o armazenamento dos parâmetros da rede, sendo que os parâmetros das camadas completamente conectadas, que atuam como classificador da RNC, demandam a maior parte da capacidade de armazenamento. Porém, os parâmetros das camadas convolucionais, mesmo demandando menor capacidade de armazenamento, exigem um grande *throughput* de leitura, pois estes parâmetros são referentes aos dados dos *kernels*, que são lidos recorrentemente da memória durante as execuções das convoluções.

Este trabalho utilizou a memória interna BRAM do FPGA e representação numérica de 8 bits para armazenar os parâmetros das camadas convolucionais. A adoção desta precisão numérica, de acordo com a pesquisa bibliográfica, teve um impacto na medida de erro top-5 da RNC de menos de 1%. Portanto, a adoção desta precisão numérica trouxe grande benefício para execução de RNCs em FPGAs; neste trabalho, o armazenamento de todos os parâmetros das camadas convolucionais da AlexNet na memória interna do FPGA (modelo XC7Z100 da Xilinx) foi possível e ocupou aproximadamente 93% das BRAMs disponíveis no dispositivo.

Com todos os parâmetros das camadas convolucionais armazenados na memória interna, foi possível realizar a leitura simultânea de vários BRAMs para carregar os *kernels* com alto *throughput*.

Este trabalho também apresentou um método para balanceamento do *pipeline* formado pelas camadas convolucionais. Este balanceamento permitiu que os recursos do FPGA fossem direcionados para a execução das camadas convolucionais que executam mais multiplicações, que no caso da AlexNet, são as primeiras camadas.

Finalmente, também foram apresentados os componentes de *hardware* projetados para executar as camadas convolucionais, validando a viabilidade do método apresentado.

O método utilizado para balanceamento do *pipeline* foi divulgado em apresentação oral no congresso 27th *International Conference on Artificial Neural Networks* (ICANN) (SOUSA; MIGUEL ANGELO DE ABREU DE SOUSA; EMÍLIO DEL-MORAL-HERNANDEZ, 2018), com o resumo disponível neste documento, no Apêndice B – Resumo do Artigo Publicado.

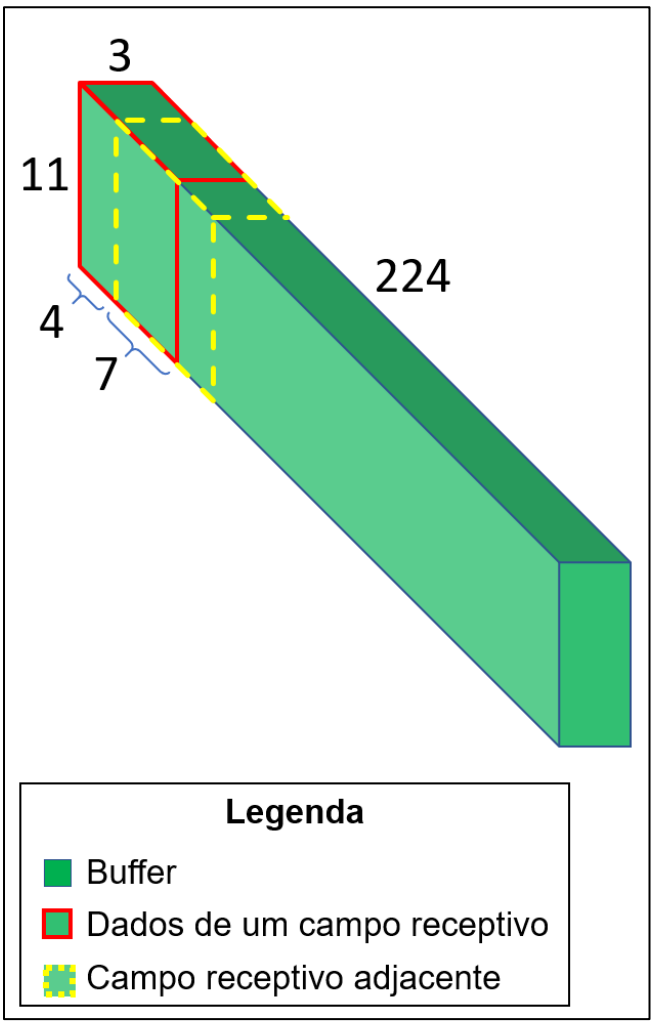
6. EXTENSÕES DESTA PESQUISA E SUGESTÕES PARA TRABALHOS FUTUROS

6.1. MELHORIAS DO *BUFFER* TUBULAR

Apesar do *buffer* tubular como desenvolvido neste trabalho armazenar os dados de maneira eficiente, a leitura destes dados do *buffer* poderia ser melhorada. As leituras dos campos receptivos poderiam ler menos dados da memória ao reaproveitar a região de sobreposição entre os campos receptivos adjacentes.

Utilizando como exemplo o *buffer* da primeira camada convolucional, que utiliza campos receptivos de 11 linhas, 11 colunas, 3 características e *stride* quatro, existe uma região de sobreposição entre os campos receptivos de sete colunas, conforme ilustrado pela Figura 28. Dos 363 dados que formam um campo receptivo, com esta sobreposição destas sete colunas, seria necessário a leitura apenas da região onde não ocorre a sobreposição no campo receptivo adjacente. Esta região sem sobreposição tem dimensão 11 x 4 x 3, portanto, resultaria na leitura de 132 dados. A leitura apenas destes 132 dados resultaria na leitura de 36,36% do campo receptivo, reaproveitando os outros 63,63% de sobreposição dos dados que já foram lidos.

