COPPE
Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia
UFRJ

ARQUITETURAS DE HARDWARE PARA REDES NEURAIS CONVOLUCIONAIS VISANDO COMPROMISSO ENTRE CUSTO E DESEMPENHO

Alexandre Bazyl Zacarias de França

Tese de Doutorado apresentada ao Programa de Pós-graduação em Engenharia Elétrica, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Doutor em Engenharia Elétrica.

Orientadores: José Gabriel Rodríguez Carneiro Gomes
Fernanda Duarte Vilela Reis de Oliveira
Nadia Nedjah

Rio de Janeiro
Outubro de 2023

ARQUITETURAS DE HARDWARE PARA REDES NEURAIS CONVOLUCIONAIS VISANDO COMPROMISSO ENTRE CUSTO E DESEMPENHO

Alexandre Bazyl Zacarias de França

TESE SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA ELÉTRICA.

Orientadores: José Gabriel Rodríguez Carneiro Gomes
Fernanda Duarte Vilela Reis de Oliveira
Nadia Nedjah

Aprovada por: Prof. José Gabriel Rodríguez Carneiro Gomes
Prof. Fernanda Duarte Vilela Reis de Oliveira
Prof. Nadia Nedjah
Prof. Amit Bhaya
Prof. Luiza de Macedo Mourelle
Prof. Vanderlei Bonato

RIO DE JANEIRO, RJ – BRASIL
OUTUBRO DE 2023

França, Alexandre Bazyl Zacarias de

Arquiteturas de Hardware para Redes Neurais Convolucionais visando Compromisso entre Custo e Desempenho/Alexandre Bazyl Zacarias de França. – Rio de Janeiro: UFRJ/COPPE, 2023.

XIII, 87 p.: il.; 29, 7cm.

Orientadores: José Gabriel Rodríguez Carneiro Gomes
Fernanda Duarte Vilela Reis de Oliveira
Nadia Nedjah

Tese (doutorado) – UFRJ/COPPE/Programa de Engenharia Elétrica, 2023.

Referências Bibliográficas: p. 80 – 87.

1. Convolutional Neural Network (CNN). 2. Hardware Neural Network (HNN). 3. FPGA. 4. LeNet-5. 5. MNIST. 6. Hardware Architecture for Machine Learning. 7. Reconfigurable computing. 8. Parallelization. 9. Low-Power. I. Gomes, José Gabriel Rodríguez Carneiro *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia Elétrica. III. Título.

Agradecimentos

Aos meus pais Antonio e Garlete, (*in memoriam*), minha gratidão por tudo o que fizeram por mim ao longo de minha vida. Desejo ter sido merecedor do esforço dedicado por vocês em todos os aspectos, especialmente no que diz respeito à minha formação.

Gostaria de expressar minha sincera gratidão aos meus Orientadores pelo apoio inestimável ao longo do meu doutorado. Agradeço por dedicarem seu tempo e expertise para me orientar no desenvolvimento deste trabalho, na escrita de artigos científicos e da minha tese. As valiosas sugestões, críticas construtivas e orientações sábias foram fundamentais na minha formação. As inúmeras discussões, reuniões e debates enriqueceram meu conhecimento acadêmico e, sem dúvidas, também me ajudaram a crescer como indivíduo. Além disso, quero reconhecer a paciência e a disponibilidade que vocês demonstraram ao longo desta longa caminhada. Todo este empenho e compromisso são sinceramente apreciados.

Por fim, gostaria de agradecer aos amigos e colegas do Instituto de Pesquisas da Marinha, que contribuíram com informações valiosas e com os quais tive a oportunidade de aprender muito.

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

ARQUITETURAS DE HARDWARE PARA REDES NEURAIS CONVOLUCIONAIS VISANDO COMPROMISSO ENTRE CUSTO E DESEMPENHO

Alexandre Bazyl Zacarias de França

Outubro/2023

Orientadores: José Gabriel Rodríguez Carneiro Gomes
Fernanda Duarte Vilela Reis de Oliveira
Nadia Nedjah

Programa: Engenharia Elétrica

As redes neurais convolucionais, especialmente as embarcadas em dispositivos com requisitos de baixo consumo de energia, apresentam um desafio significativo: equilibrar a alta precisão esperada das redes neurais, o tempo de resposta e as demandas de recursos computacionais, lógicos e de memória. Este trabalho apresenta três arquiteturas de hardware para redes neurais convolucionais com alto grau de paralelismo e reutilização de componentes implementadas em um dispositivo programável. O primeiro projeto, denominado arquitetura com memória, utiliza a quantidade de memória necessária para armazenar os dados de entrada e os resultados intermediários. O segundo projeto, denominado arquitetura sem memória, define e explora um padrão específico de sequenciamento da entrada para evitar o uso de memória RAM. O terceiro projeto, denominado arquitetura com memória cache, é uma solução intermediária, onde a padronização da sequência de entrada também é explorada mas uma memória auxiliar é utilizada para armazenar alguns resultados intermediários e, consequentemente, melhorar o tempo de processamento. Comparamos as três arquiteturas em termos de potência, área e tempo de processamento. Permitir o uso de memória aumenta o custo geral de hardware, mas reduz o tempo de processamento. Na outra extremidade, dispensar completamente o uso de memória aumenta o nível de paralelismo mas compromete o tempo de processamento. O balanceamento entre uso de memória e desempenho é alcançado na arquitetura com memória cache que otimiza o tempo de processamento mas com custo em termos de recursos de hardware.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

CONVOLUTIONAL NEURAL NETWORK HARDWARE ARCHITECTURES EXPLORING THE TRADE-OFF BETWEEN COST AND PERFORMANCE

Alexandre Bazyl Zacarias de França

October/2023

Advisors: José Gabriel Rodríguez Carneiro Gomes
Fernanda Duarte Vilela Reis de Oliveira
Nadia Nedjah

Department: Electrical Engineering

Convolutional neural networks, especially when embedded in devices with low power consumption requirements, present a significant challenge: balancing the typically high precision expected of neural networks, response time, and the demands on computational, logical, and memory resources. This work presents three hardware architectures for convolutional neural networks with high degree of parallelism and component reuse implemented in a programmable device. The first design, which is termed memoryful architecture, uses as much memory as necessary to store the input data and intermediate results. The second design, which is termed memoryless architecture, defines and explores a specific input sequencing pattern to completely avoid the use of RAM. The third design, which is termed cache memory-based architecture, is an intermediate solution, where the standardization of the input sequence is also explored but an auxiliary memory is used to store some intermediate results and, consequently, improve processing time. We compare the three designs in terms of power, area and processing time. Allowing memory usage increases overall hardware cost but reduces processing time. At the other end, completely eliminating memory usage increases operation parallelism, but compromises processing time. A trade-off between memory usage and processing performance is achieved in the cache memory-based architecture that optimizes processing time but at a cost in terms of hardware resources.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

API	Application Programming Interface, p. 73
ASIC	Application-Specific Integrated Circuit, p. 6
BRAM	block-RAM, p. 43
CMA	Cache-memory Architecture, p. 59
CNN	Convolutional Neural Network, p. 1
CPU	Central Processing Unit, p. 5
DNN	Deep Neural Network, p. 28
DPU	Deep Learning Processing Unit, p. 8
DSP	Digital Signal Processor, p. 43
FIFO	First In, First Out, p. 13
FPGA	Field-Programmable Gate Array, p. 1
FPS	Frames per Second, p. 29
FPU	Floating Point Unit, p. 19
FSM	Finite-State Machine, p. 35
GPU	Graphics Processing Unit, p. 4
HDL	Hardware Description Language, p. 7
HLS	High-Level Synthesis, p. 7
HNN	Hardware Neural Network, p. 1
IP	Intellectual Property, p. 8
IoT	Internet of Things, p. 1

LPFP	Low-Precision Floating Point, p. 29
LUT	Lookup Table, p. 15
MAC	Multiply-Accumulate, p. 11
MFA	Memoryful Architecture, p. 33
MLA	Memoryless Architecture, p. 46
MMA	Multiple MACs per Layer Architecture, p. 55
RAM	Random Access Memory, p. 17
ROM	Read-Only Memory, p. 33
RTL	Register-Transfer Level, p. 7
ReLU	Rectified Linear Unit, p. 15
SIMD	Single instruction, Multiple Data, p. 27
SoC	System-On-Chip, p. 73
TPU	Tensor Processing Units, p. 6
VHDL	VHSIC Hardware Description Language, p. 7

Capítulo 1

Introdução

Nos últimos anos, as redes neurais convolucionais (*convolutional neural network*, CNNs) se estabeleceram como uma ferramenta essencial em uma variedade de aplicações, graças à sua notável precisão na classificação de dados, especialmente em tarefas de processamento de imagens. No entanto, essa precisão muitas vezes vem com um custo significativo em termos de recursos computacionais e memória. À medida que a demanda por sistemas embarcados capazes de realizar tarefas de alto desempenho e baixo consumo de energia continua a crescer, as arquiteturas de hardware que exploram o potencial das redes neurais têm ganhado destaque.

A implementação de redes neurais em hardware (*hardware neural networks*, HNNs) em dispositivos programáveis, como *Field-Programmable Gate Arrays* (FPGAs), representa uma abordagem empolgante para a aceleração de tarefas de aprendizado de máquina. Os dispositivos FPGAs oferecem a vantagem de serem altamente configuráveis, permitindo que a arquitetura de hardware seja otimizada de acordo com as necessidades específicas do projeto. Isso significa que é possível projetar HNNs altamente especializadas, ajustadas para realizar tarefas específicas com eficiência. Além disso, os FPGAs aproveitam o paralelismo inerente das operações de rede neural, permitindo um desempenho notável em tarefas de processamento de dados em tempo real.

No entanto, o projeto de HNNs em FPGA não está isento de desafios. A escolha entre abordagens de projeto de síntese de alto nível e projeto usando descrição em nível de sinais é apenas um dos dilemas enfrentados pelos engenheiros. Compreender as nuances dessas abordagens é essencial para o sucesso da implementação de HNNs em FPGA. Além disso, a eficiência de recursos e o consumo de energia devem ser cuidadosamente equilibrados, uma vez que a alocação de memória e os requisitos de hardware podem afetar significativamente o desempenho e a eficiência de uma HNN em FPGA.

Além disso, à medida que a Internet das Coisas (*internet of things*, IoT) continua a se expandir, há uma crescente necessidade de implementar técnicas de aprendizado

de máquina em dispositivos de baixa complexidade. Desta forma, este trabalho se propõe a apresentar arquiteturas de hardware para CNN, analisar e comparar diferentes abordagens de implementação de redes neurais em hardware, destacando suas aplicações e compromissos em termos de precisão, desempenho e eficiência energética. Além disso, exploramos como essas implementações podem ser adaptadas para atender às demandas crescentes, oferecendo soluções práticas para sistemas embarcados com recursos limitados.

1.1 Motivação

O uso de CNNs permite alcançar resultados muito precisos em diversas áreas de aplicação e melhorias neste tipo de redes neurais estão surgindo em ritmo acelerado. Hardware mais poderoso, modelos e conjuntos de dados maiores, novos algoritmos e arquiteturas de redes neurais aprimoradas desafiam constantemente o estado da arte das CNNs.

A implementação de uma CNN em um FPGA pode apresentar desafios significativos em termos de uso de memória e recursos lógicos. Primeiramente, as CNNs são conhecidas por exigir uma quantidade substancial de memória devido aos seus pesos e ativações, o que pode sobrecarregar a capacidade disponível em um FPGA. Além disso, as operações de convolução e pooling consomem recursos lógicos consideráveis, tornando essencial a otimização cuidadosa do projeto para acomodar essas operações em um FPGA com recursos limitados. Estratégias como redução de precisão, compartilhamento de recursos e uso eficiente de memória cache são frequentemente necessárias para superar esses desafios e alcançar uma implementação eficaz de uma CNN em um FPGA. Portanto, a otimização de recursos de hardware, tanto em termos de memória quanto em recursos lógicos, desempenha um papel fundamental na implementação bem-sucedida de CNNs em FPGAs.

Uma alocação adequada de memória pode facilitar o armazenamento de pesos e resultados intermediários, permitindo, inclusive, que mais operações sejam realizadas simultaneamente. Vencer o desafio de otimizar a memória e o paralelismo é crucial para alcançar uma implementação eficiente em FPGA, onde a escolha cuidadosa dos tamanhos de filtros, o compartilhamento de recursos e a estratégia de alocação de memória desempenham um papel fundamental na busca pelo equilíbrio ideal entre desempenho e custo de hardware.

Neste contexto, propusemos arquiteturas de HNN distintas que se concentram no armazenamento de mapas de características, explorando o desempenho, uso de recursos lógicos e de memória. Cada uma das arquiteturas aborda de forma única a questão do armazenamento de dados, indo desde o armazenamento completo até solução sem o emprego de memória, e até mesmo níveis de armazenamento per-

sonalizado. Este trabalho busca oferecer uma compreensão mais profunda dessas abordagens e o compromisso entre custo e desempenho de uma CNN em hardware, com a implementação das arquiteturas propostas em dispositivos FPGA.

1.2 Principais Contribuições

A proposta de novas arquiteturas em hardware para CNN que podem ser implementadas em dispositivo programável de baixo custo e que não dispõe de recursos adicionais de processamento ou aritméticos. Arquiteturas que independam do fabricante do dispositivo programável e que podem ser escaladas constituem contribuições para a discussão de HNN.

A avaliação do uso de memória e o seu impacto na necessidade de outros recursos e no tempo de processamento poderá revelar-se muito útil para aplicações com restrição de consumo de energia. É relevante ressaltar que a apresentação dos resultados de acurácia, utilizando representações numéricas mais compactas e incorporando largura de bits variável em componentes de processamento específicos, contribui para as discussões relacionadas ao aumento da precisão das redes neurais. Essa abordagem também influencia de maneira direta as taxas de transferência e, novamente, tem implicações substanciais na eficaz administração dos recursos de memória.

As contribuições desta tese consistem em adicionar ao rol de trabalhos afetos ao projeto e à implementação de HNNs, o resultado da acurácia alcançada com representação numérica em ponto fixo de largura variável; projeto de arquiteturas que podem ser implementadas em RTL e de forma parametrizável, facilitando a escalabilidade; e, destacamos, como contribuição principal na discussão de HNN em FPGA, aplicável em arquiteturas de CNN no estado-da-arte, o compromisso entre o uso de memória e desempenho.

A definição de um padrão customizado para leitura dos dados de entrada e sequenciamento das operações compreende uma técnica que não é comumente abordada na literatura e resultou em um artigo de conferência. Neste artigo, foram apresentadas duas propostas distintas: uma arquitetura utilizando memória para o armazenamento dos mapas de características, enquanto a segunda arquitetura elimina essa necessidade ao explorar um padrão diferenciado para o sequenciamento de dados no processamento. Nesta segunda abordagem, é possível armazenar apenas um resultado intermediário por canal da rede neural, dispensando completamente o uso de memória RAM, além de realizar o processamento das camadas em pipeline. O artigo é intitulado "*Non-memoryless vs. Memoryless Hardware Architectures for Convolutional Neural Networks*" [2] publicado no *IEEE Latin America Symposium on Circuits and System*. Esta tese resultou em um artigo de periódico intitulado "*Hardware designs for convolutional neural networks: Memoryful, memoryless and*

cached" [3] publicado em *Integration, the VLSI Journal*. Neste artigo foi apresentada uma terceira arquitetura, que ao fazer uso de alguns elementos de memória cache, consegue melhorar significativamente o desempenho da arquitetura *memoryless*.

1.3 Organização da Tese

O presente trabalho está estruturado de forma a dar ao leitor uma visão geral de arquiteturas de CNN projetadas e otimizadas para implementação em hardware. Inicialmente, são apresentadas algumas características de dispositivos dedicados de forma a indicar a escolha pela implementação em dispositivos FPGA, seguida de tópicos relacionados a aceleração de rede neural convolucional.

A organização do restante do texto é descrita a seguir.

No Capítulo 2, são apresentadas considerações sobre as camadas e componentes que compõem uma CNN para serem implementados em dispositivos programáveis. Inicialmente são apresentadas as vantagens de utilizar tais dispositivos e, ao longo do capítulo, são discutidos alguns desafios e restrições com algumas linhas de ação para serem consideradas no projeto de uma HNN. Nesse capítulo também são apresentadas as estruturas comuns utilizadas para a implementação e teste das arquiteturas propostas.

O Capítulo 3 contém trabalhos relacionados aos desafios elencados no capítulo anterior. São indicados alguns estudos, técnicas e soluções para restrições normalmente impostas ao otimizar o uso de memória, aumentar a precisão da rede neural, usar paralelismo e pipeline.

Nos Capítulos 4, 5 e 6 são apresentadas as arquiteturas propostas neste trabalho. Cada arquitetura é descrita e detalhada com seus componentes básicos, algoritmos, características, otimizações e resultados alcançados.

O Capítulo 7 compara os resultados obtidos em cada uma das arquiteturas. O comparativo inclui outras arquiteturas e soluções apresentadas em trabalhos acadêmicos, abrangendo também dispositivos que empregam GPU (*graphics processing unit*) embarcada.

No Capítulo 8 são apresentadas as conclusões da pesquisa e sugestões de trabalhos futuros.

Capítulo 2

Redes Neurais Convolucionais em Hardware

As CNNs tornaram-se populares, em parte, por alcançarem excelentes resultados em aplicações especializadas como processamento de imagens [4–8], de vídeo [9], reconhecimento da fala [10–12], entre outras. À medida que as aplicações de classificação e reconhecimento de padrões se tornam complexas, as redes neurais convolucionais se tornam mais densas e, consequentemente, passam a exigir mais capacidade computacional para atingir melhores resultados em termos de precisão ou de velocidade de processamento. Dentro desta busca por melhorias, temos as CNN-3D, que representam variações das arquiteturas de CNNs e são adaptadas para processar dados volumétricos, como imagens de tomografia computadorizada ou ressonância magnética, e também para considerar a temporalidade no processamento de vídeos.

A implementação de CNNs em dispositivos dedicados apresenta um cenário de desafios diferente do encontrado quando estas redes neurais são desenvolvidas essencialmente em software. Algumas arquiteturas de hardware que exploram o paralelismo inerente das redes neurais foram propostas [13] e, apesar do crescimento do poder de processamento de CPUs (*central processing unit*) e GPUs, as arquiteturas implementadas em dispositivos programáveis como FPGA ganharam espaço e relevância [14]. Este tipo de dispositivo encontra aplicações em sistemas embarcados que, adicionalmente ao baixo consumo, costumam demandar resultados precisos e baixo tempo de processamento. Face a estas demandas, os FPGAs tornam-se dispositivos interessantes para a implementação de redes neurais convolucionais, atuando como aceleradores em hardware que apresentam sua capacidade de paralelismo como grande atrativo. O hardware sintetizado é composto por diversos circuitos combinacionais e sequenciais independentes que são capazes de processar dados simultaneamente, resultando em uma poderosa capacidade de processamento e de alcançar elevadas taxas de transferências.

Um acelerador é um conjunto de hardware especializado que executa tarefas com

alto desempenho e eficiência quando comparado com plataformas genéricas, como as CPUs. Alguns exemplos de dispositivos utilizados como aceleradores são GPUs, FPGAs, TPUs (*tensor processing units*) e ASICs (*application-specific integrated circuit*). Conforme apresentado na Figura 2.1, as CPUs são as mais flexíveis e os ASICs são os mais eficientes em latência e consumo de energia [15]. Os TPUs são circuitos integrados projetados especificamente para acelerar a grande carga de operações aritméticas envolvidas em aplicações de aprendizado de máquina [16]. Estes dispositivos podem acelerar multiplicações de matrizes e, portanto, podem ser aplicados em outras tarefas que requeiram computações intensivas [17]. Os dispositivos ASICs embora conhecidos pela eficiência e baixo custo de produção quando em fabricação seriada, possuem ciclo de desenvolvimento longo, extensa fase de verificação e, consequentemente, alto custo de projeto e sem margem para erros durante o processo de desenvolvimento. Os FPGAs se enquadram em uma categoria que apresenta um bom equilíbrio entre eficiência e flexibilidade, principalmente pelo paralelismo e reconfigurabilidade destes dispositivos.

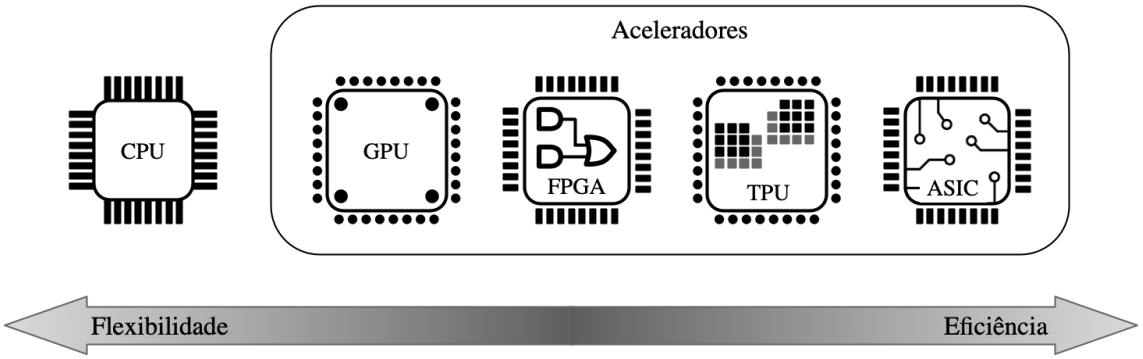


Figura 2.1: Flexibilidade × Eficiência de aceleradores de hardware.

Uma grande vantagem de sistemas baseados em FPGA sobre os sistemas que empregam CPU ou GPU é a disponibilidade de blocos lógicos programáveis que podem ser organizados em aceleradores altamente especializados para a realização de tarefas específicas. Desta forma, os FPGAs podem conseguir melhores taxas de processamento, transferência e de consumo de energia. Estas vantagens, em contrapartida, vêm com o custo do aumento da complexidade durante o desenvolvimento onde é necessário considerar cuidadosamente os recursos de hardware disponíveis e realizar o mapeamento eficiente do algoritmo para o dispositivo FPGA empregado [18].

Devido a sua característica de serem programáveis e flexíveis na organização dos blocos lógicos, os dispositivos FPGA acabam apresentando, como desvantagem, a necessidade de embarcar interconexões genéricas entre os blocos, aumentando assim a área ocupada e a taxa de latência devido à lógica de configuração. Contrapondo estas desvantagens, se sobressaem os ASICs. Por serem dispositivos ainda mais especializados, normalmente, resultam em sistemas menores, mais rápidos e com maior

eficiência energética. Entretanto, os ASICs são mais adequados para aplicações que possuem alto volume de produção, onde os custos de engenharia e fabricação podem ser compartilhados entre um grande número de dispositivos [19]. FPGAs, com sua capacidade de reprogramação e com etapas de implementação mais simples, são mais adequados para prototipagem e ciclos de desenvolvimento curtos, possibilitando que a arquitetura seja modificada e otimizada iterativamente.

Tradicionalmente, os FPGAs são programados usando uma linguagem de descrição de hardware (*hardware description language*, HDL), como VHDL ou Verilog. A maioria dos projetos são descritos em *register-transfer level* (RTL), onde o desenvolvedor realiza seu projeto utilizando processos paralelos que operam com vetores de sinais binários e representações numéricas resultantes destes sinais [19]. Estes processos descrevem lógica combinacional, operações aritméticas, registradores, entrada e saída de sinais, por exemplo. Contudo, o trabalho de mapear um algoritmo utilizando blocos lógicos, processos e máquinas de estados, conforme mencionado anteriormente, demanda maior conhecimento do dispositivo e decisões de design devem ser tomadas antes de qualquer codificação pois, dependendo da complexidade do projeto, as modificações e otimizações iterativas podem ser difíceis e custosas.

Para aumentar o nível de abstração, reduzir o ciclo de desenvolvimento e a curva de aprendizado, existem ferramentas que auxiliam na implementação de projetos em FPGA por meio de programação em alto nível. Conhecidas como HLS (*high-level synthesis*), estas ferramentas permitem algumas abstrações sobre o hardware e detalhes de implementação contribuindo para um tempo de desenvolvimento menor [20]. A síntese de alto nível permite a conversão de uma estrutura lógica descrita em uma linguagem como C ou C++ para um circuito descrito em uma linguagem de baixo nível de abstração. Ou seja, o compilador HLS é o responsável por converter a descrição sequencial utilizada no software em uma descrição concorrente do hardware, geralmente no nível RTL. Pode-se dizer que estas ferramentas constituem uma ponte que conecta um projeto de software para o domínio de hardware [21].

O aumento do nível de abstração ao utilizar síntese de alto nível promete ciclos de desenvolvimento mais rápidos, facilidade de otimização e maior produtividade ao custo de menos controle sobre o resultado final [19]. Porém, a tarefa de converter descrições de software sequenciais de alto nível em arquiteturas de hardware paralelas e otimizadas nem sempre é uma tarefa simples. Os resultados alcançados ainda são altamente dependentes do estilo de codificação e detalhes do projeto [22], além da dependência do fabricante. Ou seja, cada fabricante de FPGA apresenta um conjunto de ferramentas de HLS onde as diretivas e restrições utilizadas para gerar as funcionalidades para o hardware podem variar. Entre as ferramentas populares de HLS estão o Vivado HLS [23] da Xilinx e Intel High Level Synthesis Compiler [24]. Com o objetivo de oferecer uma abordagem mais flexível e eficiente,

a Xilinx desenvolveu o *Deep Learning Processing Unit* (DPU) que corresponde a uma série de IPs (*Intellectual Property*) dedicada a aceleração de redes neurais convolucionais. O DPU pode ser integrado, como um bloco, em dispositivos FPGA das séries Zynq 7000 e UltraScale+, e programados em HLS utilizando um conjunto especializado de instruções que permite implementar alguns tipos de redes neurais convolucionais.

Utilizar linguagens descritivas de hardware, a nível RTL, como o VHDL, facilita a otimização de recursos de hardware com ganhos consideráveis em termos de utilização de blocos lógicos e de memória e, com a grande vantagem de ser independente do fabricante do dispositivo. As otimizações do hardware sintetizado são importantes para possibilitar que redes neurais complexas sejam implementadas em FPGAs mais simples e de baixo custo, ainda que o avanço da tecnologia destes dispositivos reflita em componentes com cada vez mais recursos disponíveis e, naturalmente, menos limitados.

2.1 Aceleração de Rede Neural Convolucional

Um dos trabalhos pioneiros das CNNs foi apresentado em [25] para reconhecimento de caracteres manuscritos. Este trabalho contribuiu significativamente para a definição das arquiteturas de CNN utilizadas hoje em dia, com camadas convolucionais para extrair características, intercaladas com camadas de *pooling*, utilizadas para reduzir a dimensionalidade destas características, e camadas completamente conectadas realizando a classificação.

Os mapas de características são resultados das convoluções e de uma função de ativação utilizada para quantificar a presença de determinadas características no campo receptivo. Portanto, pode-se dizer que um mapa de características é formado pelas ativações das convoluções realizadas. É importante destacar que a equivalência espacial é mantida entre estas ativações e os campos receptivos utilizados na convolução, permitindo, assim, o mapeamento das características extraídas em cada camada convolucional.

Uma arquitetura CNN típica para classificação geralmente compreende camadas alternadas de convolução e sub-amostragem espacial (*pooling*) seguidas por uma ou mais camadas *fully-connected*. Ao final das camadas convolucionais ou de *pooling* também é inserida uma função de ativação. Para melhor compreender a função de cada uma dessas camadas e os desafios envolvidos nas arquiteturas de hardware para CNN, são apresentados os principais elementos e componentes a serem implementados.

A Figura 2.2 exemplifica uma arquitetura em que o sistema de processamento consiste principalmente de uma CPU e memória externa, conectados entre si e tam-

bém a um dispositivo FPGA. Devido à quantidade de dados de entrada e pesos, normalmente não é possível armazenar todos estes dados na memória interna do chip. A CPU pode configurar o módulo de controle implementado em lógica programável, de modo que ajuste o acelerador para acomodar diferentes números de camadas convolucionais. Adicionalmente, a CPU pode ser utilizada para realizar algumas operações simples, como a função de ativação. Como as camadas convolucionais costumam incluir a maior parte das operações aritméticas da CNN, acelerar as demais operações pode apresentar pequena melhoria no desempenho e, desta forma, estas demais operações podem ser atribuídas à CPU de forma a aproveitar sua eventual ociosidade e também reduzir o uso de área do FPGA.

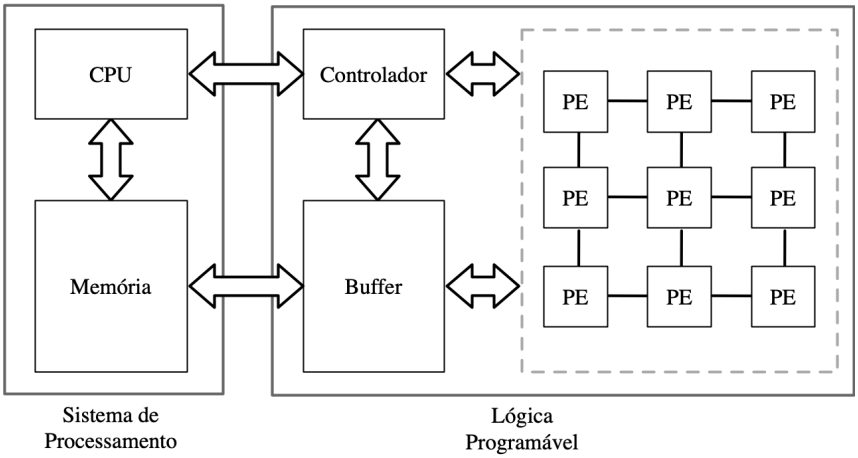


Figura 2.2: Acelerador para CNN baseado em uma plataforma com CPU e FPGA embarcados.

O bloco de lógica programável na Figura 2.2 pode compreender um FPGA que implementa a lógica das camadas que compõem a rede neural e a transferência de dados entre elas. O bloco de lógica programável é composto por alguns componentes, incluindo os elementos de processamento (*processing element*, PE), módulo de controle e buffer. Os PEs são unidades de computação e o número de PEs, principalmente quando operam em paralelo, afeta diretamente o desempenho computacional do acelerador para CNN. Os dados podem ser trocados entre PEs para serem reutilizados e dispensar a necessidade de acessar o buffer. O buffer é usado para armazenar dados, pesos e resultados localmente e de forma temporária no FPGA. Como os dados e os pesos das camadas convolucionais são reutilizados repetidamente, o armazenamento em buffer interno pode reduzir o acesso à memória externa. O módulo de controle recebe informações de configuração do sistema de processamento, controla o processo computacional e a transferência de dados nos elementos de processamento.

A Figura 2.3 apresenta uma arquitetura típica utilizada nas implementações em FPGA [26]. As informações primárias, como imagem de entrada e os pesos, são ar-

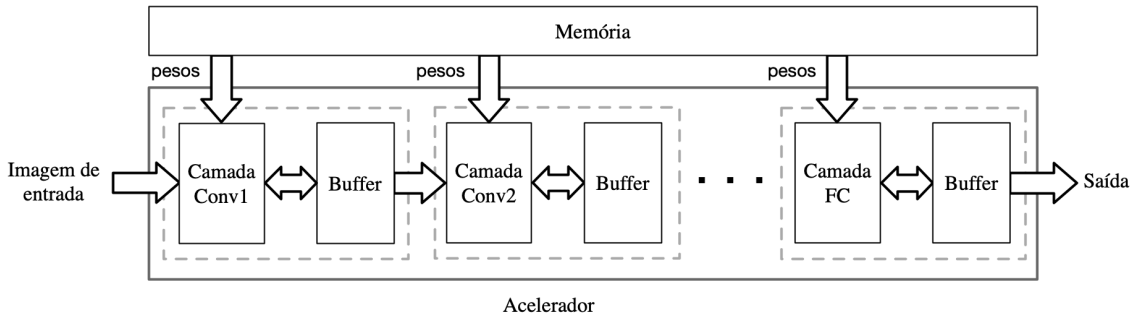


Figura 2.3: Acelerador para CNN para uma plataforma com FPGA e memória externa.

mazenadas na memória externa e devem ser transferidas inteiramente para o FPGA antes do processamento de cada camada da rede neural. Para minimizar este problema, e reduzir o acesso de dados na memória externa, os resultados intermediários computados entre as camadas são armazenados em buffers do próprio dispositivo. Esta redução costuma ser significativa pois estes dados intermediários são usados como entrada para as camadas seguintes.

2.1.1 Camadas Convolucionais

Nas camadas convolucionais está localizada a maior parte da computação da CNN. Uma camada convolucional é composta por um conjunto de pesos, que são definidos durante o treinamento da rede neural. A quantidade de pesos varia de camada para camada, e cada um deles é aplicado aos dados provenientes da camada anterior, desempenhando um papel fundamental na extração de características e informações durante a computação da rede neural. Os pesos em uma camada convolucional operam sobre os dados por meio de operações de convolução. Essas operações consistem em uma multiplicação dos pesos com os valores dos dados da camada anterior, seguida pela soma dos resultados. Isso é feito de forma a abranger diferentes partes dos dados de entrada, permitindo que a rede neural extraia características relevantes e seja capaz de reconhecer padrões específicos, tornando-se mais eficaz em tarefas como classificação de imagens, detecção de objetos e processamento de informações.

Uma camada convolucional pode ser composta por vários kernels (cada kernel é uma matriz, ou tensor, de valores), dependendo do número de canais de entrada d_e e saída d_s existentes. O total de kernels é dado por d_s. Os kernels são filtros de tamanho $k \times k \times d$, onde k é a largura do kernel e d é a profundidade do tensor e, de forma geral, corresponde ao número de canais de saída da camada anterior. O número de canais de uma camada está diretamente relacionado à profundidade do tensor usado. Esses canais também são chamados de mapas de características ou filtros, e desempenham um papel importante na extração de características e

no aprendizado das CNNs. Cada canal representa um conjunto de características específicas detectadas na imagem de entrada. Para a realização da convolução, a imagem é subdividida em segmentos conhecidos como campos receptivos. O kernel, usando um conjunto específico de pesos, é aplicado na imagem onde operações aritméticas são realizadas entre estes pesos e os elementos correspondentes do campo receptivo. A operação de convolução de duas dimensões para a primeira camada da rede neural $\mathbb{C}_1$ pode ser expressa da seguinte forma (considerando $d = 1$, para simplificar a notação no caso de uma imagem de entrada com somente um canal, ou seja, uma imagem de entrada representada em tons de cinza):

$$\mathbb{C}_1(x_1, y_1, d) = \sum_{j=y_1}^{y_1+\alpha} \sum_{i=x_1}^{x_1+\alpha} a(i,j) \cdot f(i - x_1, j - y_1, d) + bias(d), \tag{2.1}$$

onde $a(i, j)$ representa um pixel da imagem de entrada, $\alpha = k - 1$, f é o kernel, d é o número do canal, e x_1 e y_1 indicam a coordenada da imagem de entrada utilizada como referência para o posicionamento do kernel (canto superior esquerdo). Cada canal é também chamado de mapa de características. Conforme proposto em [25], a camada convolucional C_1 possui 6 mapas de características, sendo que cada um deles é calculado a partir da imagem de entrada, que só tem um canal. A camada convolucional C_3 possui 16 mapas de características, sendo que cada um deles é calculado a partir da saída da camada C_1, que tem 6 canais. E a camada convolucional C_5 possui 120 saídas, sendo que cada uma delas é calculada a partir (de um subconjunto) dos 16 canais de saída da camada C_3. As camadas 2 e 4 serão chamadas de camadas de *pooling*, conforme é descrito na Seção 2.1.2.

Os elementos de processamento da arquitetura são responsáveis por aplicar filtros (kernels) sobre os dados de entrada, à medida que são lidos da memória. O processo de convolução é constituído por multiplicação seguida de uma adição. Esse mecanismo também é conhecido pela operação *Multiply-ACcumulate* (MAC), que consiste no produto entre dois números e adição a um valor acumulado. O número de operações por convolução varia de acordo com a entrada e o tamanho do filtro. Considerando o número de kernels contabilizado anteriormente, o total de *bias* (em número equivalente ao total de canais de uma camada) e o tamanho k dos filtros, é possível dimensionar a quantidade de dados que devem ser armazenados e computados.

A implementação em hardware de uma camada convolucional apresenta considerações que devem ser estudadas para garantir que o dispositivo possa fornecer os recursos necessários. Assim, é importante analisar uma arquitetura com um mecanismo eficiente tanto no acesso quanto no processamento de dados. De forma mais específica, a implementação de uma camada convolucional envolve processos de leitura e escrita de memória, armazenamento temporário de dados e operações aritméticas como multiplicação e adição.

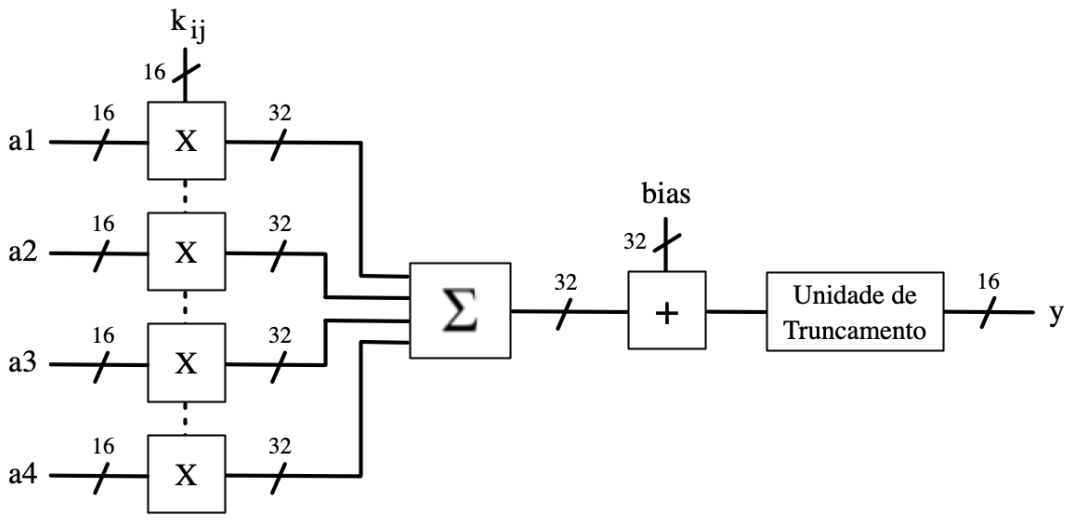


Figura 2.4: Aritmética nas camadas convolucionais com entrada e saída em ponto fixo de 16 bits.

A computação utilizando ponto fixo, mesmo variando a escala de precisão, permite ganhos no consumo de energia, latência e largura de banda em comparação com implementações de ponto flutuante de 32 bits (FP32) [27], por exemplo. Normalmente estas implementações de hardware apresentam transferências de dados com largura de bits reduzida e somente nas unidades aritméticas são utilizadas larguras maiores para reduzir a perda de precisão. Um exemplo desta combinação de diferentes representações numéricas é apresentado na Figura 2.4. Em contrapartida, a multiplicação de números binários de ponto flutuante de 32 bits no formato IEEE 754 é realizada em três etapas: multiplicação das mantissas de 24 bits, adição dos expoentes de 8 bits e a definição do bit de sinal, resultando em um processo aritmético mais complexo e custoso computacionalmente. O trabalho [28] apresenta uma proposta para a multiplicação de números em ponto flutuante de 32 bits de precisão simples no padrão IEEE 754.

2.1.2 Camadas de *Pooling*

A camada de *pooling* é a camada responsável pela operação de subamostragem ao longo das dimensões espaciais [29]. A presença desta camada ao longo de uma CNN permite reduzir progressivamente o tamanho da representação dos dados, diminuindo o número de parâmetros e consequentemente simplificando a computação. A Figura 2.5 apresenta um exemplo onde 16 mapas de características de tamanho 28×28 são reduzidos para o tamanho 14×14. Esta redução espacial não abrange a dimensão de profundidade [29] que representa os canais, ou seja, $d_e = d_s$ em uma camada de *pooling*.