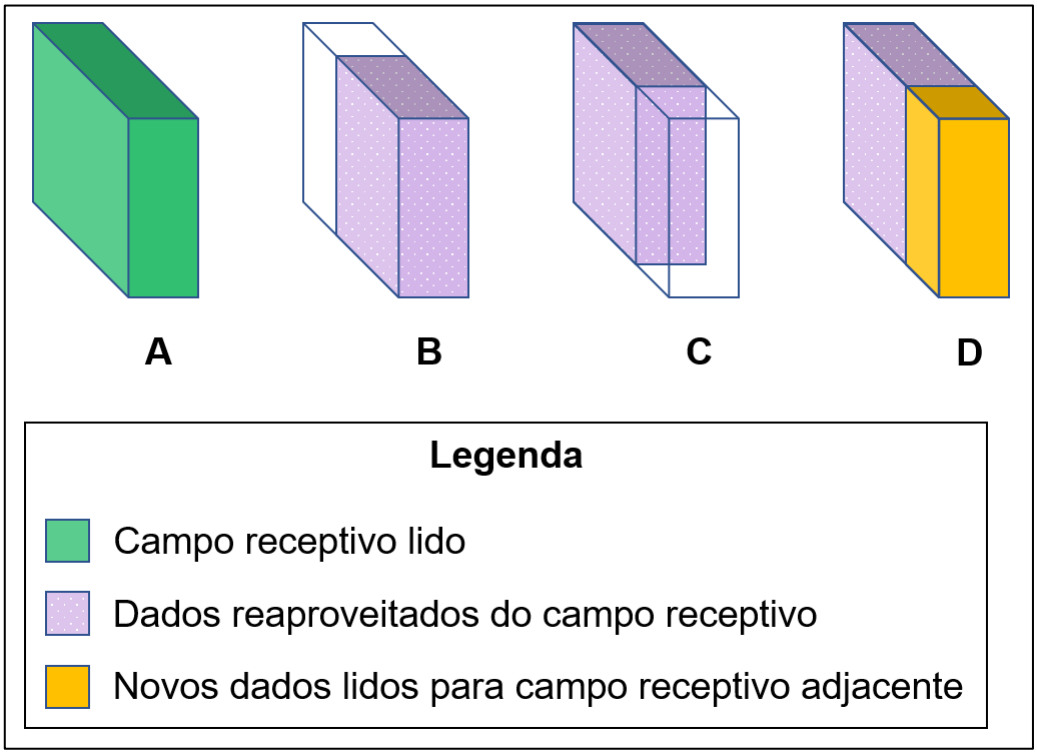
Figura 28 – Sobreposição entre campos receptivos adjacentes.



Fonte: Autoria própria (2019).

Porém, o mecanismo para realizar o aproveitamento desta região onde ocorre a sobreposição precisaria realizar dois tratamentos adicionais. O primeiro deles seria o tratamento do caso excepcional das regiões limítrofes do *buffer*, pois não ocorre sobreposição de dados entre o campo receptivo formado pelas últimas colunas e o campo receptivo formado pelas primeiras colunas do mapa de características. Portanto, após a leitura dos campos receptivos formados pelas últimas colunas do *buffer*, seria necessário realizar uma leitura completa do campo receptivo seguinte, pois este campo receptivo seria formado pelas primeiras colunas. O outro tratamento adicional seria o reposicionamento dos dados reaproveitados no campo receptivo ilustrado pela Figura 29.

Figura 29 – Reaproveitamento de dados entre campos receptivos. **(A)** campo receptivo lido. **(B)** dados reaproveitados da região de sobreposição entre campos receptivos. **(C)** reposicionamento dos dados reaproveitados. **(D)** leitura de dados adicionais para completar o campo receptivo adjacente.



Fonte: Autoria própria (2019).

A Figura 29 ilustra que dois campos receptivos adjacentes podem reaproveitar dados que já foram lidos do primeiro campo receptivo (A). Os dados reaproveitados formam uma região de sobreposição entre dois campos receptivos, abrangendo as últimas colunas (B) do primeiro campo receptivo e também as primeiras colunas do campo receptivo seguinte. Para reaproveitar estes dados, é necessário que eles sejam reposicionados (C) e mais dados sejam lidos para formar o campo receptivo adjacente (D). Portanto, este reposicionamento de dados também deve ser levado em consideração.

Estes dois tratamentos adicionais apresentados podem exigir que mais recursos do FPGA sejam utilizados, contrabalanceando os benefícios trazidos pela redução da

quantidade de leituras realizadas. Novos experimentos devem ser realizados para verificar os benefícios em termos de redução de tempo de execução e a quantidade de recursos adicionais consumidos por estes dois mecanismos introduzidos.

Um outro possível ponto de melhoria é a maneira como o componente de *buffer* gerencia os dados. Este componente calcula e compara os índices em uma matriz tridimensional, referentes aos dados escritos e lidos dos mapas de características. Este cálculo é feito em função das dimensões dos mapas de características, campo receptivo e *stride.* Como existe uma quantidade finita de possíveis estados do *buffer*, uma alternativa seria calcular estes índices previamente e armazená-los em memória para serem lidos, ao invés de calculados. Nesta mesma linha, a verificação da suficiência de dados no *buffer* também pode ser calculada previamente, resultando em um mapeamento entre quantidade de dados recebidos e número de campos receptivos formados. Dois contadores seriam introduzidos, um para a quantidade de dados recebidos e outro para campos receptivos formados. O resultado destes contadores seria utilizado para consultar o mapeamento que indicaria se um novo campo receptivo foi formado, em função da quantidade de dados recebidos. Portanto, estas duas melhorias reduziriam a necessidade de componentes de *hardware* para lógica e aumentariam a demanda por memória, sendo necessária a realização de mais experimentos para verificar os benefícios e prejuízos destas mudanças.

6.2. EXPERIMENTOS EM FPGAS COM MEMÓRIA DRAM EMBUTIDA

Uma das novidades introduzidas por dispositivos FPGAs mais modernos é a inclusão de memórias *High Bandwidth Memory (HBM) Dynamic Random-Access Memory* (DRAM) (WISSOLIK et al., 2017) no mesmo substrato de silício do FPGA. Atualmente (janeiro de 2019), estas memórias suportam o armazenamento de até 8 GB de dados, um aumento significativo em relação às centenas de MB suportados pelas memórias BRAMs dos FPGA, com largura de banda de até 460 GB/s.