Esta camada é implementada em pequenas janelas (geralmente 2×2 ou 4×4)

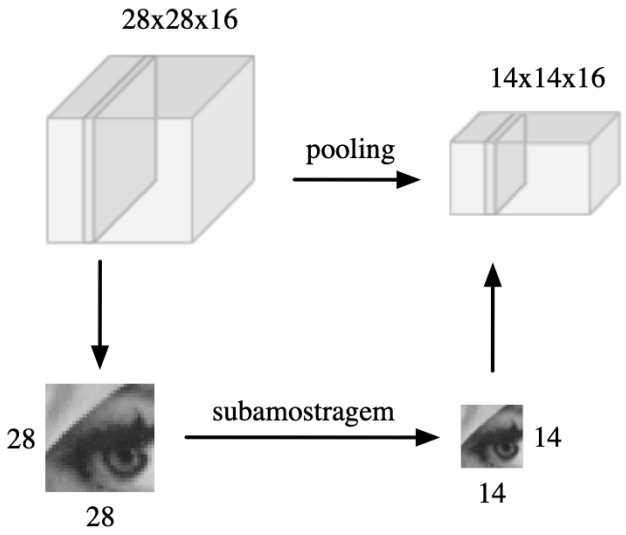


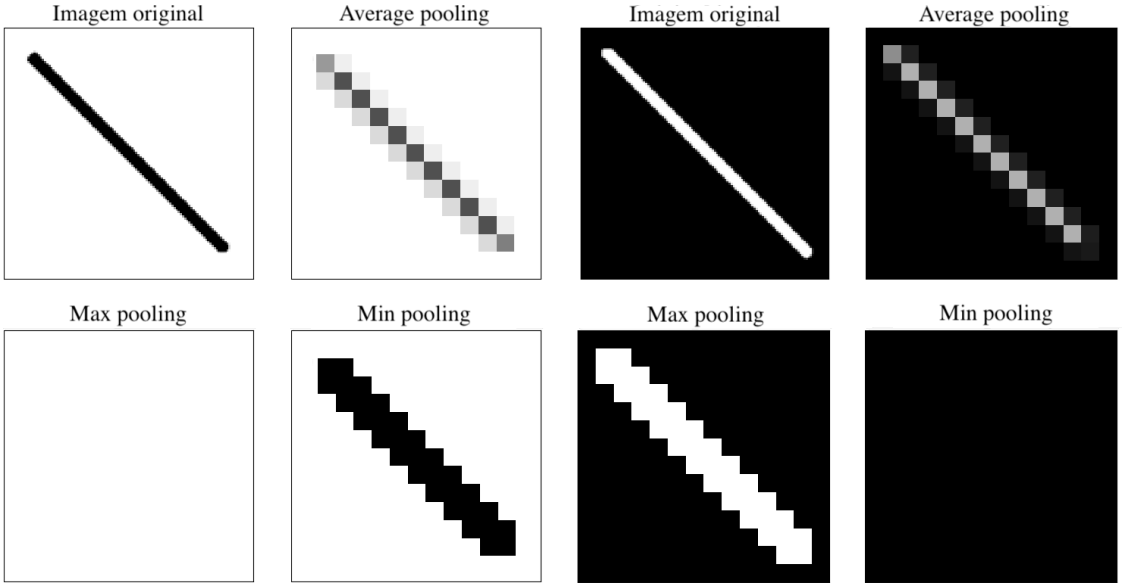
Figura 2.5: Operação de *pooling* em uma camada com 16 canais.

das matrizes de características resultantes das camadas de convolução. Dentro desta janela, a operação de *pooling* seleciona ou calcula um valor de saída de acordo com um determinado operador. Os operadores mais comumente utilizados são o *max pooling* que seleciona o pixel de maior valor, *min pooling* que seleciona o pixel de menor valor e *mean* ou *average pooling* onde a média entre os valores é calculada.

Considerando que as operações de *pooling* atuam no domínio das features (características), ao utilizar *min pooling*, o resultado é a obtenção de características com valores mais baixos. Por outro lado, o *max pooling* seleciona características com valores mais altos. A escolha da operação de *pooling* deve ser feita de acordo com os dados disponíveis. Na implementação da CNN LeNet-5 [25] para processar o conjunto de dados MNIST [30], o *max pooling* é o operador empregado. A Figura 2.6 ilustra os efeitos dos operadores de *pooling* em duas matrizes de características representadas como imagens para permitir a visualização.

Com relação ao custo computacional para a implementação dos operadores em hardware é seguro afirmar que o *average pooling* é mais custoso, por requerer as operações aritméticas necessárias para o cálculo da média entre os pixels. Para os demais operadores de *pooling*, apenas operações de comparação são necessárias. Assim como na camada convolucional, para implementação desta camada em hardware, também devem ser avaliados os processos de leitura e escrita de memória.

A Figura 2.7 apresenta uma proposta de arquitetura de hardware para realizar operações de *min pooling*. Neste exemplo, é utilizado um filtro de tamanho 2×2 indicado pela presença de quatro registradores após a fila FIFO (*first in, first out*). Cada par de valores na janela deslizante é comparado para determinar qual é o menor. Depois destas primeiras comparações, os resultados são propagados para a realização de outra comparação para, assim, determinar o menor valor entre as



(a) *Min pooling* apresenta melhor resultado com objetos com valores mais baixos

(b) *Max pooling* apresenta melhor resultado com objetos com valores mais altos

Figura 2.6: Exemplos de operação de *pooling* considerando janelas de tamanho $Y \times Y$.

quatro entradas oriundas da estrutura FIFO. Esta mesma arquitetura pode ser facilmente adaptada para implementar a função de *max pooling* com a substituição dos componentes de comparação.

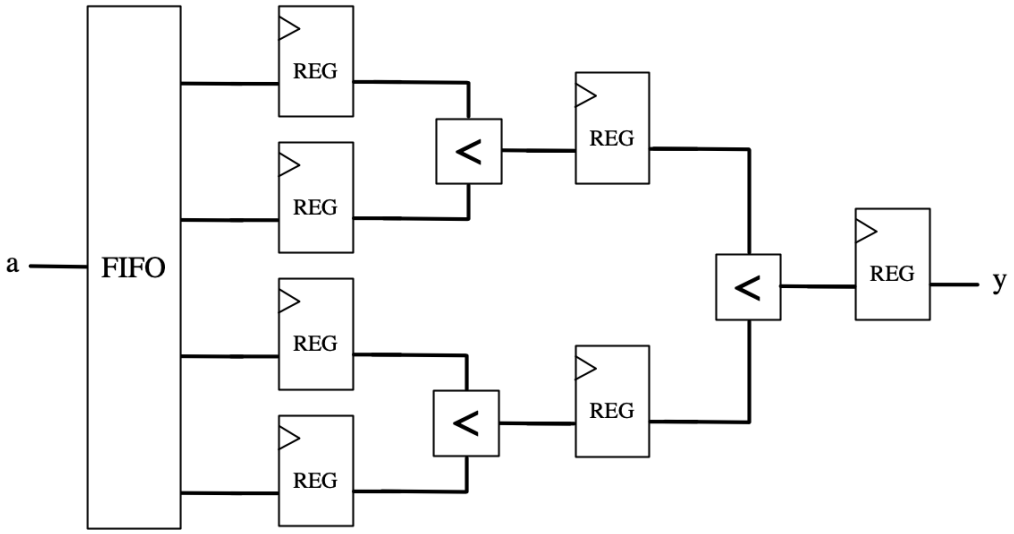


Figura 2.7: Proposta de uma arquitetura para *min pooling*.

Para calcular a média, para o caso de *average pooling*, o artigo [1] propõe o esquema apresentado na Figura 2.8. O elemento de processamento recebe os blocos de mapas de entrada sequencialmente. O circuito adiciona todos os pixels de entrada do canal atual e armazena os resultados intermediários em um registrador. Em seguida, o acelerador divide a soma pelo número de pixels do canal, indicado por s,

para calcular o valor médio. Para simplificar o projeto, é utilizada uma tabela (*lookup table* ou LUT) para implementar a divisão.

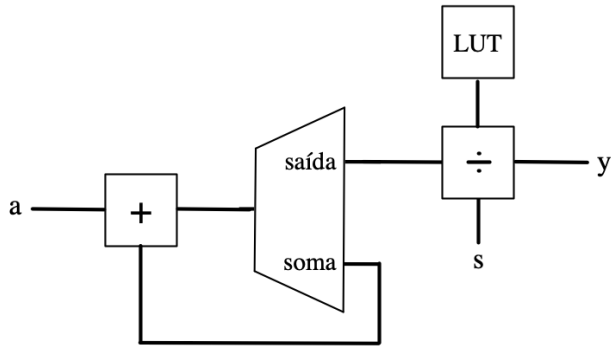


Figura 2.8: Proposta de um circuito de *average pooling* [1].

2.1.3 Função de Ativação

Uma função de ativação é utilizada para estabelecer uma relação não-linear entre a entrada e a saída de um neurônio. Sem uma função de ativação, uma rede neural é simplesmente um modelo de regressão linear que não seria capaz de resolver tarefas complexas como reconhecimento de imagem, por exemplo. Em resumo, a função de ativação habilita a rede neural, permitindo que ela aprenda e execute operações complexas [31].

A função de ativação empregada na LeNet-5 [25] é a ReLU (*rectified linear unit*), que calcula $f(x) = max(0, x)$, ou seja, elimina todos os elementos negativos, transformando-os em zero. Algumas arquiteturas utilizam funções sigmoidais, porém, elas são menos utilizadas em hardware devido a complexidade computacional.

2.1.4 Camadas Fully-Connected

Quando uma CNN é utilizada como classificador, as camadas *fully-connected* compreendem as últimas camadas da rede neural. Como seu nome indica, todos os seus neurônios estão conectados com todas as saídas da camada anterior. A saída da CNN resulta em um único vetor de valores que é utilizado como entrada da camada *fully-connected* apresentando um grande número de conexões. Esta densidade de conexões é justificada pelo fato de que cada neurônio deve possuir informações das características extraídas para poder atuar como classificador.

Este trabalho prioriza apresentar arquiteturas de hardware para CNN visando principalmente aceleração e economia de recursos e, desta forma, as camadas *fully-connected* não foram implementadas em hardware. Porém, para validar os resultados

das arquiteturas propostas, estas camadas foram implementadas em ambiente numérico. As funções implementadas recebem os dados exportados do hardware para realizar a classificação e permitir a avaliação e validação das arquiteturas.

2.2 Desafios

Apesar das vantagens citadas no início do capítulo, a execução de CNNs em FPGAs também apresenta 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 contribua para o bom desempenho das CNNs, 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 tanto a disponibilidade de recursos do dispositivo quanto o desempenho desejado.

Uma restrição muito importante em elaborar redes neurais fazendo uso exclusivamente de um FPGA é a quantidade limitada de memória interna que estes dispositivos possuem. Em algumas aplicações é necessário dispor de bancos de memória externos — e, consequentemente, alocar recursos do FPGA para constituir a interface com esta memória — ou simplificar as redes neurais. A simplificação das redes neurais pode ser realizada tanto em termos de tamanho quanto pelo uso de funções de ativação que sejam mais adequadas para implementação em hardware, tendo como objetivo a redução do custo computacional sem causar expressivo prejuízo na precisão de classificação da rede neural [31].

Considerando os requisitos para o sistema, o projeto das arquiteturas deve permitir a implementação de componentes configuráveis. Como o componente será instanciado na parte de lógica programável da plataforma FPGA, os dados a serem processados devem ser transferidos de algum elemento de memória para permitir que o componente tenha acesso aos dados e, assim, realizar as operações correspondentes. O acesso aos pesos e mapas de características é realizado através de endereços de leitura gerenciados pelo controlador, que também é responsável por alocar os elementos de processamento. O número de elementos de processamento instanciados definirá o nível de paralelismo que deve ser gerenciado pelo controlador. O número de instâncias destes elementos está diretamente associado ao consumo de recursos disponíveis.

Com os requisitos funcionais definidos, é desejável elaborar uma arquitetura que utilize componentes parametrizáveis e, consequentemente, que seja adaptável a qualquer camada da CNN. Uma vez especificado o nível de customização do componente,

também é importante analisar os requisitos de arquitetura deste componente pois a implantação no hardware está sujeita a restrições que devem ser consideradas durante a análise e projeto da arquitetura. Os dispositivos naturalmente impõem estas restrições às fases de projeto e implementação. A limitação de memória requer um design de hardware eficiente para alcançar um bom equilíbrio entre poder computacional e o consumo do recurso, por exemplo. A seguir são apresentados alguns elementos que devem ser avaliados durante a especificação e projeto da arquitetura. A solução de arquitetura adotada e a forma de implementação impactam significativamente no desempenho da CNN e no seu uso de recursos.

2.2.1 Memória

Considerando a grande quantidade de parâmetros envolvidos nas camadas convolucionais e a diferença entre a capacidade de armazenamento das memórias internas do FPGA e a disponível externamente, a escolha do local de armazenamento e o modo de acesso a estas memórias são muito importantes para evitar, ou ao menos mitigar, que este seja o ponto de baixo desempenho da CNN implementada.

Existem dispositivos que oferecem capacidade de armazenamento suficiente utilizando apenas a memória interna. Mesmo os FPGAs mais simples embarcam alguns megabits de memória RAM. Então, dependendo do dispositivo adotado e da CNN implementada, é possível fazer uso apenas da memória interna, dispensando o uso de armazenamento e, consequentemente, acessos à memória externa.

O armazenamento dos kernels internamente é conveniente por permitir a leitura com alto *throughput*. Conforme verificado e apresentado em [32, 33], a sobrecarga do acesso recorrente às memórias externas pode limitar o desempenho da execução das CNNs. A recorrência do acesso à memória se deve à quantidade de kernels utilizados nas convoluções com cada campo receptivo, pois estes parâmetros devem ser carregados da memória diversas vezes para a computação dos produtos internos.

A quantidade de dados necessários para realizar as convoluções é proporcional à quantidade de coeficientes de cada kernel. Considerando esta quantidade e o número de ciclos de clock por convolução é possível determinar a taxa mínima de leitura dos dados da memória e, desta forma, evitar que os processos de leitura sejam responsáveis pela perda de eficiência da rede neural. Portanto, os kernels devem ser armazenados na memória considerando o *throughput* adequado para a taxa de execução das convoluções de cada camada convolucional.

O armazenamento interno é realizado com a alocação de diversos blocos de memória com a flexibilidade de permitir mais liberdade para a configuração da porta de acesso e manipular diferentes profundidades e larguras de memória. Desta forma, é possível definir que a memória tenha mais endereços, cada um armazenando me-

nos bits, ou menos endereços armazenando mais bits, facilitando a manipulação dos dados carregados sem necessidade de preocupação com alinhamento de dados e desperdício de células de memória. Para aumentar o *throughput*, os blocos de memória podem ser utilizados como memórias individuais para armazenar segmentos de 16 bits, por exemplo, para os kernels e 32 bits para armazenamento dos valores de *bias*. Esta organização individual também permite a leitura simultânea destes valores.

Para armazenamento dos mapas de características podem ser utilizados buffers que reaproveitam endereços de memória e, consequentemente, reduzem a demanda por este recurso. Os buffers reutilizam espaços de memória ao permitirem a substituição das computações já utilizadas e, portanto, obsoletas, pelo armazenamento de novos dados computados. Cada instância de um buffer é conectada a dois outros componentes: um deles é responsável por fornecer dados a serem armazenados e o outro é responsável pela leitura e uso destes dados. Os componentes que consomem dados também realizam processamento e, desta forma, transferem o resultado para ser armazenado em outro buffer.

Os buffers podem ser utilizados em todas as camadas convolucionais, para armazenar o resultado das computações dos campos receptivos, e também nas camadas de *pooling*. Desta forma, o reaproveitamento dos endereços de memória é possível porque, na prática, é implementado um buffer circular, porém de três dimensões. Esta dimensão adicional é devida aos canais existentes em cada camada. No buffer circular existe a continuidade da última para a primeira linha de endereço, ou seja, elas são consideradas adjacentes. Para otimizar a implementação, a arquitetura destes buffers deve ser suficientemente versátil para suportar o armazenamento de dados variados, sejam eles resultados de uma operação de produto interno ou de uma operação de *pooling*. Portanto, o componente de hardware referente ao buffer deve ser descrito de maneira parametrizável, permitindo que os componentes possam ser instanciados e reutilizados com diferentes configurações.

Para possibilitar a realização das convoluções, a capacidade do buffer deve ser suficiente para armazenar a quantidade de dados do mapa de características equivalente a um campo receptivo. Isso implica em redução significativa no uso de recursos em relação ao armazenamento completo dos mapas de características.

O uso de buffer circular para armazenar parcialmente os resultados das camadas exige maior complexidade do controlador, pois, além da configuração do tipo de dados e dimensões dos mapas, são necessários sinais de controle suficientes para regular o seu comportamento. Estes sinais podem ser informações de uso, como buffer vazio ou cheio, e também informações para o endereçamento de escrita e de leitura. Uma forma de simplificação dos controles necessários é utilizar o buffer para armazenar completamente os mapas de características, ou seja, o resultado completo

de uma camada. Isto pode ser realizado diminuindo a generalização dos buffers. Utilizar componentes mais específicos contribui para a redução da complexidade de controle e gerenciamento, porém, tem grande impacto no reuso e, consequentemente, no uso de recursos.

2.2.2 Precisão Numérica

Outro desafio para implementação de CNNs em FPGA e que está intimamente relacionado tanto com o desempenho da rede neural quanto com o uso de memória é a precisão numérica usada. Uma solução que pode ser empregada para tentar equilibrar a precisão e o uso de memória ou de recursos do FPGA é a redução do tamanho dos operandos. Algumas técnicas utilizadas envolvem alterar a representação dos dados de ponto flutuante para ponto fixo, reduzir o número de bits para representar os dados, ou, ainda, realizar o compartilhamento de pesos.

Quando números reais precisam ser representados em hardware, normalmente é usado um formato de dados de ponto flutuante. A representação em ponto flutuante normalmente utiliza a notação científica, ou seja, um número base e um expoente. Por exemplo, $123,456$ pode ser representado como $1,23456 \times 10^2$. Essa representação resolve alguns tipos de problemas, por exemplo, números em ponto fixo possuem um espaço fixo de representação que limita o uso de números muito grandes ou muito pequenos [34]. O formato de ponto flutuante de precisão simples utiliza 32 bits e pode representar números usando a combinação de sinal (1 bit), mantissa (23 bits) e expoente (8 bits). Este formato de dados pode representar todos os números de aproximadamente $1,18 \times 10^{-38}$ a $+3,4 \times 10^{38}$ com pelo menos 6 dígitos decimais significativos. No entanto, a complexidade computacional das operações aritméticas com dados em ponto flutuante é alta e geralmente é necessário hardware especializado como, por exemplo, unidades de ponto flutuante conhecidas como FPU (*floating point unit*).

Normalmente as implementações de números em ponto flutuante, suas operações aritméticas, regras para arredondamento, tratamento de exceções (divisão por zero, por exemplo) etc., seguem o padrão IEEE 754 publicado originalmente em 1985 e atualizado em 2008. Este padrão é comumente denominado IEEE 754-2008 [35].

Alternativamente ao ponto flutuante de precisão simples, a revisão da norma em 2008 [35] definiu um formato de ponto flutuante de 16 bits. Conhecido como ponto flutuante de meia precisão (fp16), este novo formato foi especificado, inicialmente, para reduzir os requisitos de armazenamento e largura de banda das memórias. Os números reais podem ser representados com a combinação de sinal (1 bit), mantissa (10 bits) e expoente (5 bits). Esta forma de representação e a computação entre números utilizando meia precisão contribuem para a economia de recursos e dimi-

nuição da complexidade das operações porém, em contrapartida, dispositivos mais antigos não oferecem suporte em hardware para operações com este formato.

Outra abordagem para reduzir o tamanho dos dados e a complexidade das operações consiste em converter os valores dos operandos e das operações de ponto flutuante para ponto fixo. A precisão do número está relacionada com a quantidade de bits que representa a parte fracionária, por exemplo, utilizando 8 bits ($2^8 = 256$ possibilidades), a precisão dessa representação é dada por $1/256 = 0.00390625$. Convém destacar que o produto entre dois números de ponto fixo de N bits resulta em um número de $2N$ bits. Esta característica pode contribuir para a diminuição do erro nas operações de multiplicação pois o arredondamento ou truncamento para N bits não precisa ser realizado antes do término de uma série de operações.

A representação em ponto fixo dinâmico é constituída pelo bit de sinal, pela mantissa e pelo campo fracionário que representa um fator de escala que determina a posição da vírgula. Este formato de ponto fixo dinâmico também é conhecido como quantização linear. O campo fracionário utilizado na representação de ponto fixo dinâmico pode ser alterado de forma a enquadrar o número em uma determinada escala, uma solução interessante na implementação de CNNs pois a faixa de valores pode variar conforme as camadas e os filtros.

Utilizar ponto fixo dinâmico permite reduzir o número de bits dos operandos sem a necessidade de ajustar os valores dos pesos da rede neural. Desta forma, com a redução do número de bits para representar os dados, a área de armazenamento necessária diminui, e a largura de banda necessária para acesso à memória também diminui, com a consequente redução do consumo de energia. Porém, a redução no número de bits deve ser bem avaliada pois ela determina a precisão dos resultados que podem ser obtidos pela CNN. Ressalta-se, também, que optar por ponto fixo dinâmico em vez de ponto flutuante sem reduzir a largura de bits não reduz a área de armazenamento necessária e nem o consumo de energia [36].

2.2.3 Paralelismo

Existem algumas possibilidades para exploração do paralelismo nas CNNs, principalmente nas camadas convolucionais, onde está concentrada a maior parte do custo computacional. Este custo computacional das camadas convolucionais está associado à execução dos produtos internos, pois estas operações são realizadas repetidamente para extrair características para cada um dos kernels utilizados, formando os mapas de características. Além da recorrência, estas operações também apresentam um custo computacional alto pois compreendem, principalmente, operações de multiplicação.

As convoluções na CNN compreendem operações altamente recorrentes e com-

putacionalmente custosas, então, é de se esperar que a execução paralela das multiplicações 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. Esta abordagem também é conhecida por paralelismo intra-camada.