A inclusão destas memórias no mesmo substrato de silício aumenta significativamente a vazão de leitura em relação às memórias externas. Portanto, esta novidade é uma

potencial solução para o gargalo do acesso às memórias externas existente na execução de RNCs em FPGAs.

Estas memórias utilizam protocolos de acesso específicos, diferentes de como são acessadas as memórias BRAMs do FPGA. Portanto, os componentes projetados neste trabalho precisariam ser readaptados para utilizarem este tipo de memória.

6.3. OTIMIZAÇÕES DE ÁREA E TEMPO DE LATÊNCIA

Os componentes projetados neste trabalho podem ser melhorados em alguns aspectos. Umas das melhorias que poderiam trazer benefícios em relação ao tempo de execução é uma análise do tempo de latência dos componentes projetados. O tempo de latência determina o menor período que um ciclo de *clock* deve ter para que ocorra a estabilização dos elementos do FPGA, durante a execução de uma operação. Portanto, a diminuição do tempo de latência permitiria a adoção de períodos de *clock* menores, consequentemente, possibilitando aumentar a frequência de operação sem comprometer a integridade das operações.

A diminuição do ciclo de *clock* também está diretamente ligada ao tempo de execução da RNC em *hardware*, pois esta execução demanda uma quantidade fixa de ciclos de *clock* e, com o encurtamento do ciclo de *clock*, o tempo total de execução da RNC também é diminuído. Porém, uma das consequências do aumento da frequência de operação é o aumento do consumo de energia.

6.4. ANÁLISE DO CONSUMO ENERGÉTICO DO DISPOSITIVO

Finalmente, uma informação que pode agregar valor ao trabalho é o consumo de energia necessário para a execução da AlexNet utilizando os componentes projetados. Como a frequência máxima de operação de 76,9 MHz é menor do que a

frequência de operação reportada nos trabalhos comparados na Tabela 5, que variam entre 100 MHz e 193,6 MHZ, é possível que o sistema apresentado neste trabalho também seja mais eficiente em relação ao consumo de energia. Porém, isso só pode ser comprovado com análises complementares dos experimentos realizados.

REFERÊNCIAS

ADHIANTO, L.; BANERJEE, S.; FAGAN, M.; KRENTEL, M.; MARIN, G.; MELLOR-CRUMMEY, J.; TALLENT, N. R.; QIAO, Y.; SHEN, J.; XIAO, T.; YANG, Q.; WEN, M.; ZHANG, C. FPGA-Accelerated Deep Convolutional Neural Networks for High Throughput and Energy Efficiency. **Concurrency Computation Practice and Experience**, v. 22, n. 6, p. 685–701, 2016. Disponível em: <http://doi.wiley.com/10.1002/cpe.3850>. Acesso em: 20 feb. 2017.

ALOM, M. Z.; TAHA, T. M.; YAKOPCIC, C.; WESTBERG, S.; HASAN, M.; VAN ESESN, B. C.; AWWAL, A. A. S.; ASARI, V. K. The History Began from AlexNet: A Comprehensive Survey on Deep Learning Approaches. **Preprint arxiv.org/pdf/1803.01164**, 2018. Disponível em: <http://arxiv.org/abs/1803.01164>.

ARIF, S.; LAL, R. K. Design and Performance Analysis of Various Adder and Multiplier Circuits Using VHDL. **International Journal of Applied Engineering Research**, v. 10, n. 20, p. 17016–17020, 2015.

AYDONAT, U.; O'CONNELL, S.; CAPALIJA, D.; LING, A. C.; CHIU, G. R. An OpenCL(TM) Deep Learning Accelerator on Arria 10. **Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays**, p. 55–64, 2017. Disponível em: <http://dl.acm.org/citation.cfm?doid=3020078.3021738>. Acesso em: 20 feb. 2017.

BEAR, M. F.; CONNORS, B. W.; PARADISO, M. A. **Neuroscience : exploring the brain**. 4. ed. Philadelphia: Wolters Kluwer, 2016.

BENGIO, Y. Learning Deep Architectures for AI. **Foundations and Trends® in Machine Learning**, v. 2, n. 1, p. 1–127, 2009. Disponível em: <http://www.nowpublishers.com/article/Details/MAL-006>. Acesso em: 2 apr. 2017.

BOUREAU, Y. L.; PONCE, J.; LECUN, Y. A Theoretical Analysis of Feature Pooling in Visual Recognition. **27th International Conference on Machine Learning**, p. 111–118, 2010.

CHAKRADHAR, S.; SANKARADAS, M.; JAKKULA, V.; CADAMBI, S. A Dynamically

Configurable Coprocessor for Convolutional Neural Networks. **ACM SIGARCH Computer Architecture News**, v. 38, n. 3, p. 247, 2010. Disponível em: <http://dl.acm.org/citation.cfm?id=1815993>. Acesso em: 11 apr. 2017.

CHI, P.; LI, S.; XU, C.; ZHANG, T.; ZHAO, J.; LIU, Y.; WANG, Y.; XIE, Y. PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory. In: 43rd International Symposium on Computer Architecture, ISCA, **Anais**...IEEE, Jun. 2016. Disponível em: <http://ieeexplore.ieee.org/document/7551380/>. Acesso em: 20 feb. 2017.

CIRE, D. C.; MEIER, U.; MASCI, J.; GAMBARDELLA, L. M.; SCHMIDHUBER, J. High-Performance Neural Networks for Visual Object Classification. **Preprint arxiv.org/abs/1102.0183**, 2011. Disponível em: <https://arxiv.org/pdf/1102.0183>.

COURBARIAUX, M.; BENGIO, Y.; DAVID, J.-P. Training Deep Neural Networks with Low Precision Multiplications. **Preprint arxiv.org/abs/1412.7024**, p. 1–10, 2014. Disponível em: <http://arxiv.org/abs/1412.7024>.

CYBENKO, G. Approximation by Superpositions of a Sigmoidal Function. **Mathematics of Control, Signals, and Systems (MCSS)**, v. 2, n. 4, p. 303–314, 1989. Disponível em: <http://www.springerlink.com/index/N873J15736072427.pdf>. Acesso em: 2 jun. 2017.

DETTMERS, T. 8-Bit Approximations for Parallelism in Deep Learning. **Preprint arxiv.org/abs/1511.04561**, n. 2, p. 1–14, 2015. Disponível em: <http://arxiv.org/abs/1511.04561>.

DI CECCO, R.; LACEY, G.; VASILJEVIC, J.; CHOW, P.; TAYLOR, G.; AREIBI, S. Caffeinated FPGAs: FPGA framework for convolutional neural networks. In: International Conference on Field-Programmable Technology, FPT 2016, **Anais**...2017. Disponível em: <https://arxiv.org/abs/1609.09671>. Acesso em: 22 feb. 2017.

DIGIKEY. **Embedded - FPGAs (Field Programmable Gate Array) | Integrated Circuits (ICs) | DigiKey**. Disponível em: <https://www.digikey.com/products/en?keywords=xcvu13p>. Acesso em: 3 aug. 2018.

ESSER, S. K.; MEROLLA, P. A.; ARTHUR, J. V; CASSIDY, A. S.; APPUSWAMY, R.; ANDREOPOULOS, A.; BERG, D. J.; MCKINSTRY, J. L.; MELANO, T.; BARCH, D. R.; DI NOLFO, C.; DATTA, P.; AMIR, A.; TABA, B.; FLICKNER, M. D.; MODHA, D. S. Convolutional Networks for Fast, Energy-Efficient Neuromorphic Computing. **Proceedings of the National Academy of Sciences**, v. 113, n. 41, p. 11441–11446, 2016. Disponível em: <http://www.pnas.org/content/113/41/11441.abstract>.

FARABET, C.; MARTINI, B.; AKSELROD, P.; TALAY, S.; LECUN, Y.; CULURCIELLO, E. Hardware Accelerated Convolutional Neural Networks for Synthetic Vision Systems. **Proceedings of 2010 IEEE International Symposium on Circuits and Systems**, p. 257–260, 2010. Disponível em: <http://ieeexplore.ieee.org/document/5537908/>.

FARABET, C.; POULET, C.; HAN, J. Y.; LECUN, Y. CNP: An FPGA-based processor for Convolutional Networks. In: 19th International Conference on Field Programmable Logic and Applications, 1, **Anais**...IEEE, 2009. Disponível em: <http://ieeexplore.ieee.org/document/5272559/>. Acesso em: 11 apr. 2017.

FUKUSHIMA, K. Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position. **Biological Cybernetics**, v. 36, n. 4, p. 193–202, 1980.

GLOROT, X.; BORDES, A.; BENGIO, Y. Deep Sparse Rectifier Neural Networks. **Proceedings of the 14th International Conference on Artificial Intelligence and Statistics**, v. 15, p. 315–323, 2011. Disponível em: <http://www.jmlr.org/proceedings/papers/v15/glorot11a/glorot11a.pdf>. Acesso em: 2 jun. 2017.

GOKHALE, V.; JIN, J.; DUNDAR, A.; MARTINI, B.; CULURCIELLO, E. A 240 G-ops/s Mobile Coprocessor for Deep Neural Networks. In: IEEE Conference on Computer Vision and Pattern Recognition Workshops, **Anais**...IEEE, Jun. 2014. Disponível em: <http://ieeexplore.ieee.org/document/6910056/>. Acesso em: 26 sep. 2017.

GUPTA, S.; AGRAWAL, A.; GOPALAKRISHNAN, K.; NARAYANAN, P. Deep Learning with Limited Numerical Precision. In: 32nd International Conference on Machine Learning, PMLR 37, **Anais**...2015. Disponível em: <http://arxiv.org/abs/1502.02551>.

HAN, S.; POOL, J.; TRAN, J.; DALLY, W. J. Learning both Weights and Connections for Efficient Neural Networks. In: 28th International Conference on Neural Information Processing Systems, NIPS 15, **Anais**...2015. Disponível em: <http://arxiv.org/abs/1506.02626>.

HAYKIN, S. **Neural Networks and Learning Machines**. 3. ed. Upper Saddle River: Pearson Education, 2008.

HE, K.; ZHANG, X.; REN, S.; SUN, J. Deep Residual Learning for Image Recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition Deep, **Anais**...2016. Disponível em: <http://arxiv.org/abs/1703.10722>.

HUBEL, D. H.; WIESEL, T. N. Receptive Fields, Binocular Interaction and Functional Architecture in the Cat's Visual Cortex. **Journal of Physiology**, v. 160, n. 1, p. 106–154.2, 1962.

KRIZHEVSKY, A. One Weird Trick for Parallelizing Convolutional Neural Networks. **Preprint**, 2014. Disponível em: <http://arxiv.org/abs/1404.5997>.

KRIZHEVSKY, A.; HINTON, G. E.; SUTSKEVER, I.; GEOFFREY, H.; HINTON, G. E. ImageNet Classification with Deep Convolutional Neural Networks. **Advances in Neural Information Processing Systems 25 (NIPS2012)**, p. 1–9, 2012. Disponível em: <http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks>. Acesso em: 6 apr. 2017.

KUON, I.; TESSIER, R.; ROSE, J. **FPGA Architecture: Survey and Challenges**. Hanover: Now Publishers Inc, 2007. v. 2

LECUN, Y.; BOSER, B.; DENKER, J. S.; HENDERSON, D.; HOWARD, R. E.; HUBBARD, W.; JACKEL, L. D. Backpropagation Applied to Handwritten Zip Code Recognition. **Neural computation**, v. 1, n. 4, p. 541–551, 1989. Disponível em: <http://www.mitpressjournals.org/doi/abs/10.1162/neco.1989.1.4.541>. Acesso em: 17 apr. 2017.

LECUN, Y.; BOTTOU, L.; BENGIO, Y.; HAFFNER, P. Gradient-Based Learning Applied to Document Recognition. **Proceedings of the IEEE**, v. 86, n. 11, p. 2278–2323, 1998. Disponível em: <http://ieeexplore.ieee.org/abstract/document/726791/>.