Uma técnica utilizada para acelerar a computação nas camadas convolucionais é o desenrolamento de laço (*loop unrolling*) aplicado a cada canal d descrito na Equação 2.1. 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 [37]. O fator de desenrolamento do *loop* determina a quantidade de iterações executadas simultaneamente.

Além da possibilidade do paralelismo intra-camada, também é possível paralelizar a execução das diferentes camadas. Este paralelismo inter-camadas é possível porque, a partir do momento em que uma determinada camada tenha extraído características suficientes para formar um campo receptivo, é possível iniciar o processamento da camada seguinte. Portanto, é formado um pipeline ao longo das camadas da rede neural, com cada camada consumindo dados da camada anterior e gerando dados para a camada seguinte. Mais à frente, na Seção 2.2.4, são apresentadas algumas considerações sobre o uso de pipeline.

As duas possibilidades de paralelismo apresentadas podem ser exploradas numa mesma arquitetura, executando as diversas operações aritméticas de cada camada e o processamento das cinco camadas da LeNet-5, por exemplo, em paralelo. Considerando que as camadas convolucionais incorporam uma quantidade significativa de operações de MAC e que os dispositivos FPGA possuem multiplicadores e capacidade de paralelismo finitos, o nível de paralelismo entre as camadas deve ser escolhido cuidadosamente visando a eficiência da implementação. O número de multiplicações simultâneas está diretamente ligada ao consumo de recursos do FPGA, pois quanto mais multiplicações são executadas em paralelo, mais multiplicadores existentes no hardware são necessários. Desta forma, é importante balancear o paralelismo (e também o pipeline) das camadas para evitar desperdício de recursos do FPGA e minimizar o tempo de execução. Esta afirmação pode parecer óbvia porém convém ressaltar que normalmente no paralelismo inter-camadas também se encontra a execução simultânea dos canais de cada uma das camadas. Considerando, ainda, que o número de canais pode aumentar a cada camada convolucional, então, existe um grande número de operações a serem realizadas em paralelo e o balanceamento deve ser bem avaliado na definição da arquitetura.

2.2.4 Pipeline

Para acelerar a execução da CNN, é importante avaliar a estrutura das camadas convolucionais, pois, dependendo da arquitetura utilizada, o processamento realizado nestas camadas pode concentrar acima de 90% do custo computacional [38]. O número de convoluções de cada camada equivale ao produto da quantidade de campos receptivos e da quantidade de kernels utilizados em cada campo receptivo. É importante ressaltar que cada camada convolucional depende dos dados originados pelas camadas anteriores ou pela imagem de entrada, no caso da primeira camada.

A estrutura sequencial entre as camadas apresenta uma oportunidade para implementação de um pipeline formado pelas camadas da CNN, sendo este pipeline iniciado com o primeiro produto interno da primeira camada e finalizado com o último produto interno da última camada. O pipeline pode ser balanceado controlando a quantidade de ciclos de clock necessários para executar cada operação nas camadas convolucionais e de *pooling*. O número de ciclos de clock nas camadas convolucionais é consequência da quantidade de multiplicações executadas simultaneamente. A redução do número de multiplicações em paralelo implica maior quantidade de ciclos de clock para execução da convolução, porém, em contrapartida, menos recursos de hardware são utilizados pois os mesmos multiplicadores podem ser reaproveitados. O nível de paralelismo de cada camada convolucional deve ser determinado durante o seu planejamento.

Apesar da operação de adição também ser amplamente utilizada nas camadas convolucionais, as considerações são feitas em relação às multiplicações porque os multiplicadores demandam uma quantidade significativamente maior de recursos do FPGA [39]. Além disso, a quantidade de adições tem uma relação direta com a quantidade de multiplicações, sendo que, ao reduzir a quantidade de multiplicações executadas por ciclo de clock, também se reduz a quantidade de adições.

O balanceamento do nível de paralelismo de cada camada é importante para a definição do pipeline, porque qualquer camada que execute suas operações aritméticas em um tempo muito inferior estaria contribuindo para o desperdício de recursos do FPGA, realizando operações rapidamente e depois aguardando a conclusão da computação das camadas anteriores. Portanto, o trabalho de balanceamento deve evitar ou minimizar que o processamento de alguma camada diminua o *throughput* geral da rede neural e que recursos de hardware sejam subutilizados.

O uso de pipeline é complementar ao processamento em lote, onde um conjunto de entradas é processado de uma vez. Enquanto o processamento em lote foca na eficiência ao processar um conjunto de dados em uma única interação, o pipeline se concentra na otimização temporal ao permitir que múltiplas entradas sejam processadas pelas diferentes camadas da rede neural simultaneamente, aumentando a

eficiência computacional e potencialmente diminuindo o tempo total de processamento.

2.3 Hardware Comum para as Arquiteturas Propostas

Neste trabalho, quatro arquiteturas de hardware são propostas para implementar a CNN LeNet-5 [25]. Esta CNN foi adotada devido ao seu tamanho e complexidade, pois permitem a sua implementação em dispositivos FPGA de baixo custo e menor capacidade de processamento. Ela potencialmente evidencia de forma significativa o compromisso entre custo e desempenho, facilitando a implementação e verificação de níveis de paralelismo, por exemplo, e apresentando aplicações práticas na área de visão computacional em sistemas embarcados. A saída da HNN, representada pelo conjunto de saídas da camada convolucional mais profunda, compreende 120 features (características) representadas em formato de ponto fixo de 16 bits. A representação adotada está principalmente no padrão (16,8), sendo oito bits representando a parte inteira e oito bits para a fração, com o bit mais significativo reservado para o sinal. O uso deste padrão foi verificado na implementação de uma arquitetura de referência e se mostrou adequado para garantir a precisão, ao comparar com os resultados obtidos em uma implementação em software, e evitar a ocorrência de *overflow* nas operações aritméticas. A Figura 2.9 apresenta a arquitetura do *hardware-in-the-loop* empregada para os quatro projetos.

O software de controle do sistema de processamento compreende um programa executável escrito em linguagem C para carregar as imagens da base de dados, transferi-las para o dispositivo FPGA, receber as features (características) extraídas pela HNN e alimentar as entradas das camadas de classificação (*fully-connected*) implementadas em ambiente numérico. Como o objetivo deste trabalho é avaliar o compromisso custo×desempenho da CNN utilizando as arquiteturas propostas, as camadas densas não estão incluídas nos projetos. Adicionalmente, os pesos são implementados utilizando elementos lógicos do FPGA, esta abordagem utiliza multiplexadores ou outros circuitos combinacionais roteados diretamente para o VCC e GND para definir as constantes. Para evitar o uso de recursos lógicos e de roteamento necessários para a implementação dos pesos, as camadas de classificação foram implementadas como parte de um software externo, que é usado para validação dos resultados e compreendem uma rede neural de duas camadas densas com topologia 120-84-10. Os pesos utilizados na rede densa e também nas HNNs, incluindo os valores de bias, são obtidos da rede neural implementada em software e treinada em [40], utilizando ponto flutuante.

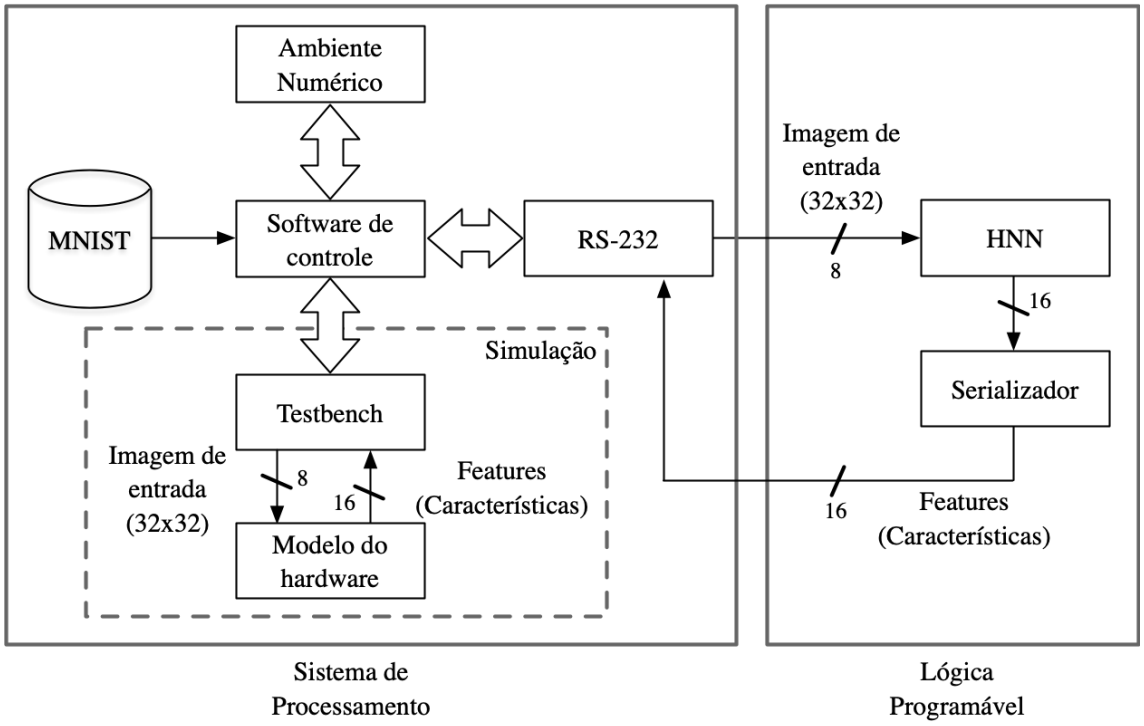


Figura 2.9: Arquitetura empregada para implementação e teste dos projetos propostos.

A arquitetura de *hardware-in-the-loop* é utilizada para validar os resultados e a aplicação prática dos projetos em um dispositivo FPGA de baixo custo. O FPGA utilizado é da família Altera Cyclone II 2C70 [41] empregado na placa de desenvolvimento DE2-70 [42]. Este FPGA utiliza tecnologia de fabricação de 90 nm e os principais recursos disponíveis são: 68.416 elementos lógicos, 150 multiplicadores de 18×18 bits e 1,15 Mbits de memória RAM.

O sistema de processamento também inclui um ambiente para a simulação do código VHDL implementado. O arquivo de *testbench*, escrito em VHDL, permite a especificação comportamental e facilita a interação com os projetos. O arquivo de *testbench* é o módulo principal do ambiente de simulação e contém sinais locais que são utilizados como entrada e saída das HNNs implementadas em RTL, além de instanciar os módulos integrantes de cada arquitetura proposta. Ao empregar a simulação, é bastante facilitada a verificação de contadores para contar o número de ciclos de clock, por exemplo.

É relevante destacar que a HNN no bloco de lógica programável da Figura 2.9 consiste no *bitstream*[1] gerado a partir da síntese dos códigos VHDL de cada uma das arquiteturas. Enquanto isso, o Modelo de hardware no bloco de simulação representa a execução da simulação dos códigos VHDL que especificam o hardware.

[1] *Bitstream* é a sequência de bits que compreende a configuração e define a operação do dispositivo FPGA. O arquivo de *bitstream* é resultado da síntese de um projeto.

2.4 Considerações Finais

Neste capítulo foram apresentados a estrutura e os principais elementos de uma CNN com a indicação de alguns desafios para a elaboração de arquiteturas de hardware. Adicionalmente aos requisitos para a realização de operações aritméticas como multiplicação e adição, acesso à memória, controladores que também possuem a função de configurar os componentes, ainda existem dois outros requisitos importantes que devem ser avaliados: a representação numérica e o uso de pipeline.

O formato e a quantidade de bits utilizados para representar números no FPGA determinam a complexidade das operações e o consumo de recursos, sejam de memória ou elementos lógicos. Já a definição de arquitetura capaz de operar em paralelo, seja entre as camadas (pipeline) ou intra-camada, possui grande impacto no desempenho da implementação. Alguns trabalhos relacionados, contendo soluções elaboradas e difundidas na literatura, são apresentados no próximo capítulo.

Para este trabalho foi adotada uma abordagem de projeto em nível de arquitetura. Ou seja, este trabalho contempla a definição de um projeto de um acelerador, e a sua otimização, com o objetivo de alcançar um desempenho melhor, consumindo menos recursos, de forma a permitir a utilização de um dispositivo FPGA de entrada e de baixo custo.

Capítulo 3

Trabalhos Relacionados

Diversos algoritmos e arquiteturas foram estudados em trabalhos científicos, com o objetivo de encontrar soluções que combinem precisão nos resultados da CNN, tempo de inferência reduzido e baixo consumo de energia, para citar algumas das características desejadas.

A implementação em hardware da extração de características possibilita a aceleração da computação, mas o grande número de operações e parâmetros necessários para o processamento de uma CNN aumenta a exigência da computação e do armazenamento dos resultados intermediários. Então, considerando as restrições impostas pelo hardware é possível destacar dois objetivos principais para a implementação eficiente de uma CNN: reduzir a complexidade e o uso de recursos. Para alcançar estes objetivos, alguns esforços de implementação devem ser adotados para mitigar as restrições e os desafios apresentados no Capítulo 2.

3.1 Otimização do Uso de Memória

A alocação de espaço de memória para armazenar um grande número de pesos e resultados intermediários é um grande desafio uma vez que a memória disponível internamente aos FPGAs costuma ser bastante limitada. Para estes casos, o uso de memória externa é inevitável, porém os sucessivos acessos à memória externa resultam em grande latência e aumento do consumo de energia. Nos casos em que não são implementadas otimizações nas camadas convolucionais, ao operações dos MACs resultam em muitos acessos à memória. Cada uma destas operações de MAC pode exigir dois acessos de leitura e um de escrita à memória [26].

Com propostas para redução do uso de memória, o trabalho apresentado em [43] implementa um buffer para armazenar os mapas de características de entrada e de saída das camadas. Então, quando há sobreposição entre os filtros e a entrada dos diversos canais é possível fazer reuso dos filtros. Essa reutilização é possível pois, quando ocorre a sobreposição, os pesos são os mesmos e as computações que ocor-

rem em uma região da entrada também podem ser usadas para outas regiões que compartilham o mesmo conjunto de pesos. Um acelerador centrado na memória foi desenvolvido em [44] para melhorar o desempenho, sem aumentar a largura de banda da memória, por meio de uma hierarquia de memória flexível. O acelerador usa buffers internos para reduzir o número de acessos externos por meio da reutilização de dados. Adicionalmente, para garantir reconfigurabilidade e capacidade de programação, um *cluster* do tipo SIMD (*single instruction, multiple data*) composto por componentes MAC é usado para acelerar as convoluções. Baseado na implementação no FPGA Virtex-6 trabalhando na frequência de 150 MHz, o acelerador alcança uma redução de recursos do FPGA de até 13×, mantendo o desempenho.

Ainda no contexto de reduzir o uso de memória externa, a proposta de [45] acelera CNN para aplicações de reconhecimento de voz. A transferência total de dados entre as camadas é reduzida utilizando uma proposta de processamento de camadas multi-ConvNet (modelo para explorar a localidade de dados de uma convolução) e evita o uso de memória externa para armazenar dados intermediários entre as camadas. As matrizes de entrada, de saída e de pesos fazem uso de buffers internos. Esta implementação utiliza uma ferramenta HLS para transformar código C++ em hardware e facilitar a composição do pipeline entre as unidades aritméticas e transferências com a memória. O projeto ainda emprega transformações de *loop* para reordenar cálculos e reduzir a transferência de dados. Implementado no dispositivo Virtex-7, o projeto alcançou 28% de economia na transferência de dados entre as camadas convolucionais e a memória.

O aumento na taxa de reconhecimento de objetos é tratado em [46]. Lá, a transferência de dados é dividida em alguns estágios. Neste trabalho são empregadas estruturas FIFO para conectar as camadas convolucionais. A função ReLU é usada como função de ativação. Acumuladores seguidos de buffers são usados para obter a soma de cada mapa de recursos e armazenar os resultados da computação para servir à entrada da próxima camada, aumentando a eficiência de acesso à memória.

Para melhorar a taxa de transferência entre o FPGA e a memória externa, alguns trabalhos utilizam técnicas para o tratamento dos dados ou formas diferenciadas de escrita e leitura. Os autores de [26] avaliam técnicas de compressão para reduzir a utilização de memória externa e verificam como o arranjo dos dados na memória impacta no desempenho da rede neural. Outro trabalho que explora a organização dos dados na memória como forma de acelerar os acessos de leitura e escrita é proposto em [47].

Os autores de [48] evitam carregar parâmetros de uma memória externa, armazenando-os em uma memória interna ao FPGA. Na implementação, eles adotam um padrão paralelo-serial para aumentar a taxa de transferência. No entanto, esta estratégia não aproveita ao máximo o paralelismo disponível na CNN, assim

como diferentes camadas não operam simultaneamente. Eles implementaram uma rede neural de pequena escala que realiza o reconhecimento de dígitos no dispositivo Xilinx XC7Z045.

3.2 Aumento da Precisão

Reduzir o número de bits utilizados para representar os operandos nas computações é uma forma direta de reduzir o tamanho das unidades aritméticas e acelerar o processamento. Diversas implementações substituem as unidades de ponto flutuante de 32 bits por representação em ponto fixo. A representação em ponto fixo de 16 bits é amplamente utilizada [49, 50]. O trabalho desenvolvido em [9] propõe o projeto e a implementação, em FPGA, de CNN com representação de ponto fixo que permite o reconhecimento de números desenhados à mão. Embora tenha sido implementado em um FPGA de entrada e de baixo custo, o projeto é escalável para dispositivos maiores. No trabalho também são sugeridas modificações no conjunto de imagens da MNIST, para que representem imagens oriundas de uma câmera, e os valores dos pesos utilizados são transformados para uma escala entre -1 e 1.

Duas estratégias para uso do ponto fixo foram avaliadas em [9]: truncamento após cada operação e precisão total (onde o truncamento é realizado apenas ao final da convolução). Para a estratégia de truncamento após cada operação, utilizar representação em 12 bits foi suficiente para melhorar o desempenho, porém com perda significativa na precisão e com *overhead* no uso de memória. Desta forma, foi verificada a estratégia de precisão total com arredondamento após a convolução. Nesta segunda implementação foram utilizados 17 bits e, consequentemente, houve aumento de recursos utilizados, mas a precisão alcançada foi similar à implementação de referência realizada em software. Com estas abordagens, o trabalho apresentado em [9] alcançou o processamento das imagens de dígitos em tempo real com média de classificação de 150 imagens/s. No dispositivo utilizado, esta média equivale a aproximadamente 230.000 ciclos de clock necessários para classificar uma imagem.

O trabalho [51] implementa uma DNN (*deep neural network*) para mitigar a interferência em links de fibra ótica. Para reduzir a complexidade de hardware para equalização, os autores investigaram o uso de paralelismo e de representação em ponto fixo. Foram configurados alguns cenários comparando as taxas de erro utilizando pesos da rede neural representados por: ponto flutuante de 16 bits, ponto fixo de 8, 5, 4, 3 e 2 bits. Com a redução dos pesos da DNN de 8 para 4 bits, os autores conseguiram obter uma redução de até 40% no uso de LUT com pequena redução na sensibilidade.

Os autores de [52] avaliaram a exequibilidade da utilização de ponto flutuante (32 bits) na implementação das diversas operações utilizadas nos algoritmos de

backpropagation e compararam com outra implementação utilizando ponto fixo de 16 bits. Os autores concluem que a implementação em ponto flutuante sem utilização de recursos específicos como DSP ou FPU é possível, porém a representação de 16 bits apresenta o melhor *trade-off* entre precisão e área utilizada, ocupando menos recursos que a implementação que utiliza ponto flutuante.

Utilizando um coprocessador programável para realizar a comunicação com a memória externa, os autores de [53] buscam acelerar um modelo de CNN. O coprocessador usa a memória externa como um rascunho para gerenciar o grande volume de dados intermediários entre as camadas da CNN. A carga de memória neste trabalho é reduzida pelo uso de dados de baixa precisão, compactando várias entradas e saídas em cada operação de memória. Eles utilizam ponto fixo de 20 bits para pesos do kernel e 16 bits para todos os outros valores. A arquitetura do sistema é organizada em paralelo como um *cluster* de elementos de processamento vetorial, compostos por componentes de convolução de duas dimensões. Neste trabalho, o paralelismo é usado principalmente em mapas de recursos e nas convoluções. No Xilinx Virtex-5 LX330T operando a 115 MHz, o coprocessador alcançou taxa de 6 FPS (*frames per second*), usando representação de 16 bits e consumindo 11 W.

Um estudo sobre o efeito da representação numérica, ou seja, da computação com dados de precisão limitada, no treinamento de redes neurais foi conduzido em [54]. Este trabalho baseia-se na ideia de que a tolerância do algoritmo, ao nível de ruído, pode ser aproveitada para simplificar os requisitos de hardware. O sistema consiste em uma matriz de multiplicadores, uma memória interna configurada como fila FIFO e outros controladores que coordenam a transferência de dados e a comunicação com a memória externa. Para imagens de entrada com resolução 28×28, e implementada no Kintex K325T, a ferramenta de síntese estimou uma frequência máxima de operação do circuito de 166 MHz e um consumo de energia de 7 W.