Acesso em: 11 apr. 2017.

LIN, M.; CHEN, Q.; YAN, S. Network In Network. In: 2nd International Conference on Learning Representations, ICLR 2014, Banff. **Anais**... Banff: 2014. Disponível em: <http://arxiv.org/abs/1312.4400>.

MA, Y.; CAO, Y.; VRUDHULA, S.; SEO, J. Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks. In: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17, **Anais**...2017.

MA, Y.; SUDA, N.; CAO, Y.; SEO, J. S.; VRUDHULA, S. Scalable and modularized RTL compilation of Convolutional Neural Networks onto FPGA. In: 26th International Conference on Field-Programmable Logic and Applications, FPL 2016, **Anais**...2016.

Post-training quantization. Disponível em: <https://www.tensorflow.org/performance/post_training_quantization>. Acesso em: 19 sep. 2018.

PUTNAM, A.; CAULFIELD, A. M.; CHUNG, E. S.; CHIOU, D.; CONSTANTINIDES, K.; DEMME, J.; ESMAEILZADEH, H.; FOWERS, J.; GOPAL, G. P.; GRAY, J.; OTHERS; PRASHANTH, G.; JAN, G.; MICHAEL, G.; SCOTT, H.; STEPHEN, H.; HORMATI, A.; SITARAM, J. K.; JAMES, L.; ERIC, L. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. **IEEE Micro**, v. 35, n. 3, p. 10–22, 2015. Disponível em: <http://ieeexplore.ieee.org/abstract/document/7106407/>. Acesso em: 25 sep. 2017.

RUSSAKOVSKY, O.; DENG, J.; SU, H.; KRAUSE, J.; SATHEESH, S.; MA, S.; HUANG, Z.; KARPATHY, A.; KHOSLA, A.; BERNSTEIN, M.; BERG, A. C.; FEI-FEI, L. ImageNet Large Scale Visual Recognition Challenge. **International Journal of Computer Vision (IJCV)**, v. 115, n. 3, p. 211–252, 11 Dec. 2015. Disponível em: <http://link.springer.com/10.1007/s11263-015-0816-y>. Acesso em: 30 may. 2018.

SANKARADAS, M.; JAKKULA, V.; CADAMBI, S.; CHAKRADHAR, S.; DURDANOVIC, I.; COSATTO, E.; GRAF, H. P. A Massively Parallel Coprocessor for Convolutional Neural Networks. In: 20th IEEE International Conference on Application-specific Systems, Architectures and Processors, **Anais**...IEEE, Jul. 2009. Disponível em: <http://ieeexplore.ieee.org/document/5200010/>. Acesso em: 22 apr. 2017.

SAVICH, A. W.; MOUSSA, M.; AREIBI, S. The Impact of Arithmetic Representation on Implementing MLP-BP on FPGAs: A Study. **IEEE Transactions on Neural Networks**, v. 18, n. 1, p. 240–252, 2007.

SCHMIDHUBER, J. Deep Learning in Neural Networks: An Overview. **Neural Networks**, v. 61, p. 85–117, 2015.

SHAFIEE, A.; NAG, A.; MURALIMANOHAR, N.; BALASUBRAMONIAN, R.; STRACHAN, J. P.; HU, M.; WILLIAMS, R. S.; SRIKUMAR, V. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In: Proceedings - 2016 43rd International Symposium on Computer Architecture, ISCA 2016, **Anais**...IEEE, Jun. 2016. Disponível em: <http://ieeexplore.ieee.org/document/7551379/>. Acesso em: 20 feb. 2017.

SHARMA, H.; PARK, J.; MAHAJAN, D.; AMARO, E.; KIM, J. K.; SHAO, C.; MISHRA, A.; ESMAEILZADEH, H.; THWAITES, B.; KOTHA, P.; GUPTA, A.; KIM, J. K.; MISHRA, A.; ESMAEILZADEH, H. From high-level deep neural models to FPGAs. In: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), **Anais**...IEEE, 2016. Disponível em: <http://ieeexplore.ieee.org/abstract/document/7783720/>. Acesso em: 23 feb. 2017.

SIMARD, P. Y.; STEINKRAUS, D.; PLATT, J. C. Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis. In: Seventh International Conference on Document Analysis and Recognition, ICDAR ’03, **Anais**...2003. Disponível em: <http://dblp.uni-trier.de/db/conf/icdar/icdar2003.html>.

SIMONYAN, K.; ZISSERMAN, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. In: International Conference on Learning Representations, ICLR 2015, **Anais**...2015. Disponível em: <http://arxiv.org/abs/1409.1556>.

SOUSA, M. C. F. de; MIGUEL ANGELO DE ABREU DE SOUSA; EMÍLIO DEL-MORAL-HERNANDEZ. Balancing Convolutional Neural Networks Pipeline in FPGAs. In: 27th International Conference on Artificial Neural Networks, **Anais**...2018.

SUDA, N.; CHANDRA, V.; DASIKA, G.; MOHANTY, A.; MA, Y.; VRUDHULA, S.; SEO, J.; CAO, Y. Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks. In: International Symposium on Field-Programmable

Gate Arrays - FPGA ’16, **Anais**...ACM Press, 2016. Disponível em: <http://dl.acm.org/citation.cfm?doid=2847263.2847276>. Acesso em: 22 feb. 2017.

SZE, V.; CHEN, Y. H.; YANG, T. J.; EMER, J. S.; MEMBER, S. S.; CHEN, Y. H.; MEMBER, S. S.; YANG, T. J. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. In: Proceedings of the IEEE, 12, **Anais**...2017.

SZE, V.; CHEN, Y.-H.; EMER, J.; SULEIMAN, A.; ZHANG, Z. Hardware for Machine Learning: Challenges and Opportunities. In: 2017 IEEE Custom Integrated Circuits Conference (CICC), **Anais**...2016. Disponível em: <http://arxiv.org/abs/1612.07625>.

SZEGEDY, C.; IOFFE, S.; VANHOUCKE, V.; ALEMI, A. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. In: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI-17), **Anais**...2016. Disponível em: <http://arxiv.org/abs/1602.07261>.