Em relação às implementações que utilizam ponto flutuante, o trabalho [55] propõe a implementação em FPGA de um acelerador de CNN utilizando ponto flutuante de baixa precisão (8-bits LPFP, *low-precision floating-point*). Sem retreino da rede neural, a perda de precisão fica entre 0,3% e 0,5% comparada com a implementação de referência em 32 bits. Esta proposta customizada permitiu a implementação da multiplicação LPFP utilizando um MAC de 4 bits e um somador de 3 bits. Desta forma, quatro multiplicadores LPFP de 8 bits podem ser implementados usando um DSP de um FPGA da família Kintex 7. Embora uma multiplicação de ponto fixo de 16 bits ocupe o mesmo recurso de DSP, nenhum outro recurso como LUT ou elementos lógicos são necessários.

3.3 Uso de Paralelismo e Pipeline

A técnica de utilizar pipeline permite, em uma sequência de tarefas, que uma tarefa seja executada sem que a anterior esteja completada. Ou seja, a execução de uma tarefa pode ser iniciada assim que os dados necessários estiverem disponíveis. As redes neurais convolucionais consistem em uma estrutura de sucessivas camadas inter-dependentes. Ao utilizar uma estrutura de pipeline é permitido que o processamento de uma camada seja iniciado antes da conclusão da camada anterior.

Dispositivos como FPGA são adequados para processamento utilizando pipeline e operações realizadas paralelamente, porém, normalmente, exigem um número substancial de elementos de DSP e de memória [56]. Em [57] é proposta uma arquitetura de acelerador utilizando pipeline entre as camadas, onde todas as camadas da CNN podem ser processadas concorrentemente utilizando os recursos do FPGA. Um framework utilizando HLS para aceleração do processamento baseado em pipeline é proposto por [58]. A proposta dos autores é facilitar o ajuste do nível de paralelismo de acordo com o dispositivo utilizado.

Buscando um diagnóstico mais detalhado das demandas computacionais de uma determinada CNN e buscando obter uma configuração otimizada na forma de maximização das operações paralelas, o trabalho [13] avalia os níveis de paralelismo no processamento dos kernels e a sua relação com o uso de memória. A utilização de coprocessador para otimizar dinamicamente os recursos e memória necessários de acordo com a aplicação da CNN é apresentada em [59]. Esta otimização dinâmica tem o intuito de oferecer um sistema adaptativo que priorize, tanto quanto possível, o paralelismo e evite a geração de dados intermediários — que, por sua vez, demandam memória. A proposta de [60] é basicamente oposta ao anterior, ou seja, busca reduzir o uso de memória propondo um algoritmo de reuso de buffer interno às camadas.

O acelerador apresentado em [13] foi inicialmente proposto em [61], onde os autores buscaram considerar todas as fontes de paralelismo. É apresentado um mecanismo de convolução paralela (os autores chamaram este mecanismo de *parallel convolution engine* ou PCE) composto por blocos de multiplicadores e seus respectivos somadores para explorar o paralelismo intra-kernel em cada convolução. Então, uma combinação de PCE com seus somadores correspondentes executa o paralelismo inter-kernel. A disposição espacial é aproveitada nas operações utilizando o kernel e os mapas de características para gerenciar a transferência de dados e melhorar o desempenho. Os buffers no chip são usados para armazenar os dados intermediários. Implementado no FPGA VX485T e com precisão de ponto flutuante de 32 bits, o acelerador alcançou um desempenho de até 1,9× mais rápido em comparação com o trabalho de referência.

O aumento do paralelismo foi conseguido em [62] implementando um processador para aprimorar o controle da transferência de dados no FPGA. Este processador é composto por uma unidade de controle, aritmética vetorial paralela, unidade lógica, unidade de controle de E/S e uma interface de memória. A unidade de controle é usada para sequenciar as operações conectadas com os demais componentes. A paralelização foi obtida por meio de um árbitro que acessa o mesmo local de memória simultaneamente por meio de 8 portas com buffer do tipo fila FIFO. Todas as operações foram executadas com precisão de ponto fixo de 16 bits.

Os autores de [63] apresentam um acelerador CNN com todas as camadas trabalhando simultaneamente utilizando pipeline para aumentar a taxa de transferência. Um método de computação em lote é implementado e aplicado nas camadas *fully-connected* para aumentar a utilização da largura de banda da memória. Entre cada camada, existem dois buffers onde a camada anterior pode ler e armazenar dados a partir de um dos buffers, enquanto a próxima camada carrega os dados do outro buffer. Estes buffers também são utilizados para armazenar os dados intermediários, os dados de entrada e pesos, reduzindo assim a carga de trabalho de acesso aos dados. Esta tese adota os paralelismos intra-camada e inter-camadas conforme proposto em [62].

3.4 Considerações Finais

A implementação de CNN em FPGA apresenta diversas vantagens, como um desempenho notável, alta eficiência energética e a flexibilidade da reconfiguração, que permite adaptar a arquitetura para a CNN em questão. Entretanto, apesar do aumento na quantidade de blocos lógicos e na capacidade de memória dos FPGA, a execução de CNN em FPGA ainda possui limitações em comparação com as GPUs e as TPUs, como a largura de banda para acesso às memórias e frequência máxima de clock.

Analisando a diversidade de soluções experimentadas e aplicadas em várias implementações de CNN em hardware apresentadas neste capítulo, é evidente a ampla gama de abordagens disponíveis para enfrentar ou atenuar os desafios inerentes ao processamento de grandes volumes de dados. No geral, essas arquiteturas têm buscado alcançar o máximo desempenho possível. Essa melhoria pode ser obtida por meio de arquiteturas mais sofisticadas e otimizadas, ou optando por um dispositivo FPGA de maiores capacidades, com uma maior quantidade de recursos à disposição e, ocasionalmente, fazendo uso de ferramentas de síntese de alto nível (HLS).

Os trabalhos relacionados sintetizam os principais objetivos relacionados ao uso de memória RAM na implementação de redes neurais em FPGA, que são: diminuir o uso da memória externa e aumentar a taxa de transferência entre a memória e o

FPGA. Diversas técnicas podem ser implementadas para que a necessidade de uso de memória RAM externa ao FPGA não seja o limitador de desempenho da rede neural. Entre as técnicas abordadas, podemos destacar o uso de buffers e filas FIFO para armazenar pesos e resultados intermediários, compressão de dados e reúso de pesos.

Quanto às técnicas para conseguir a mesma precisão obtida nas implementações em software, podemos destacar a representação numérica. Alguns trabalhos apresentam discussões entre o uso de representação numérica em ponto fixo e ponto flutuante. Estas discussões são especialmente importantes quando a implementação da rede neural em FPGA é feita em RTL. A definição da representação numérica é apenas o primeiro passo, ainda deve-se considerar a largura de bits que será adotada. Desta forma, alguns trabalhos verificam a precisão alcançada ao utilizar ponto flutuante de tamanho reduzido ou de baixa precisão, enquanto outros trabalhos verificam o comportamento das redes neurais ao empregar ponto fixo de tamanho reduzido ou de tamanho variável.

O balanceamento entre o uso otimizado dos recursos existentes no FPGA, com a consequente redução do consumo de energia, e o tempo de processamento pode ser realizado ao empregar algum nível de paralelismo intra-camada e de pipeline entre as camadas. Alguns trabalhos relacionados buscam realizar operações em paralelo, entre os quais destacamos, o processamento concorrente de todas as camadas da CNN, operações com o kernel realizadas em paralelo, acesso simultâneo à memória e uso de elementos como buffer para conectar as camadas da rede neural em pipeline.

As CNNs demandam consideráveis recursos computacionais e de memória, o que torna desafiadora a sua implementação em FPGA de baixo custo e com recursos limitados. Nas arquiteturas desenvolvidas nesta tese, escolhemos adotar a implementação usando RTL para garantir um maior controle e uma maior flexibilidade na criação dos componentes, possibilitando assim a utilização de dispositivos FPGA acessíveis e de baixo custo. Essa abordagem nos permite manter a independência em relação a dispositivos ou fabricantes específicos. No Capítulo 4 é apresentada a primeira arquitetura proposta, contendo a descrição dos seus principais elementos, os caminhos críticos e os resultados de custo e desempenho.

Capítulo 4

Arquitetura com Memória

A arquitetura com memória envolve a concepção e a implementação em hardware de uma CNN que foi previamente treinada em ambiente de software. As arquiteturas desenvolvidas nesta tese tomam como base os tipos de camadas e os pesos estabelecidos na CNN treinada. A capacidade de examinar os resultados de classificação obtidos no software não apenas possibilita a validação dos resultados na nova plataforma, mas também permite uma avaliação crítica da utilização de recursos, abrindo espaço para eventuais ajustes ou a criação de novas arquiteturas. Isso pode ser direcionado tanto para melhorar a precisão, como para aprimorar o desempenho ou otimizar a utilização de recursos.

A avaliação dos resultados desta arquitetura implementada em um FPGA de entrada considera, desde o seu conceito inicial, a representação numérica empregada e a utilização da memória interna. Considera-se a utilização de um FPGA que não dispõe de recursos especializados, como FPU ou CPU embarcados. Mesmo sem o uso destes elementos, a arquitetura é elaborada para não perder precisão utilizando representação numérica com quantidade reduzida de bits, onde as operações aritméticas são computacionalmente eficientes, e fazendo uso exclusivo dos blocos de memória RAM disponíveis internamente no FPGA.

Esta arquitetura com memória é formalmente chamada de *Memoryful Architecture* ou MFA. A denominação está associada à característica do projeto contemplar toda a memória necessária para armazenar os mapas de características e, desta forma, o fluxo de dados entre as camadas ser realizado por meio destas memórias.

4.1 Componentes Básicos

As arquiteturas em hardware apresentadas neste trabalho fazem uso de dois componentes básicos para executar operações aritméticas: MAC (da operação de *multiply–accumulate*) e MAX (da operação de *max pooling*), e um componente de memória somente de leitura (ROM, do inglês *Read-Only Memory*). Os componentes MAC

são responsáveis pelas multiplicações e adições que compõem os produtos internos, e os componentes MAX são usados para executar as operações de *max pooling* e para a implementar a função de ativação ReLU. O projeto destes componentes é ligeiramente diferente do usual. Além dos registradores destes componentes serem inicializados com valores específicos, uma função de truncamento também é adicionada no MAC. A Figura 4.1 apresenta a microarquitetura do MAC enquanto o Algoritmo 1 descreve o seu funcionamento. Este algoritmo é iniciado pelo sinal mac_start e recebe os parâmetros definidos no Estado 4 do Algoritmo 3. Desta forma, são atribuídos valores para d, $data\_in$ e $kernel$. O sinal $reset$ é definido como mac_rst no Estado 6 do Algoritmo 3. Note que quando $reset = 1$, o registrador sum é inicializado com o valor de *bias* do canal. Ambos os valores $data\_in$ e $kernel$ são de 16 bits, e sum é de 32 bits. As larguras, em bits, de todos dados utilizados no componente são exibidas na figura.

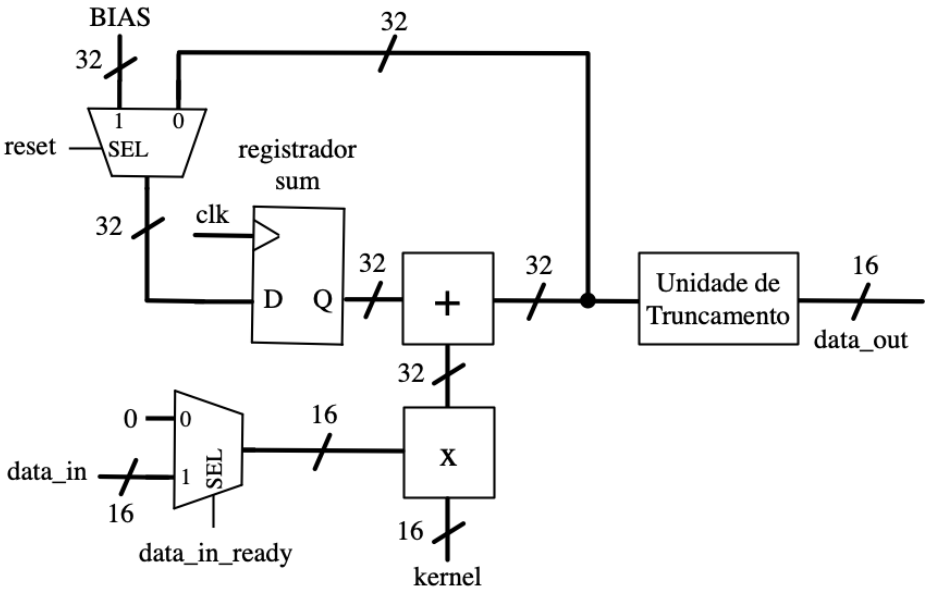


Figura 4.1: Microarquitetura do MAC com um bloco de “truncamento" adicionado à saída.

Algorithm 1 MAC$(d, data_in, kernel)$

if $reset = 1$ **then**
  $sum \leftarrow ROM\_B(d)$
else if $mac\_start = 1$ **then**
  $sum \leftarrow sum + data\_in * kernel$
end if
$data\_out \leftarrow truncate(sum, qi, qf)$
return $(data\_out)$

O componente MAC é usado principalmente para o cálculos dos produtos internos nas camadas convolucionais. O resultado de um produto interno iniciado na

linha y_1 e coluna x_1 (canto superior esquerdo), para cada canal d da camada convolucional C_1, pode ser expresso pela Equação 2.1. Nesta equação é possível verificar o uso das operações de multiplicação e adição. Como o produto de dois números de 16 bits resulta em um número de 32 bits, todos os barramentos internos do MAC devem ter 32 bits para manter a precisão, incluindo o valor inicial de BIAS conforme apresentado na Figura 4.1. O truncamento para 16 bits é realizado apenas na saída da função, conforme mostrado na Figura 4.1 e Algoritmo 1. A notação Q-Point [64] é usada para a representação numérica com o formato Q[qi].[qf], onde qi indica o número de bits inteiros e qf o número de bits fracionários. Portanto, o valor de *sum* é um número com sinal no formato Q16.16. Nas camadas C_3 e C_5, o truncamento para 16 bits é feito usando Q7.9, enquanto na camada C_1 é usado o formato Q2.14. Essa diferença ocorre porque os bytes da imagem de entrada são usados na parte fracionária do multiplicando utilizado no MAC.

O bloco MAX, detalhado na Figura 4.2, é inicializado com o menor valor negativo de 16 bits, ou seja, 8000h ou −32768. Como este componente realiza apenas comparações, todos os dados são processados em 16 bits. Para executar a operação de *max pooling*, uma máquina de estados finitos (FSM) simples controla a operação do componente MAX. A função ReLU, aplicada após as operações de MAX, utiliza o bit de sinal do resultado para controlar um multiplexador em que uma das entradas é o próprio resultado e a outra entrada recebe zero. As funções *max pooling* e ReLU são descritas no Algoritmo 2.

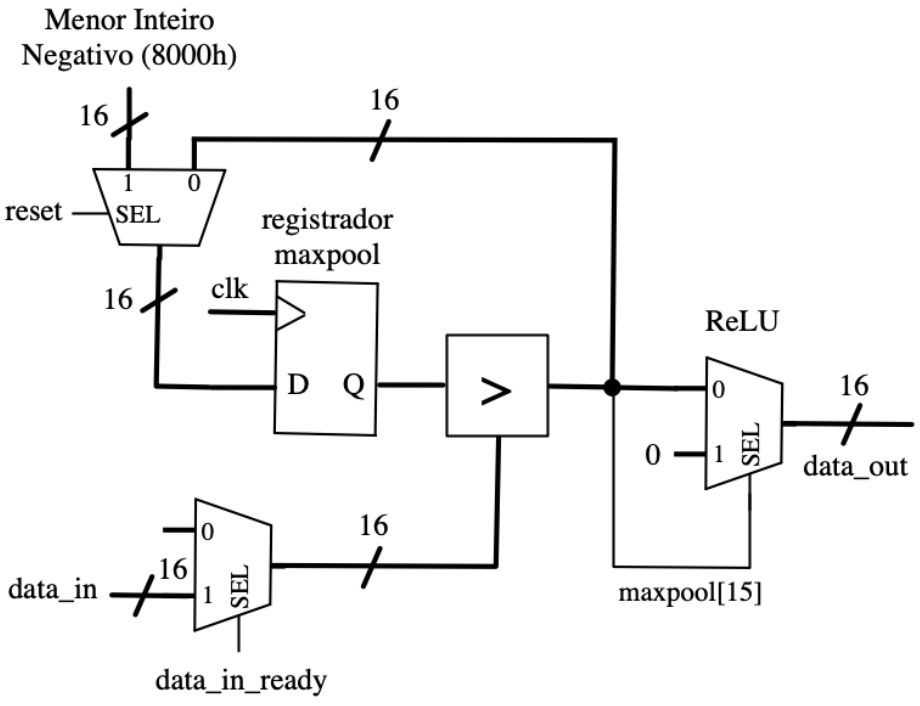


Figura 4.2: Microarquitetura de *max pooling* e ReLU.

Algorithm 2 Operação MAX($data\_in$) para a entrada $data\_in$

```
if reset = 1 then
    maxpool ← 8000h
else if data_in_ready = 1 then
    if data_in > maxpool then
        maxpool ← data_in
    end if
end if
if maxpool(15) = 1 then
    data_out ← 0
else
    data_out ← maxpool
end if
return (data_out)
```

No Algoritmo 2, quando o *max pooling* é redefinido, o registrador interno *maxpool* é inicializado com o menor valor negativo de 16 bits. Quando $data\_in\_ready = 1$, o valor do registrador é comparado com o valor de entrada $data\_in$ e o maior valor é armazenado em *maxpool*. Antes de retornar o valor de $data\_out$, é verificado o sinal (bit 15) do número armazenado no registrador *maxpool*. Se for igual a 1, então o valor de saída $data\_out$ é mantido igual ao conteúdo do registrador, caso contrário, é definido como zero.

A memória somente de leitura ROM_K armazena os kernels e a ROM_B armazena os valores de *bias*. Estas memórias somente de leitura são implementadas utilizando elementos lógicos, como multiplexadores, roteados para o VCC e GND para definir os valores dos pesos, e uma interface para permitir a leitura destes valores utilizando endereços. A vantagem dessa abordagem é que a operação de leitura dessas memórias pode ser realizada no mesmo ciclo de clock. As desvantagens são a dificuldade em prever o número de elementos lógicos necessários e o aumento considerável no tempo de síntese. Este aumento no tempo de síntese é devido, principalmente, à grande quantidade de roteamento necessário para o FPGA criar e configurar cada uma das constantes.

A Figura 4.3 apresenta os sinais necessários para o endereçamento destas memórias. A camada C_1, que possui kernel com profundidade $d = 1$, o valor de pos_d não é utilizado. Os demais sinais para endereçar a memória de kernel são o número do canal (d), definido durante a síntese e corresponde ao número da instância da camada, e o pos_k que é uma composição dos sinais X e Y, apresentados no Algoritmo 3, para endereçar posições específicas do kernel. O retorno são n kernels (um para cada posição do tensor) de 16 bits. O endereçamento da memória de *bias*, por depender apenas do número do canal, é significativamente mais simples, retornando n valores de *bias* de 32 bits.

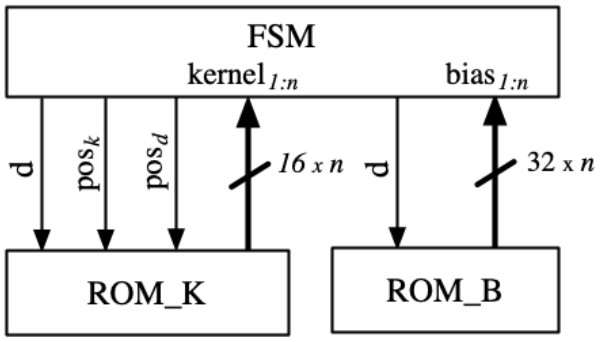


Figura 4.3: Microarquitetura das ROMs de kernel e *bias*.

4.2 Arquitetura

As camadas da rede na MFA são compostas essencialmente por três componentes: memória, unidade aritmética e unidade de controle. A Figura 4.4 apresenta o diagrama de blocos da arquitetura com a representação e relação entre estes componentes. Os blocos de memória RAM_{Si}, onde Si representa o tamanho da memória e é indexado pelo respectivo índice da camada $i = 1, 2, \ldots, 5$, possuem tamanhos distintos. A unidade aritmética compreende um componente MAC ou MAX. Os componentes de controle (FCtrl), como o nome sugere, são responsáveis por coordenar as etapas executadas em cada camada: escrita na memória local, operações aritméticas e leitura do resultado.

Os componentes de controle FCtrl são implementados usando uma máquina de estados finitos (FSM). Observe que cada camada inclui um único controlador e quantos componentes de RAM, MAC e MAX forem necessários (um componente para cada canal da camada). Com exceção da memória RAM_{S1}, que utiliza dados de 8 bits, todas as outras memórias, assim como os barramentos de dados, usam largura de 16 bits.

Os números de MACs e blocos de RAM, bem como o tamanho do barramento de E/S, variam de acordo com o número de canais. Cada bloco RAM_{Si} possui o tamanho Si suficiente para armazenar os mapas de características, por exemplo, $S2$=28×28=784, $S3$=14×14= 196, $S4$=10×10=100 e $S5$=5×5=25 palavras de 16 bits. Como o número de canais da camada P_2 é seis, a memória total usada pela camada P_2 é $S2$×6=4704 palavras de 16 bits, e assim por diante para cada uma das camadas.

A camada C_1, adicionalmente à largura dos dados utilizados pela memória, também difere das demais por possuir apenas um bloco de memória, que é utilizado para armazenar a imagem de entrada, recebida por $pixel\_in$ na forma de bytes. A entrada é salva na RAM_{S1}, onde $S1 = 1024$ bytes, indicando o tamanho da imagem de entrada com 32×32 pixels em escala de cinza. O controlador determina os sinais que transferem os dados da RAM_{S1} diretamente para os seis componentes MAC, um

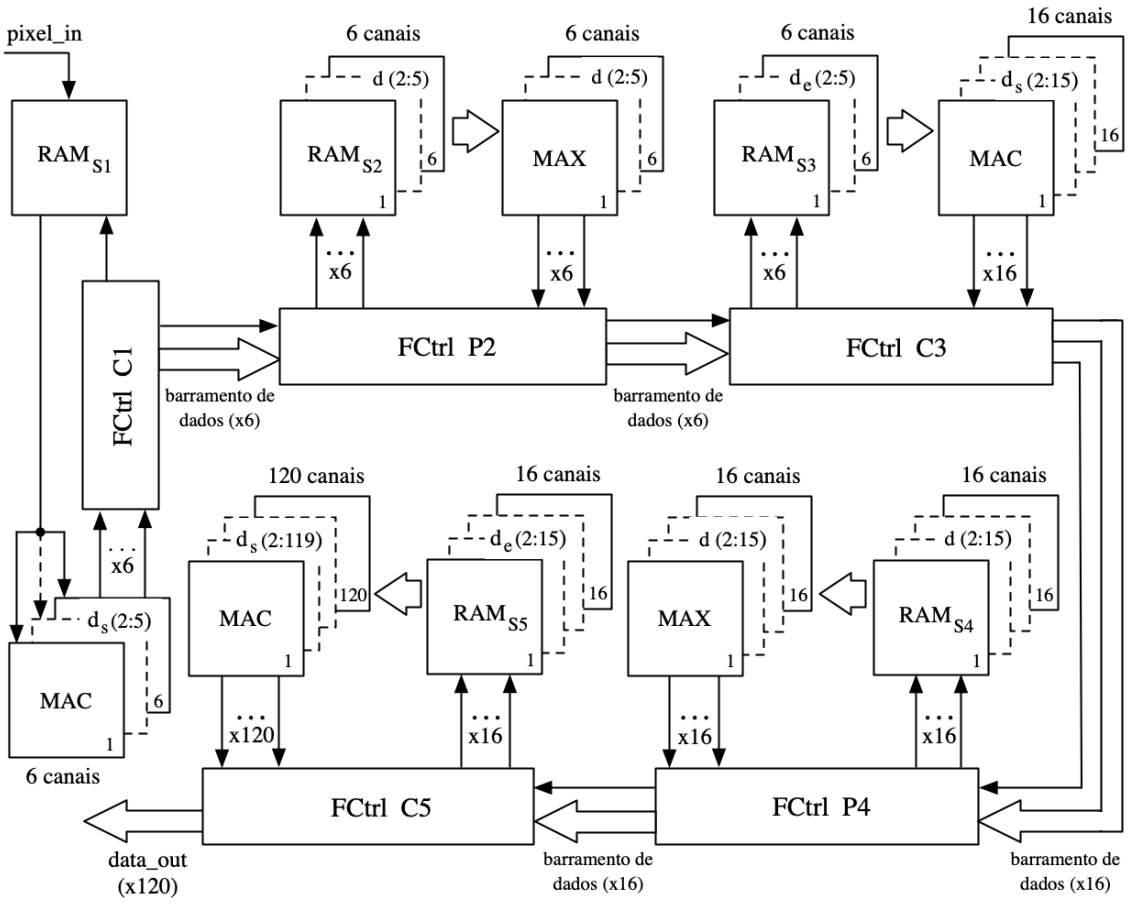


Figura 4.4: Diagrama de blocos da MFA.

para cada canal. Após o cálculo de cada produto interno, os dados dos seis MACs são transferidos para a próxima camada por meio de barramentos.

No diagrama do controlador FCtrl C1 apresentado na Figura 4.5, a FSM recebe dados de um contador, para endereçar as memórias RAM_{S1} e ROM_K, para cada operação MAC. A memória somente de leitura ROM_K é utilizada para armazenar os valores dos kernels. Cada MAC é inicializado com o valor obtido da memória ROM_B, que armazena os valores de *bias* da camada. Como já mencionado, a camada C_1 possui apenas um componente de memória RAM, que é utilizado para alimentar os seis MACs, conforme é indicado pelo barramento na parte superior da Figura 4.4. A saída da camada *data_out* consiste em seis barramentos, um para cada canal. Todas essas operações entre os canais são realizadas simultaneamente. O fluxo de dados da camada C_1 é detalhado no Algoritmo 3.

A FSM descrita no Algoritmo 3 começa no Estado 0 e possui oito estados. Quando o estado atual é o Estado i e o próximo estado não é explicitamente indicado, ocorre a transição para o Estado $i+1$. O Estado 0 redefine o endereço *ram_addr* da RAM, define o número de canais nesta camada (existem seis na camada C_1), define o tamanho do kernel em 5×5 ($k = 5$), define o tamanho da imagem I_s, e aguarda o

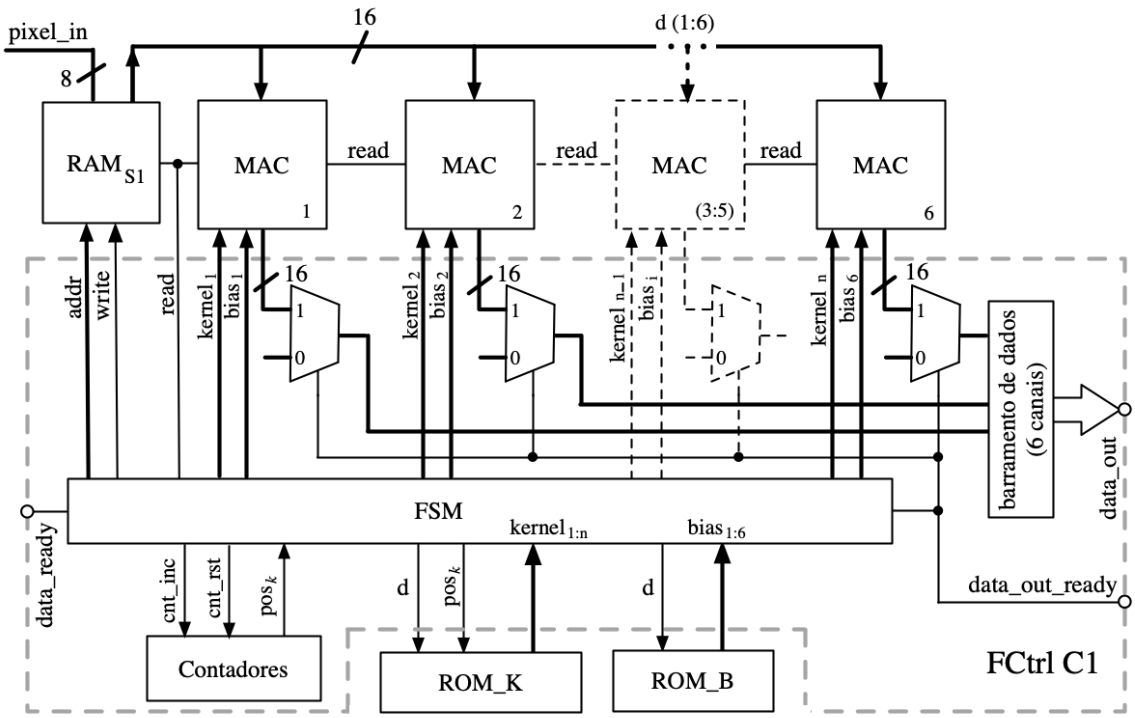


Figura 4.5: Microarquitetura da camada C_1 da MFA.

sinal *data_ready* que indica a disponibilidade da imagem de entrada. No Estado 1, cada pixel *pixel_in* da imagem de entrada é escrito na RAM, definindo o respectivo sinal *write*. A FSM permanece no Estado 1 até receber e armazenar todos os 1024 bytes da imagem na RAM. Esses dois estados são, portanto, responsáveis por receber e armazenar a imagem de entrada. Os próximos estados controlam o cálculo dos produtos internos até que todas as operações de convolução estejam completas. O Estado 2 redefine os sinais que controlam as coordenadas da imagem (x, y) e do kernel (X, Y). A computação realizada durante o Estado 3 para definir o valor *ram_addr* é uma conversão de um par de coordenadas para um endereço linear para a RAM. No Estado 4, os valores do kernel para cada canal d são lidos de uma ROM e as operações do MAC são executadas. As interconexões das ROMs de kernel (ROM_K) e de *bias* (ROM_B) são detalhadas na microarquitetura apresentada na Figura 4.5. O Estado 5 controla os contadores para endereçar todas as posições da imagem de entrada e do kernel. Quando um produto interno é concluído, os resultados do MAC são propagados para a saída *data_out*, o MAC é reinicializado para seu valor inicial (definido pelo sinal *mac_rst*) e a FSM retorna ao Estado 3. Quando uma convolução é concluída, o resultado é propagado para a saída e a FSM avança para o Estado 7, que é o estado final. O Estado 6 é acessado ao final do processamento de um kernel e habilita a saída dos MACs para escrita nas memórias.

Diferentemente da camada C_1, as camadas convolucionais C_3 e C_5 incluem um componente de memória em cada um dos canais de entrada. A microarquitetura

Algorithm 3 FSM do controlador FCtrl C_1

```
Estado 0: k := 5; channels_C1 := 6;
          I_s := 32 - k + 1;
          reset ram_addr;
          if data_ready = 1 then goto Estado 1 else goto Estado 0;
Estado 1: RAM(ram_addr) := pixel_in;
          write := 1; ram_addr := ram_addr + 1;
          if ram_addr = 1023 then goto Estado 2 else goto Estado 1;
Estado 2: reset ram_addr, X, Y, x, y;
Estado 3: ram_addr := (y + Y) * 32 + x + X;
          b := RAM(ram_addr); read := 1;
Estado 4: for d = 0 to channels_C1 - 1:
            kernel(d) := ROM_K(d, X, Y);
            MAC(d, b, kernel(d));
            mac_start := 1;
Estado 5: ram_addr := ram_addr + 1; X := X + 1;
          next_state := Estado 3;
          if X = k then X := 0; Y := Y + 1;
          if Y = k then Y := 0; x := x + 1; next_state := Estado 6;
          if x = I_s then X := 0; y := y + 1;
          if y = I_s then next_state := Estado 7;
          goto next_state;
Estado 6: data_out := MAC; data_out_ready := 1;
          mac_rst := 1; goto Estado 3;
Estado 7: data_out := MAC; data_out_ready := 1;
```

das camadas C_3 e C_5 é apresentada na Figura 4.6. O número de componentes de memória, denotado por d_e, é igual ao tamanho do barramento de entrada, e o número de componentes MAC é igual ao tamanho do barramento de saída, denotado por d_S. A Figura 4.6 também exibe um conjunto de barramentos de dados que conectam todas as d_e RAMs aos d_s componentes MAC. Para a camada C_3 temos $d_e = 6$ e $d_s = 16$, enquanto para a camada C_5 temos $d_e = 16$ e $d_s = 120$. Portanto, em C_3, existem seis memórias recebendo dados dos canais P_2, e dezesseis MACs realizando operações referentes ao cálculo do tensor de saída com dimensão de profundidade igual a dezesseis.

A Figura 4.7 apresenta a microarquitetura das camadas de *pooling* P_2 e P_4. Nessas camadas, como os tamanhos dos barramentos de entrada e saída são iguais (indicados por d), cada componente de memória é conectado diretamente a um componente MAX. Para a camada P_2, temos $d_e = d_s = 6$, enquanto para P_4, $d_e = d_s = 16$.

O número de canais em cada camada pode ser modificado alterando o número de componentes RAM e MAC (Figura 4.6) ou MAX (Figura 4.7). Isto poderia ser implementado usando componentes parametrizados para definir o número desses pares de componentes e, assim, permitir a escalabilidade do projeto. Camadas adicionais também poderiam ser implementadas da mesma forma, parametrizando o tamanho da entrada e saída de cada componente FCtrl. Aumentar o tamanho da imagem de entrada implica aumentar o tamanho de cada bloco de memória e consequentemente o tempo de processamento da camada. De forma prática, parametrizar os valores

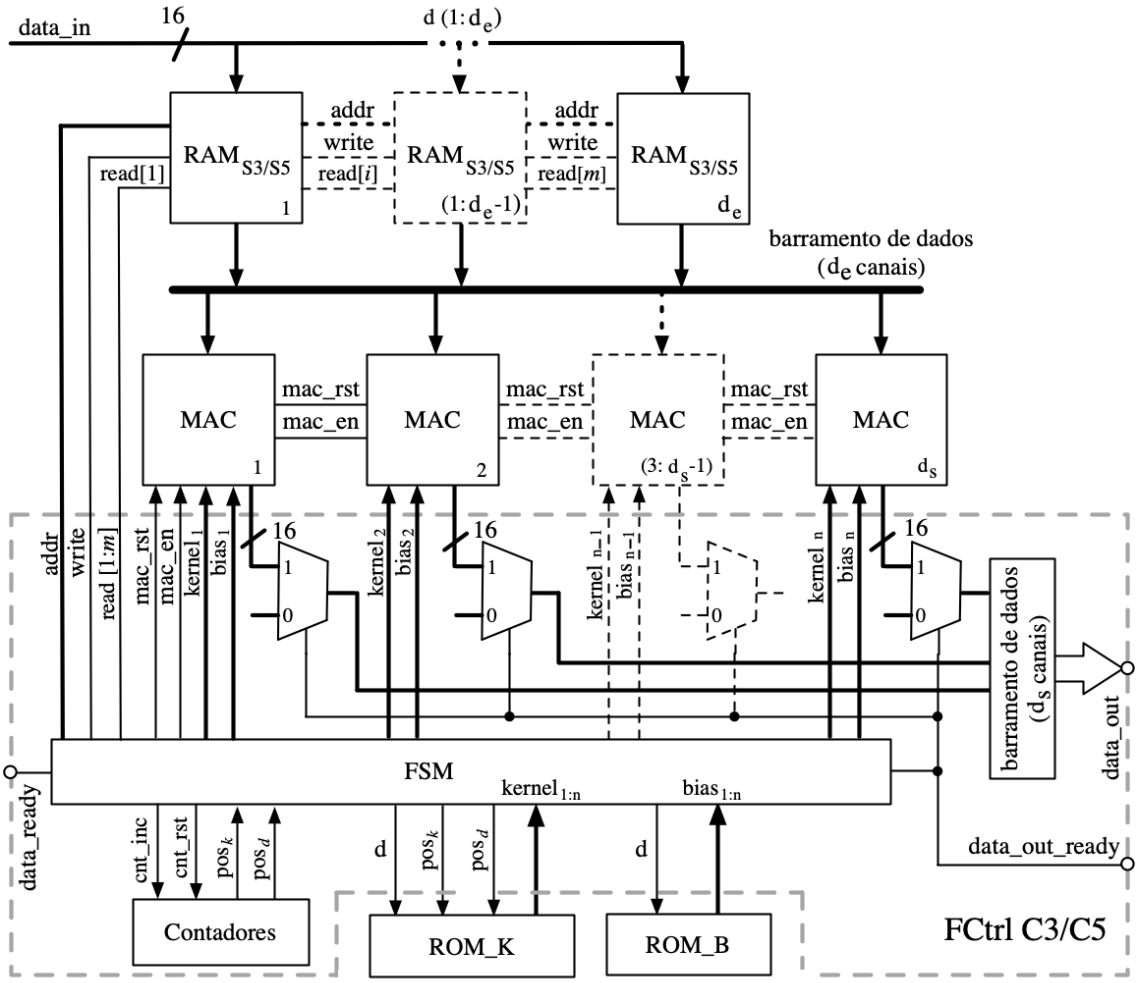


Figura 4.6: Microarquiteturas das camadas C3 (ou C5) da MFA.

de d_e, d_s e d apresentados nas Figuras 4.5, 4.6 e 4.7 permitiria a escalabilidade de cada camada desta arquitetura.

A proposta para esta arquitetura é a realização do processamento das camadas de forma sequencial, ou seja, todos os dados de uma camada são computados antes do fluxo de execução passar para a camada seguinte. Desta forma, uma parcela dos dados da RAM_{Si} é carregada, processada e transferida para ser armazenada na RAM_{Si+1}. Este processo se repete até que a RAM_{Si} seja integralmente processada. Ao final, o índice i é incrementado, indicando o início do processamento da camada seguinte. O processamento sequencial foi empregado para obter os resultados da implementação da CNN sem otimizações de recursos lógicos, desempenho e consumo de energia. Esse método foi empregado para estabelecer uma arquitetura de referência.

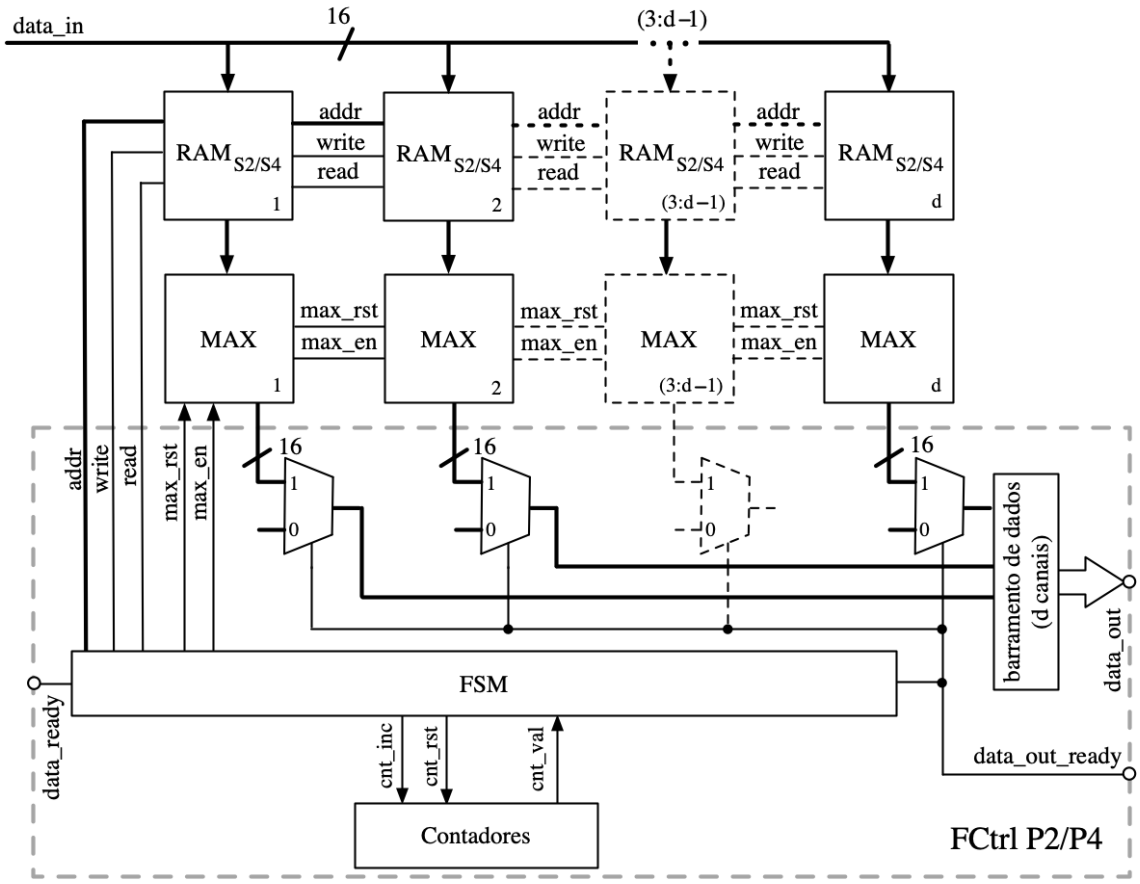


Figura 4.7: Microarquitetura das camadas P2 (or P4) da MFA.