SZEGEDY, C.; LIU, W.; JIA, Y.; SERMANET, P.; REED, S.; ANGUELOV, D.; ERHAN, D.; VANHOUCKE, V.; RABINOVICH, A. Going deeper with convolutions. In: Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, **Anais**...2015.

TAPIADOR, R. Comprehensive Evaluation of OpenCL-Based Convolutional Neural Network Accelerators in Xilinx and Altera FPGAs. **preprint arXiv:1609.09296**, 2016. Disponível em: <https://arxiv.org/abs/1609.09296>. Acesso em: 20 feb. 2017.

VANHOUCKE, V.; SENIOR, A.; MAO, M. Improving the speed of neural networks on CPUs. In: Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011, **Anais**...2011. Disponível em: <http://research.google.com/pubs/archive/37631.pdf>.

WANG, C.; YU, Q.; GONG, L.; LI, X.; XIE, I. Y. DLAU: A Scalable Deep Learning Accelerator Unit on FPGA. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 3, Piscataway. **Anais**... Piscataway: IEEE, 2017. Disponível em: <http://ieeexplore.ieee.org/abstract/document/7505926/>. Acesso em: 20 feb. 2017.

WANG, Y.; XIA, L.; TANG, T.; LI, B.; YAO, S.; CHENG, M.; YANG, H. Low Power Convolutional Neural Networks on a Chip. In: IEEE International Symposium on

Computer Architecture, ISCAS 2016, 1, Montreal. **Anais**... Montreal: IEEE, 2016. Disponível em: <http://ieeexplore.ieee.org/document/7527187/?arnumber=7527187>.

WISSOLIK, B. M.; ZACHER, D.; TORZA, A.; DAY, B. **Virtex UltraScale+ HBM FPGA : A Revolutionary Increase in Memory Performance Industry Trends: Bandwidth and Power**. [s.l: s.n.]. Disponível em: <https://www.xilinx.com/support/documentation/white_papers/wp485-hbm.pdf>.

XILINX. **7 Series FPGAs Memory Resources**, 2017. . Disponível em: <https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf>.

XILINX. **UltraScale+™ Product Tables and Product Selection Guide**, 2018. . Disponível em: <https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf>.

ZEILER, M. D.; FERGUS, R. Visualizing and Understanding Convolutional Networks. In: Computer Vision–ECCV 2014, **Anais**...Springer, 2014. Disponível em: <http://link.springer.com/chapter/10.1007/978-3-319-10590-1_53>. Acesso em: 6 apr. 2017.

ZEILER, M. D.; TAYLOR, G. W.; FERGUS, R. Adaptive deconvolutional networks for mid and high level feature learning. In: Proceedings of the IEEE International Conference on Computer Vision, **Anais**...IEEE, 2011. Disponível em: <http://ieeexplore.ieee.org/abstract/document/6126474/>. Acesso em: 30 may. 2017.

ZHANG, C.; LI, P.; SUN, G.; GUAN, Y.; XIAO, B.; CONG, J. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In: Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA ’15, **Anais**...ACM, 2015. Disponível em: <http://dl.acm.org/citation.cfm?id=2689060>. Acesso em: 23 feb. 2017.

ZHANG, C.; WU, D.; SUN, J.; SUN, G.; LUO, G.; CONG, J. Energy-Efficient CNN Implementation on a Deeply Pipelined FPGA Cluster. In: Proceedings of the 2016 International Symposium on Low Power Electronics and Design, **Anais**...ACM Press, 2016. Disponível em: <http://dl.acm.org/citation.cfm?doid=2934583.2934644>. Acesso em: 23 feb. 2017.

ZHAO, W.; FU, H.; LUK, W.; YU, T.; WANG, S.; FENG, B.; MA, Y.; YANG, G.; WENLAI ZHAO; HAOHUAN FU; LUK, W.; TENG YU; SHAOJUN WANG; BO FENG; YUCHUN MA; GUANGWEN YANG; ZHAO, W.; FU, H.; LUK, W.; YU, T.; WANG, S.; FENG, B.; MA, Y.; YANG, G.; WENLAI ZHAO; HAOHUAN FU; LUK, W.; TENG YU; SHAOJUN WANG; BO FENG; YUCHUN MA; GUANGWEN YANG. F-CNN: An FPGA-based framework for training Convolutional Neural Networks. In: IEEE 27th International Conference on Application-specific Systems, Architectures and Processors, ASAP '16, **Anais**...IEEE, 2016. Disponível em: <http://ieeexplore.ieee.org/abstract/document/7760779/>. Acesso em: 22 apr. 2017.

APÊNDICE A – ENDEREÇAMENTO CONJUGADO DE BRAMS

Este apêndice apresenta como combinar endereçamentos para disponibilizar uma quantidade de endereços alocados em BRAMs diferente de potências de dois. Este apêndice estende os mecanismos descritos na subseção 3.4, onde os BRAMs foram utilizados para armazenar dados utilizando potências de dois como quantidade de endereços possíveis. Neste apêndice, o exemplo utilizado combina 1.024 com 512 endereços para possibilitar a formação de 1.536 endereços, útil para o armazenamento de *kernels* da terceira e quarta camadas convolucionais.

A adoção de 1 ciclo de *clock* para leitura de cada *kernel* foi descrita na seção 3.4 e utiliza 512 memórias individuais para armazenar os *kernels* da terceira camada convolucional e 384 memórias individuais para os *kernels* da quarta camada, com aproveitamento 75% dos endereços, pois utiliza 384 endereços dos 512 endereços disponíveis em cada memória individual. Porém, de acordo com o balanceamento do *pipeline*, descrito na seção 3.2, tanto a terceira quanto a quarta camada convolucional utilizam 4 ciclos de *clock* para execução de cada convolução, consequentemente, cada *kernel* destas camadas convolucionais pode ser lido da memória em até 4 ciclos de *clock*.

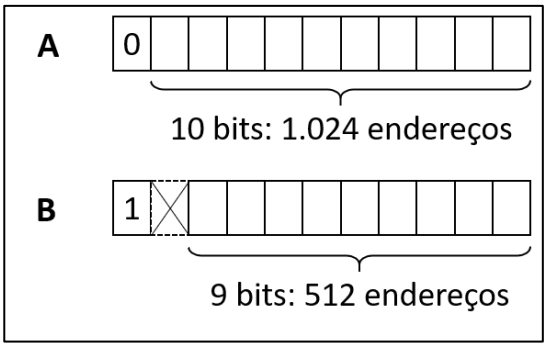
Ao utilizar 2 ciclos de *clock* para leitura de cada *kernel*, ao invés de 1 ciclo, a quantidade de memórias individuais lidas em cada ciclo de *clock* é reduzida pela metade, pois metade do *kernel* é lida em um ciclo de *clock* e a outra metade é lida de outro endereço em outro ciclo. Portanto, seriam necessários dois endereços de memória para cada um dos 384 *kernels,* ou seja, seriam necessários 768 endereços, ultrapassando os 512 endereços disponíveis nas memórias individuais. Para suportar esta quantidade de endereços seria necessário utilizar um segundo conjunto de memórias individuais, formando 1.024 endereços. Consequentemente, a quantidade de memórias individuais necessária se iguala ao caso em que cada *kernel* é lido em apenas um ciclo de *clock*. Algo semelhante acontece ao utilizar 3 ciclos de *clock* para leitura de cada *kernel*. A quantidade de memórias lidas em paralelo é reduzida para um terço, mas, como são utilizados 3 endereços por *kernel,* são necessários 1.152

endereços. Para esta quantidade de endereços são necessários 3 conjuntos de memórias, formando 1.536 endereços.

Ao utilizar 4 ciclos de *clock* para leitura de cada *kernel*, a quantidade de memórias lidas em paralelo é reduzida para um quarto, mas, neste caso, são necessários apenas 3 conjuntos de memórias, pois utilizando 4 endereços por *kernel* são necessários 1.536 endereços. Portanto, para este caso, a quantidade de memória necessária é 25% menor que a quantidade de memória necessária para leitura de 1 *kernel* a cada ciclo de *clock*. Porém, 1.536 não é uma potência de 2 e isso implica a subutilização de 512 endereços ao utilizar endereços de 11 bits. Para evitar esta subutilização, é necessário conjugar dois endereçamentos, um de 1.024 endereços e outro de 512, utilizando 10 e 9 bits respectivamente.

Ao conjugar dois endereçamentos, é necessário utilizar um bit adicional, portanto,

Figura 30 – Bits para endereçamento conjugado de 1.024 e 512 endereços.

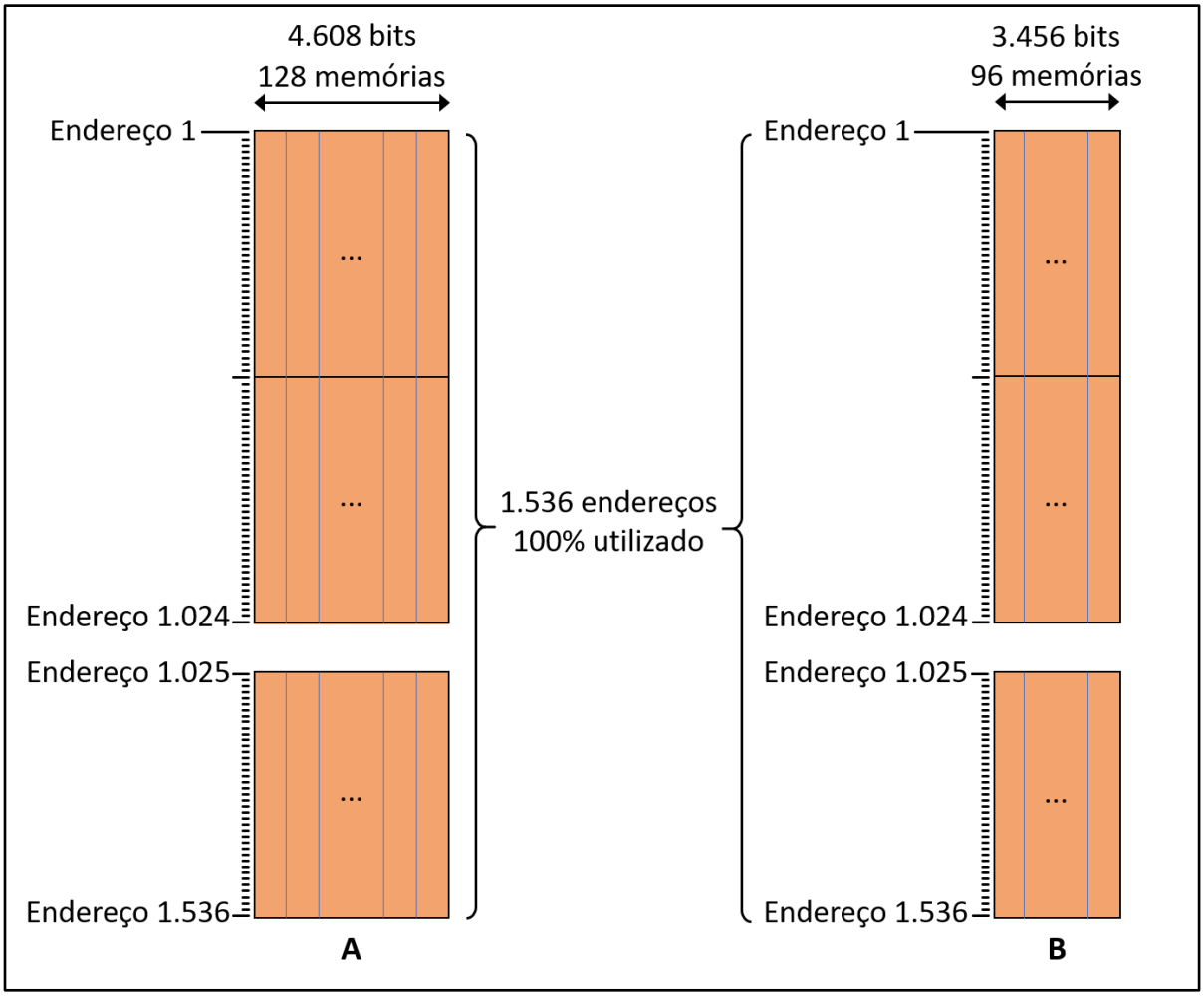


Fonte: Autoria própria (2019).