4.3 Resultados

A MFA, e as demais arquiteturas propostas nesta tese, implementa a CNN LeNet-5 [25], e utiliza pesos pré-treinados com a base de dados MNIST [30] para classificar dígitos. A MNIST é uma base de dados de dígitos escritos manualmente, contendo dígitos de 0 a 9. A base de dados MNIST contém 60.000 imagens para treino e 10.000 imagens de teste. A saída da HNN, representada pelo conjunto de saídas da camada convolucional mais profunda, compreende 120 features (características) representadas em formato de ponto fixo de 16 bits. A avaliação dos resultados de precisão é apresentada no Capítulo 7.

A arquitetura de hardware é implementada, simulada e sintetizada em VHDL utilizando um dispositivo Altera Cyclone II 2C70. Na simulação do projeto da MFA destaca-se o desempenho para extrair as características da imagem de entrada com o tempo de 1,375 ms, correspondendo a aproximadamente 727 imagens por segundo com consumo de potência dinâmica[1] de 326 mW. A potência dinâmica é estimada pelo software de síntese e o tempo de processamento é função do número de ciclos de

[1]Consumo dinâmico está relacionado à atividade ativa do FPGA, enquanto o consumo estático está relacionado à energia consumida quando o FPGA está em estado ocioso ou em modo de espera.

clock e da frequência máxima. A frequência pode ser melhorada com a investigação e a remoção de algumas restrições, de caminhos críticos, e com otimização, para diminuir os atrasos de propagação dos sinais entre as portas lógicas.

Todas as constantes (pesos e *biases*) são implementadas no próprio dispositivo FPGA, resultando no uso, para o projeto completo, de 43.876 elementos lógicos. Da mesma forma, o armazenamento dos mapas de características e resultados intermediários consumiu cerca de 214 kb de memória RAM disponível no dispositivo.

A MFA foi implementada também em um FPGA moderno dispondo de mais recursos para verificar as melhorias alcançadas pelo projeto. Como a implementação é realizada em RTL, a migração pode ser realizada de um dispositivo de entrada, Altera Cyclone II 2C70, para um dispositivo Zynq UltraScale+ RFSOC (Xilinx XCZU28DR) sem a necessidade de alteração do projeto. Este dispositivo utiliza tecnologia de fabricação de 16 nm e embarca os principais recursos: 930.300 células lógicas, 4.272 DSPs (*digital signal processor*) e 38 Mb de memória BRAM (*block* RAM). A Tabela 4.1 apresenta os principais dados extraídos da síntese e simulação da MFA para os dois dispositivos FPGA utilizados, onde o tempo indicado representa o período necessário para a extração das características de uma imagem e o consumo representa o consumo dinâmico do FPGA. A Tabela 4.2 apresenta o resumo do resultados da síntese das camadas da MFA. A diferença entre os recursos utilizados nas camadas se deve ao número de canais existentes em cada uma delas. De acordo com os recursos mostrados na Tabela 4.2, seria possível estimar a área ocupada pelos principais componentes e dimensionar o projeto para uma nova topologia de HNN.

Tabela 4.1: Recursos utilizados nas implementações em VHDL.

Métrica	Cyclone II 2C70	Zynq XCZU28DR
Elementos lógicos/LUT	43.876	9.399
Memória	$214.344^{(*)}$	$22^{(**)}$
DSP	278	142
Registradores	6.198	7.590
Frequência (MHz)	47,15	88,07
Ciclo de clock	64.650	64.650
Tempo (ms)	1,375	0,734
Consumo (W)	0,326	0,143

$^{(*)}$ memória em bits; $^{(**)}$ memória em BRAM.

Tabela 4.2: Recursos usados pela MFA em síntese para o dispositivo Zynq UltraScale+ XCZU28DR.

	MFA				
Métrica	C_1	C_3	C_5	P_2	P_4
LUT	198	632	7.167	329	504
Registradores	338	664	4.019	237	395
DSP	6	16	120	0	0
BRAM	0,5	3	8	3	8

Um caminho crítico em um projeto RTL é a sequência mais longa de portas lógicas, incluindo suas conexões, que determina a frequência máxima de operação de um circuito. Identificar e otimizar os caminhos críticos é essencial para atender os requisitos de desempenho. Desta forma, é importante identificar o caminho de atraso mais longo pois ele é o responsável por determinar o período de clock mínimo que o circuito pode operar sem violar as restrições de *setup* e *hold* dos flip-flops.

O tempo de atraso, conhecido como *slack time*, é a diferença entre o período de clock em um dado ponto do circuito e a soma dos atrasos de propagação e dos tempos de chegada de dados até este ponto. Um *slack time* positivo significa que o caminho atende às restrições de tempo, enquanto um folga (*slack time*) negativa significa que o caminho viola as restrições de tempo. Normalmente o software de projeto de circuitos e de síntese permitem a geração de relatórios, facilitando a verificação dos valores de folga e dos elementos lógicos presentes em cada caminho.

A otimização do caminho crítico requer uma compreensão da origem do atraso e das restrições do projeto. Os métodos comuns incluem a redução da quantidade de lógica, incluindo operações aritméticas, simplificando ou reestruturando estas expressões, a minimização do *fanout*[2], ou a melhoraria da distribuição do sinal de clock pelo circuito. Isso pode ser obtido por meio de técnicas como pipeline, divisão de *fanout*, duplicação de lógica, multiplexação de clock, ou, ainda, por meio do aprimoramento dos algoritmos ou do roteamento.

A síntese da MFA, utilizando período de clock de 20 ns (frequência de 50 MHz) como restrição de tempo, resulta no valor de WNS (*worst negative slack*) de -1,208 ns, indicando que as restrições não são atendidas. Desta forma, a frequência de operação pretendida de 50 MHz não pode ser alcançada. A frequência máxima, utilizando os valores do período de clock e do WNS, pode ser calculada utilizando a Equação 4.1.

$$F_{max} = \frac{1}{periodo - WNS}. \tag{4.1}$$

A Tabela 4.3 apresenta o valor de *worst negative slack* obtido com a análise de tempo para cada um dos dispositivos FPGA, Cyclone e Zynq, e a frequência máxima de operação do projeto. As linhas "De" e "Para" indicam o caminho crítico. No FPGA Cyclone II, o caminho crítico encontra-se na transferência de dados entre a memória RAM e o MAC da camada C_5, enquanto no dispositivo Zynq, o caminho crítico está na transferência entre as camadas C_3 e P_4. Essas restrições resultam da realização de operação aritmética no mesmo ciclo de clock em que ocorre a leitura ou escrita na memória RAM.

[2] *Fanout* se refere ao número de saídas que uma porta lógica pode fornecer sinal, ou seja, é a capacidade de uma porta lógica de transmitir o seu valor de saída para múltiplas portas ou componentes sem degradação significativa do sinal.

Tabela 4.3: Caminhos críticos nas implementações da MFA.

Métrica	Cyclone II 2C70	Zynq XCZU28DR
De	C5:MEM	C3:MAC
Para	C5:MAC	P4:MEM
WNS (ns)	-1,208	8,645
F_{max} (MHz)	47,15	88,07

4.4 Considerações Finais

A síntese para o dispositivo Altera Cyclone II 2C70 utilizou 43.876 elementos lógicos, aproximadamente 62% dos blocos disponíveis no FPGA, enquanto para o dispositivo da Xilinx foram utilizados 9.399 blocos lógicos. Mesmo desconsiderando a diferença entre as tecnologias de fabricação dos dois FPGAs, Cyclone II de 90 nm e Zynq de 16 nm, o dispositivo da Xilinx consegue gerenciar melhor a implementação e roteamento do grande número de constantes utilizadas para os pesos. Para as demais métricas, como frequência máxima e consumo, já era previsível a grande diferença entre os dispositivos.

A MFA apresenta a desvantagem de processar as camadas sequencialmente. Na prática, a computação de uma camada inicia após a sua memória interna estar totalmente preenchida. Desta forma, o tempo de latência para o processamento de uma camada ser iniciado é bastante longo. Uma forma de reduzir essa latência está na eliminação da memória, permitindo que a computação seja iniciada tão logo o dado seja transferido para a camada permitindo, assim, a operação em pipeline. Esta premissa foi utilizada no projeto de uma nova arquitetura, apresentada no Capítulo 5, juntamente com a proposta de melhorar o paralelismo entre as camadas.

Capítulo 5

Arquitetura sem Memória

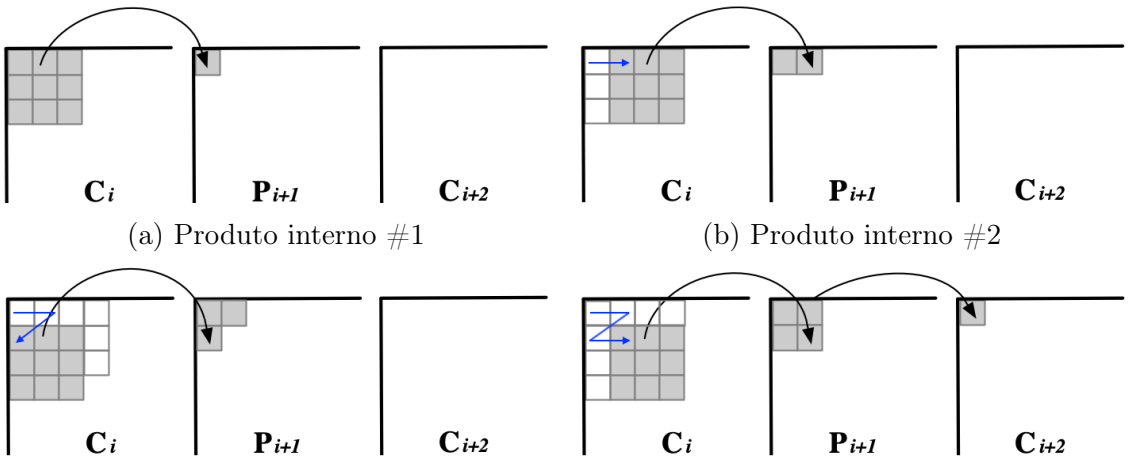
A avaliação dos resultados obtidos com a implementação da arquitetura com memória (MFA) permite identificar quais modificações podem ser importantes para adaptar o projeto da CNN, a fim de aprimorar o desempenho da implementação em um FPGA de entrada ou otimizar o uso dos recursos de hardware. Como mencionado no capítulo anterior, a precisão dos resultados é detalhada no Capítulo 7. No Capítulo 7, é mostrado que a representação numérica e os truncamentos empregados são apropriados para garantir alta precisão, ao mesmo tempo em que contribuem para economia de recursos lógicos, redução do consumo de energia e aproveitamento eficaz dos recursos disponíveis no dispositivo FPGA utilizado.

Um recurso sujeito a otimização é o emprego da memória RAM. Nesta arquitetura sem memória, a proposta consiste em eliminar por completo o uso desse recurso. Para isso, será apresentada uma forma modificada do padrão de leitura da imagem de entrada. Essa forma modificada de leitura viabiliza a execução das operações aritméticas na sequência necessária para assegurar o fluxo contínuo de dados para as camadas subsequentes, sem a necessidade de alocar memória para armazenar os mapas de características.

Esta arquitetura é formalmente chamada de *Memoryless Architecture* ou, abreviando, MLA. É uma denominação apropriada para uma arquitetura que dispensa totalmente o uso de memória RAM.

5.1 Arquitetura

A MLA é projetada para dispensar o uso de memória, antecipando as operações para as próximas camadas da CNN. Para permitir esta antecipação e o fluxo de dados para as demais camadas, é utilizado um padrão de varredura específico (uma sequência para os endereços dos dados) para ler e processar os dados de entrada e, assim, evitar completamente o armazenamento dos mapas de características. Consequentemente,



(a) Produto interno #1 (b) Produto interno #2

(c) Produto interno #3 (d) Produto interno #4

Figura 5.1: Produtos internos necessários para gerar a entrada da próxima camada. O exemplo desta figura utiliza kernel de tamanho $3 \times 3 \times 1$.

os respectivos blocos de memória são removidos do projeto, reduzindo o requisito de hardware para a implementação da rede.

O padrão utilizado para ler e processar os dados de entrada na MLA é ilustrado na Figura 5.1, que é válida para os índices $i = 1$ ou $i = 3$ (desta forma, é possível representar as camadas C_1, P_2 e C_3, ou C_3, P_4 e C_5). O kernel da camada C_i é representado em cinza. Para evitar o uso de memória, os dados devem ser processados em uma determinada ordem, de modo que as camadas seguintes tenham os dados de entrada necessários para calcular as próximas operações no processamento da CNN.

As quatro operações apresentadas na Figura 5.1, ou seja, o cálculo de quatro produtos internos na camada C_i, são responsáveis por gerar os quatro números para a camada P_{i+1}. Ao realizar uma operação de *max pooling* na Figura 5.1(d), obtém-se o primeiro número da camada C_{i+2}. Esta forma de deslocar o kernel para o cálculo dos produtos internos em C_i é a inspiração para a MLA. As sucessivas direções ao longo das quais o kernel é deslocado definem um padrão semelhante à letra "Z". Assim, esse padrão em Z compreende h operações adjacentes de produto interno, onde h é o tamanho da janela na camada P_{i+1}. Neste trabalho, usamos $h = 4$.

Para definir o padrão de leitura e processamento dos dados de entrada são necessários cinco pares de contadores, ou seja, um par (x_i, y_i) para cada camada i da rede. Observe que nas camadas convolucionais onde os kernels possuem tamanho 5×5, seus respectivos pares de contadores contam de 0 até 4. Da mesma forma, como as operações de *max pooling* são executadas com 4 números (formato 2×2), os contadores referentes às camadas de *pooling* variam de 0 a 1. Aninhando os contadores x e y de cada camada, o padrão é especificado para gerar o fluxo de entrada para a MLA. Um pseudo-código para gerar a ordem customizada deste padrão é apresentado no Algoritmo 4.

Algorithm 4 Sequência customizada para os pixels de entrada para gerar os padrões em Z necessários para garantir o fluxo para todas as camadas da MLA.

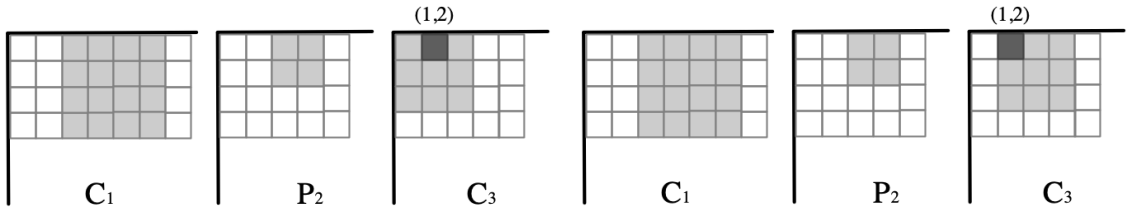
```
x0 := 0;
for y5 := 0 to 4: for x5 :=0 to 4:
  for y4 := 0 to 1: for x4 := 0 to 1:
    for y3 := 0 to 4: for x3 := 0 to 4:
      for y2 :=0 to 1: for x2 := 0 to 1:
        for y1 := 0 to 4: for x1 := 0 to 4:
          β:= x1 + 32*y1 + x2 + 32*y2 +
              2*(x3 + 32*y3 + x4 + 32*y4 +
              2*(x5 + 32*y5));
          buffer(x0) := input_image(β);
          x0 := x0 + 1;
return(buffer);
```

O vetor *input_image* contém a imagem de 32×32 pixels em formato linear de 1024 bytes. Ao final, o *buffer*, que não é armazenado no FPGA, terá a ordem customizada exigida pelo padrão em Z com tamanho de 250 kbytes. Este tamanho é resultado dos dez *loopings* presentes no Algoritmo 4 e utilizados para reordenar a imagem de entrada. Ou seja, $5 \times 5 \times 2 \times 2 \times 5 \times 5 \times 2 \times 2 \times 5 \times 5 = 250.000$ bytes. Esta "nova imagem" (vetor *buffer*) é transferida para o FPGA, byte a byte, pelo software de controle do sistema de processamento apresentado na Seção 2.3. O índice β para endereçar os pixels da imagem é calculado da seguinte forma:

$$\beta = x_1 + 32y_1 + x_2 + 32y_2 + 2(x_3 + 32y_3 + x_4 + 32y_4 + 2(x_5 + 32y_5)), \tag{5.1}$$

onde x_i é a coluna e y_i é a linha da imagem de entrada, e i indica o nível do *loop* aninhado.

No vetor *buffer* gerado pelo Algoritmo 4, os bytes da imagem são repetidos para que a MLA possa realizar seus cálculos sem uso de memória. Considerando que apenas o último resultado é armazenado em cada canal, é necessário uma grande quantidade de repetições de operação nas camadas C_1 e P_2. A Figura 5.2 exemplifica as operações executadas nessas duas camadas para calcular o ponto (1,2) na Camada C_3. Os pontos em destaque na Camada C_1 representam um padrão em Z (quatro produtos internos) necessários para calcular os quatro números (pontos) destacados na Camada P_2. Estes, por sua vez, são necessários para executar as operações de *maxpooling* e gerar o resultado de (1,2) na Camada C_3. O valor indicado pelo ponto (1,2) é usado para calcular dois produtos internos na Camada C_3, sendo o primeiro indicado na Figura 5.2a e o segundo na Figura 5.2b. O MAC das camadas convolucionais armazena apenas o último resultado, ou seja, o resultado intermediário de um produto interno. Assim, quando o segundo produto interno (Figura 5.2b) da Camada C_3 for calculado, o valor de (1,2) será recalculado. Para essa computação, todas as operações destacadas nas Camadas C_1 e P_2 serão repetidas.



(a) Produto interno #1 na Camada C_3 (b) Produto interno #2 na Camada C_3

Figura 5.2: Operações nas Camadas C_1 e P_2 para calcular o valor de (1,2) na Camada C_3.

Observa-se, portanto, que muitas operações na camada C_1 são utilizadas em mais de um produto interno e *max pooling* nas camadas seguintes. Calculando todas as repetições para uma imagem de 32×32 pixels e kernel de 5×5, obtemos as contagens apresentadas na Figura 5.3, agrupadas em padrões em Z.

A matriz da Figura 5.3 possui tamanho 14×14 porque um padrão em Z corresponde a quatro produtos internos e, devido ao kernel 5×5, as últimas quatro colunas e linhas da imagem de entrada não são usadas. Desta forma, a imagem 32×32 tem suas entradas (pixels) endereçadas por um mapa 14×14 de ocorrências de padrões em Z.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	1/1	2/2	3/3	4/4	5/5	26/5	37/5	42/5	49/5	54/5	61/4	66/3	73/2	78/1
2	6/2	7/4	8/6	9/8	10/10	27/10	38/10	43/10	50/10	55/10	62/8	67/6	74/4	79/2
3	11/3	12/6	13/9	14/12	15/15	28/15	39/15	44/15	51/15	56/15	63/12	68/9	75/6	80/3
4	16/4	17/8	18/12	19/16	20/20	29/20	40/20	45/20	52/20	57/20	64/16	69/12	76/8	81/4
5	21/5	22/10	23/15	24/20	25/25	30/25	41/25	46/25	53/25	58/25	65/20	70/15	77/10	82/5
6	31/5	32/10	33/15	34/20	35/25	36/25	47/25	48/25	59/25	60/25	71/20	72/15	83/10	84/5
7	85/5	86/10	87/15	88/20	89/25	90/25	97/25	98/25	101/25	102/25	105/20	106/15	109/10	110/5
8	91/5	92/10	93/15	94/20	95/25	96/25	99/25	100/25	103/25	104/25	107/20	108/15	111/10	112/5
9	113/5	114/10	115/15	116/20	117/25	118/25	125/25	126/25	129/25	130/25	133/20	134/15	137/10	138/5
10	119/5	120/10	121/15	122/20	123/25	124/25	127/25	128/25	131/25	132/25	135/20	136/15	139/10	140/5
11	141/4	142/8	143/12	144/16	145/20	146/20	153/20	154/20	157/20	158/20	161/16	162/12	165/8	166/4
12	147/3	148/6	149/9	150/12	151/15	152/15	155/15	156/15	159/15	160/15	163/12	164/9	167/6	168/3
13	169/2	170/4	171/6	172/8	173/10	174/10	181/10	182/10	185/10	186/10	189/8	190/6	193/4	194/2
14	175/1	176/2	177/3	178/4	179/5	180/5	183/5	184/5	187/5	188/5	191/4	192/3	195/2	196/1

Figura 5.3: Matriz com 196 células no formato A/B, onde A é o índice da sequência e B é o número de ocorrências de padrões em Z na camada C_1.