para conjugar um endereçamento de 10 bits com um de 9 bits, são necessários 11 bits. O bit mais significativo dos endereços é utilizado para discriminar qual dos endereçamentos é utilizado; quando este bit for 0, o endereçamento de 1.024 endereços é utilizado, quando este bit for 1, o endereçamento de 512 endereços é utilizado, como ilustrado na Figura 30. Quando o endereçamento de 1.024 endereços é utilizado, dos 11 bits de endereço, os 10 bits menos significativos são utilizados para identificar os 1.024 endereços possíveis, conforme ilustrado na Figura 30 **(A)**. Quando o endereçamento de 512 endereços é utilizado, o 2º bit mais significativo é desprezado, utilizando apenas os 10 bits menos significativos para identificar os 512 endereços possíveis, como ilustrado na Figura 30 **(B)**. Como resultado, são formados os 1.536 endereços, o suficiente para armazenar cada um dos 384 *kernels* da terceira ou quarta camada convolucional.

Utilizando o endereçamento conjugado, o armazenamento dos 18.432 bits de cada um dos 384 *kernels* da terceira camada convolucional utiliza 384 memórias individuais, como ilustrado na Figura 31 (**A)**. São utilizados 3 conjuntos de 128 memórias individuais, cada uma com profundidade 512 e largura de 36 bits, 2 conjuntos são utilizados para o endereçamento de 1.024, enquanto 1 conjunto é utilizado para o endereçamento de 512. Os bits de cada *kernel* da terceira camada são divididos em 4 pedaços de 4.608 bits que são armazenados em 4 endereços contíguos, que por sua vez, são divididos em 128 segmentos de 36 bits, armazenados nas memórias individuais.

Figura 31 – Endereçamento conjugado de 1.034 e 512 endereços.



Fonte: Autoria própria (2018).

Para armazenamento 13.824 bits de cada um dos 384 *kernels* da quarta camada convolucional, 288 memórias individuais são utilizadas, como ilustrado na Figura 31 (**B)**. São utilizados 3 conjuntos de 96 memórias individuais, cada uma com profundidade 512 e largura de 36 bits, 2 conjuntos são utilizados para o endereçamento de 1.024, enquanto 1 conjunto é utilizado para o endereçamento de 512. Os bits de cada *kernel* da terceira camada são divididos em 4 pedaços de 3.456 bits, armazenados em 4 endereços contíguos, que por sua vez, são divididos em 96 segmentos de 36 bits, armazenados nas memórias individuais.

Ao utilizar 4 ciclos de *clock* para leitura de cada *kernel* e o endereçamento conjugado, 128 memórias individuais são poupadas na terceira camada convolucional e 96 memórias são poupadas na quarta camada, reduzindo 224 memórias individuais no total. Dado que cada BRAM é formado por duas memórias individuais, são poupados 112 BRAMs em comparação com o armazenamento dos *kernels* destas camadas para leitura em 1 ciclo de *clock*.

APÊNDICE B – RESUMO DO ARTIGO PUBLICADO NO 27TH *INTERNATIONAL CONFERENCE ON ARTIFICIAL NEURAL NETWORKS* (ICANN)

Figura 32 – Resumo do artigo apresentado no congresso 27th *International Conference on Artificial Neural Networks* (ICANN).

Check for updates

Balancing Convolutional Neural Networks Pipeline in FPGAs

Mark Cappello Ferreira de Sousa[1(✉)],
Miguel Angelo de Abreu de Sousa[2],
and Emilio Del-Moral-Hernandez[1]

[1] Department of Electronic Systems Engineering, School of Engineering, University of São Paulo, São Paulo, Brazil
markfsousa@gmail.com, emilio@lsi.usp.br
[2] Electrical Department, Federal Institute of Education, Science and Technology – IFSP, São Paulo, Brazil
angelo@ifsp.edu.br

Abstract. Convolutional Neural Networks (CNNs) have achieved excellent performance in image classification, being successfully applied in a wide range of domains. However, their processing power demand offers a challenge to their implementation in embedded real-time applications. To tackle this problem, we focused in this work on the FPGA acceleration of the convolutional layers, since they account for about 90% of the overall computational load. We implemented buffers to reduce the storage of feature maps and consequently, facilitating the allocation of the whole kernel weights in Block-RAMs (BRAMs). Moreover, we used 8-bits kernel weights, rounded from an already trained CNN, to further reduce the need for memory, storing them in multiple BRAMs to aid kernel loading throughput. To balance the pipeline of convolutions through the convolutional layers we manipulated the amount of parallel computation in the convolutional step in each convolutional layer. We adopted the AlexNet CNN architecture to run our experiments and compare the results. We were able to run the inference of the convolutional layers in 3.9 ms with maximum operation frequency of 76.9 MHz.

Keywords: CNN · FPGA · Object recognition

1 Introduction

The adoption of embedded image recognition has grown in popularity with emerging applications such as autonomous car, unmanned aerial vehicle (UAV), smart glasses, Internet of Things and Smart Cities. One powerful tool for image recognition is Convolutional Neural Networks (CNNs), whose computational power demand poses a big challenge for embedded systems using real time image recognition. This scenario represents an opportunity for specialized hardware, such as FPGAs, to accelerate CNNs with low power consumption.

CNNs are composed of a sequence of convolutional (Conv) layers followed by a sequence of Fully Connected (FC) layers. The Conv layers work as feature extractors

V. Kůrková et al. (Eds.): ICANN 2018, LNCS 11139, pp. 166–175, 2018.
https://doi.org/10.1007/978-3-030-01418-6_17