Os números em cada célula da Figura 5.3 representam o número sequencial de cada padrão em Z, não considerando suas repetições, e o número de ocorrências. Por exemplo, o par na posição (2,2) indica que este é o sétimo padrão em Z executado

pela CNN. Ele é computado quatro vezes ao longo da CNN (das quais, três são repetições). O número de ocorrências $B_{i,j}$ exibido na matriz da Figura 5.3 é calculado usando as Equações (5.2) e (5.3):

$$B_{i,1} = \begin{cases} i, & \text{if } i < k \\ k, & \text{if } k \leq i \leq 2k-1 \\ 3k-i, & \text{if } i > 2k-1, \end{cases} \tag{5.2}$$

$$B_{i,j} = \begin{cases} jB_{i,1}, & \text{if } 1 < j < k \\ kB_{i,1}, & \text{if } k \leq j \leq 2k \\ B_{i,(3k-j)}, & \text{if } j > 2k, \end{cases} \tag{5.3}$$

onde k é o tamanho do kernel. A matriz da Figura 5.3 é gerada usando $k = 5$.

O pseudo-código que representa a arquitetura MLA é detalhado no Algoritmo 5. Este algoritmo calcula um *array* em cada camada para ser processado na camada seguinte e, então, estes *arrays* são descartados. Os valores destas estruturas são sobrescritos na próxima iteração, evidenciando que os mapas de características não são armazenados. Observe que o tamanho de cada *array* é definido pelo número de canais na camada, e os *arrays* são implementados pelos mesmos registradores existentes e apresentados nas Figuras 4.1 e 4.2.

No início do algoritmo e no final do processamento de cada camada, os contadores são zerados. Cada canal das camadas convolucionais é inicializado com seu respectivo valor de bias (detalhado na Figura 4.1). Nas camadas de *pooling* esses valores são -32768, representando o menor valor negativo em 16 bits (Figura 4.2). Os valores de bias e kernels (K_1, K_3 e K_5) são lidos das memórias somente de leitura ROM_B e ROM_K, respectivamente. Cada camada possui um controlador e um número d de blocos MAC e MAX executados em paralelo, então a notação "parallel for" é usada no Algoritmo 5 para enfatizar o fato de que todos os canais de saída c são computados simultaneamente. Nas camadas convolucionais, o cálculo dos valores de C_1, C_3 e C_5 para cada canal é feito em paralelo. Porém, a combinação dos mapas de características, como depende de seu valor anterior, é realizado sequencialmente. O Algoritmo 6 apresenta estes mesmos cálculos realizados nas camadas convolucionais e detalha a operação do MAC na forma de uma FSM.

Nas camadas de *pooling* são executadas duas operações MAX. A primeira compara os valores atuais com os recebidos da camada anterior. Depois de receber quatro números, a segunda operação é executada comparando a saída com 0 para executar a função ReLU.

Todos os processos são executados simultaneamente. Enquanto os processos das

Algorithm 5 Computações em cada camada da MLA.

```
let x_i := 0, for i := 0...5;
let y_i := 0, for i := 1...5;
let C_i := Bias_i, for i := 1, 3, 5;
let P_i := [-32768 ... -32768], for i := 2, 4;
let cnt_i := 0, for i := 2, 4;
let done_i := false, for i := 1...5;

process Layer_1()
  parallel for d := 1 to 6:
    C_1(d) := C_1(d) + buffer(x_0) * K_1(y_1,x_1,d);
  x_0 := x_0 + 1; x_1 := x_1 + 1;
  if done_5 then halt;
  if x_1 = 5 then x_1 := 0; y_1 := y_1 + 1;
  if y_1 = 5 then y_1 := 0; done_1 := true;
end process

process Layer_2(done_1)
  if done_1 then
    parallel for d := 1 to 6:
      if C_1(d) > P_2(d) then P_2(d) := C_1(d);
    cnt_2 := cnt_2 + 1;
    if cnt_2 = 4 then
      parallel for d := 1 to 6:
        if P_2(d) < 0 then P_2(d) := 0;
      cnt_2 := 0; done_2 := true;
    C_1 := Bias_1; done_1 := false;
end process;

process Layer_3(done_2)
  if done_2 then
    parallel for d := 1 to 16:
      for t := 1 to 6:
        C_3(d) := C_3(d) + P_2(t) * K_3(y_3,x_3,t,d);
    x_3 := x_3 + 1;
    if x_3 = 5 then x_3 := 0; y_3 := y_3 + 1;
    if y_3 = 5 then y_3 := 0; done_3 := true;
    P_2 := [-32768 ... -32768]; done_2 := false;
end process;

process Layer_4(done_3)
  if done_3 then
    parallel for d := 1 to 16:
      if C_3(d) > P_4(d) then P_4(d) := C_3(d);
    cnt_4 := cnt4 + 1;
    if cnt_4 = 4 then
      parallel for d := 1 to 16:
        if P_4(d) < 0 then P_4(d) := 0;
      cnt_4 := 0; done_4 := true;
    C_3 := Bias_3; done_3 := false;
end process;

process Layer_5(done_4)
  if done_4 then
    parallel for d := 1 to 120:
      for t := 1 to 16:
        C_5(d) := C_5(d) + P_4(t) * K_5(y_5,x_5,t,d);
    x_5 := x_5 + 1;
    if x_5 = 5 then x_5 := 0; y_5 := y_5 + 1;
    if y_5 = 5 then done_5 := true;
    P_4 := [-32768 ... -32768]; done_4 := false;
end process;
```

camadas mais profundas são habilitados por um determinado sinal, a camada C_1 é executada continuamente dependendo apenas do vetor de entrada *buffer*, conforme apresentado no Algoritmo 4. Observe que esses processos formam um pipeline de 5 estágios.

Algorithm 6 FSM do MAC para o canal d da camada l

```
Estado 0: total_k := 25;
          reset cnt_d, cnt_k;
          sum := BIAS(l, d);
Estado 1: if data_in_ready = 1 then data := data_in; goto Estado 2;
          goto Estado 1;
Estado 2: multi_a := data(cnt_d);
          multi_b := ROM_K(d, cnt_d, cnt_k);
          sum := sum + multi_a * multi_b;
          cnt_d := cnt_d + 1;
          if cnt_d = d then
            cnt_k := cnt_k + 1; reset cnt_d;
            if cnt_k = total_k then goto Estado 3;
          goto Estado 1;
Estado 3: data_out := sum(17,2); data_out_ready := 1;
          goto Estado 0;
```

A arquitetura da MLA é apresentada na Figura 5.4, contendo um controlador por camada e blocos MAC e MAX em número correspondente à quantidade de canais. Diferentemente da MFA, a MLA não inclui nenhuma memória auxiliar, ou seja, nenhum bloco de RAM é usado neste projeto. O controlador LCtrl para cada camada da MLA determina a operação correta dos blocos MAC e MAX. Na MLA, esses controladores são mais simples do que os presentes na MFA.

A Figura 5.5 apresenta a microarquitetura das camadas convolucionais da MLA. Ao contrário da MFA, este projeto não apresenta diferença entre a camada C_1 e as camadas C_3 e C_5. Dispensando o uso de memória, os d_e canais de entrada são conectados diretamente aos d_s componentes MAC.

A escalabilidade desse projeto também é mais intuitiva do que a MFA. Para aumentar o número de canais, basta incluir componentes MAC de forma paralela e, para processar uma imagem de entrada maior, os contadores devem ser ajustados de forma a permitir que o padrão em Z seja aplicado a toda a imagem. Esses valores podem ser parametrizados na implementação dos componentes. Observe que a memória RAM_{S1}, usada para armazenar a imagem de entrada na MFA (Figura 4.4), não existe na MLA (Figura 5.4). A imagem de entrada é recebida sequencialmente via *pixel_in*, que alimenta diretamente os MACs da primeira camada. O resultado do MAC é o único valor armazenado na MLA. Este resultado é mantido no registrador *sum* interno do MAC. No Algoritmo 6, a variável cnt_k é controlada pela FSM e indica o número de operações que devem ser realizadas para completar um produto interno. Como o tamanho do kernel é $5\times5\times d$, onde d representa a profundidade do tensor, as operações de multiplicação e adição são executadas $25d$ vezes. A profun-

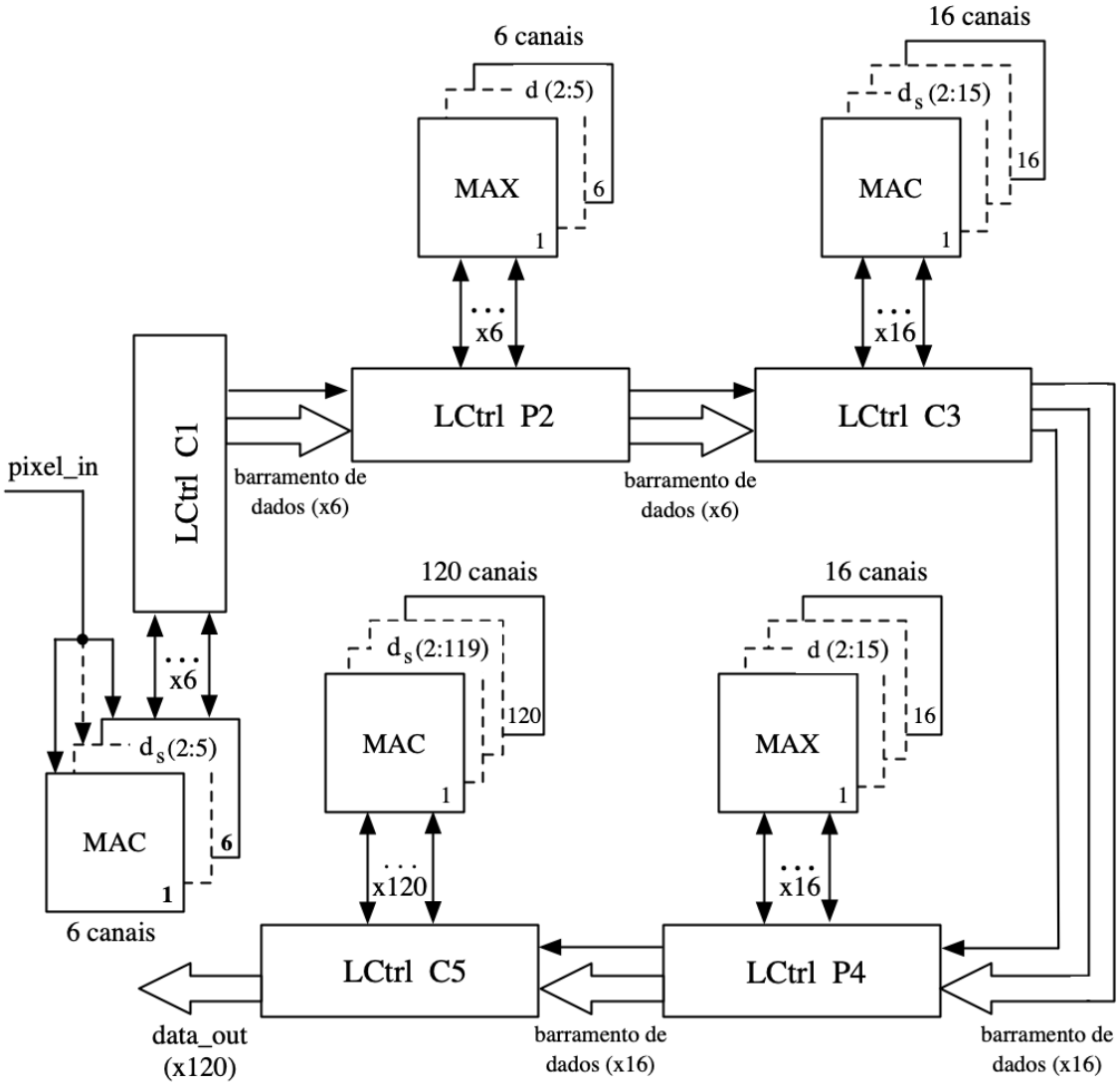


Figura 5.4: Diagrama de blocos da MLA.

didade do tensor depende da camada convolucional, com d tendo os valores 1, 6 e 16 nas camadas C_1, C_3 e C_5, respectivamente. Esses valores de d são parametrizados na instanciação do componente.

Nas camadas de *pooling*, os resultados das comparações são armazenados em um registrador interno. Este registrador é atualizado a cada novo resultado de produto interno recebido. Após a conclusão de uma janela de subamostragem, o valor resultante é usado pela função de ativação e propagado para a próxima camada da rede. A Figura 5.6 apresenta a microarquitetura das camadas de *pooling*. Neste projeto, o número de blocos MAX corresponde ao tamanho do barramento de entrada, mas sem a presença de memória RAM. Os d barramentos de entrada são conectados diretamente aos d componentes MAX.

Conforme o conceito da MLA apresentado na Figura 5.1, ao término de um determinado número de computações de uma camada, o processamento da camada

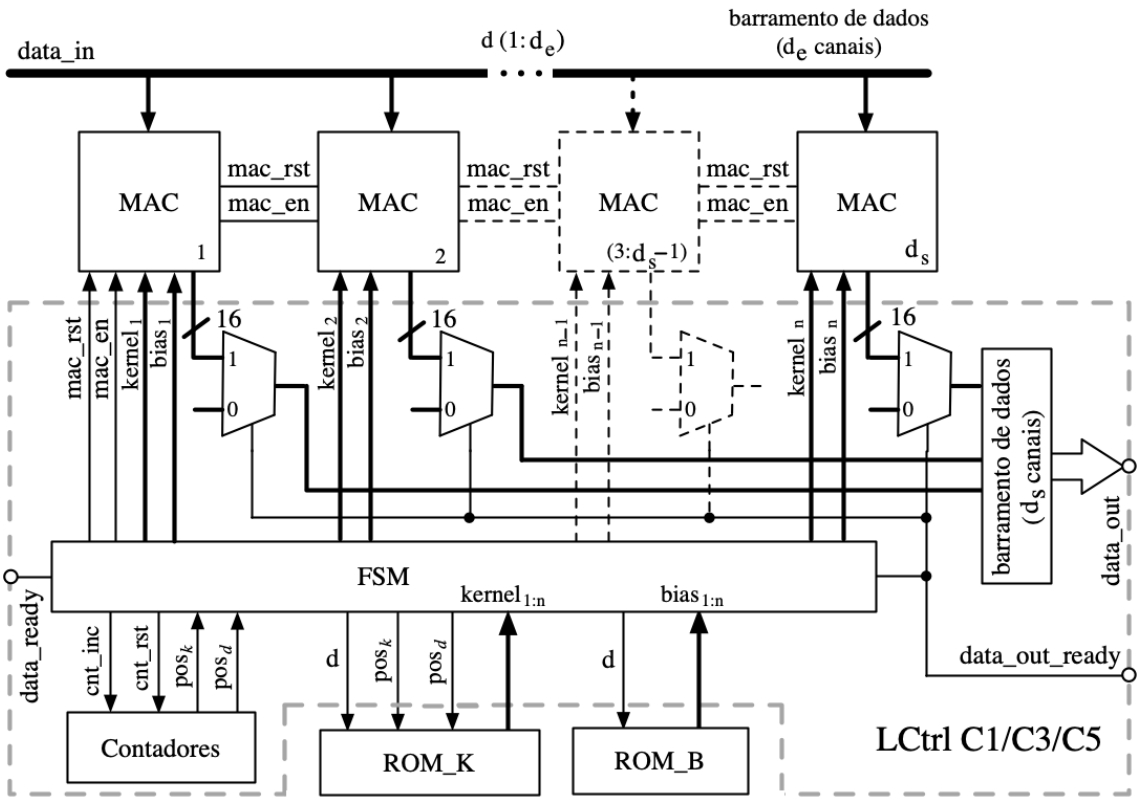


Figura 5.5: Microarquitetura das camadas convolucionais na MLA.

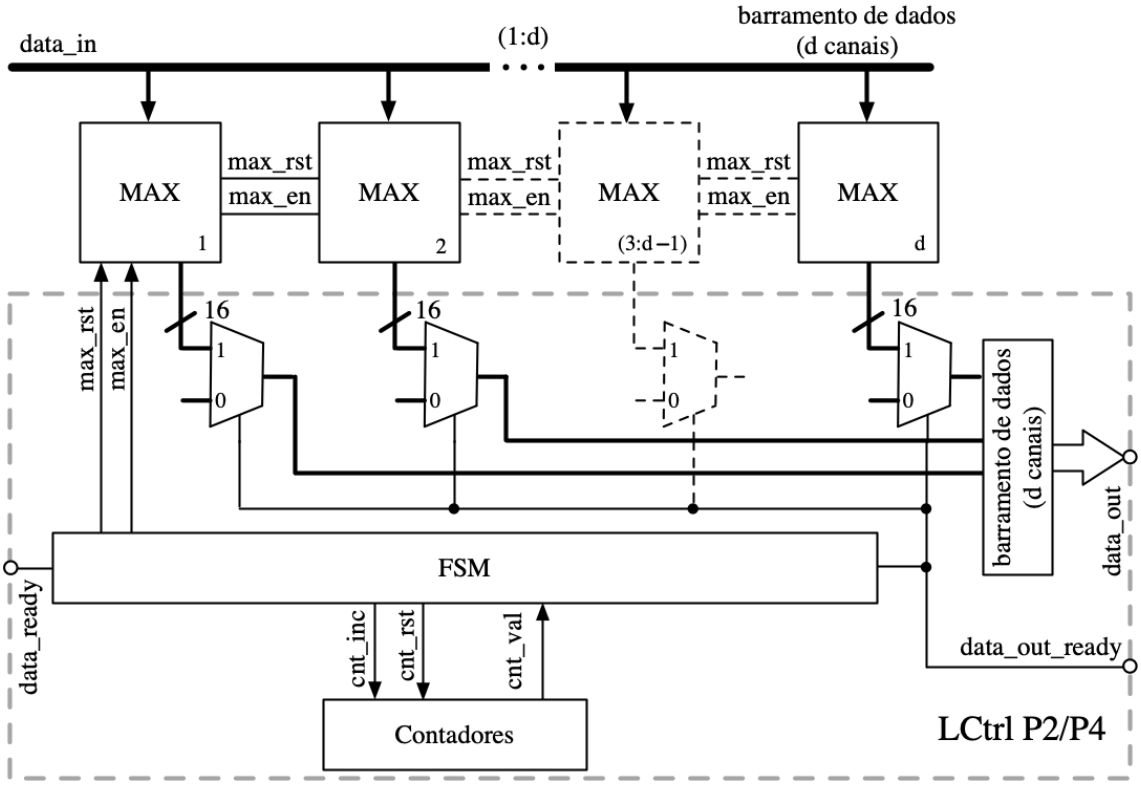


Figura 5.6: Microarquitetura das camadas de *pooling* na MLA.

seguinte pode ser iniciado. Desta forma, o fluxo de dados é propagado ao longo de toda a rede de forma quase que contínua. O fluxo não é contínuo pois existe diferença no tempo de execução das operações entre as camadas. De forma prática, o processamento da camada P_2 é iniciado após a realização de 4 produtos internos na camada C_1. As computações em C_3 se iniciam após 4 operações de *max pooling* em P_2 (considerando o tamanho da janela como 2×2), ou ainda, após 16 operações de produto interno na camada C_1; e assim por diante. Desta forma, quando a primeira operação for realizada na camada C_5, existirão outras operações sendo executadas simultaneamente nas demais camadas.

5.2 Arquitetura com Múltiplos MACs na Camada C_1

A proposta para a MLA é executar um determinado número de operações e, sem realizar armazenamento em memória, transferir os resultados para que a computação na próxima camada seja iniciada. No entanto, a execução deste fluxo requer recálculo de alguns produtos internos e operações de *max pooling*. Utilizando novamente o exemplo da Figura 5.1, para realizar os quatro produtos internos na camada C_1, é necessário o recebimento de 9 números para cada operação, ou seja, 36 números para que os quatro produtos internos sejam realizados. Uma forma de minimizar a repetição dos dados de entrada é permitir que todos os números necessários para estas operações sejam recebidos sequencialmente. Desta forma, apenas 16 números são necessários ao invés de 36.

Para alcançar esta otimização, a camada convolucional C_1 é composta por quatro componentes MAC, cada um dedicado a um produto interno. Este número está diretamente associado ao tamanho da janela utilizada na camada de *pooling* seguinte. A Figura 5.7 apresenta a microarquitetura da camada convolucional C_1. Esta arquitetura é formalmente chamada de *Multiple MACs per Layer Architecture* ou, abreviando, MMA.

Enquanto o padrão em Z na arquitetura MLA é composto por blocos de 9 pixels, considerando kernel de tamanho 3×3 como apresentado na Figura 5.1, na arquitetura com múltiplos MACs é utilizado o padrão com 16 pixels. Cada um dos MACs receberá apenas os dados referentes a um dos quatro produtos internos. Este controle é realizado utilizando um contador adicional. A Figura 5.8 apresenta a lógica de seleção para cada componente MAC.

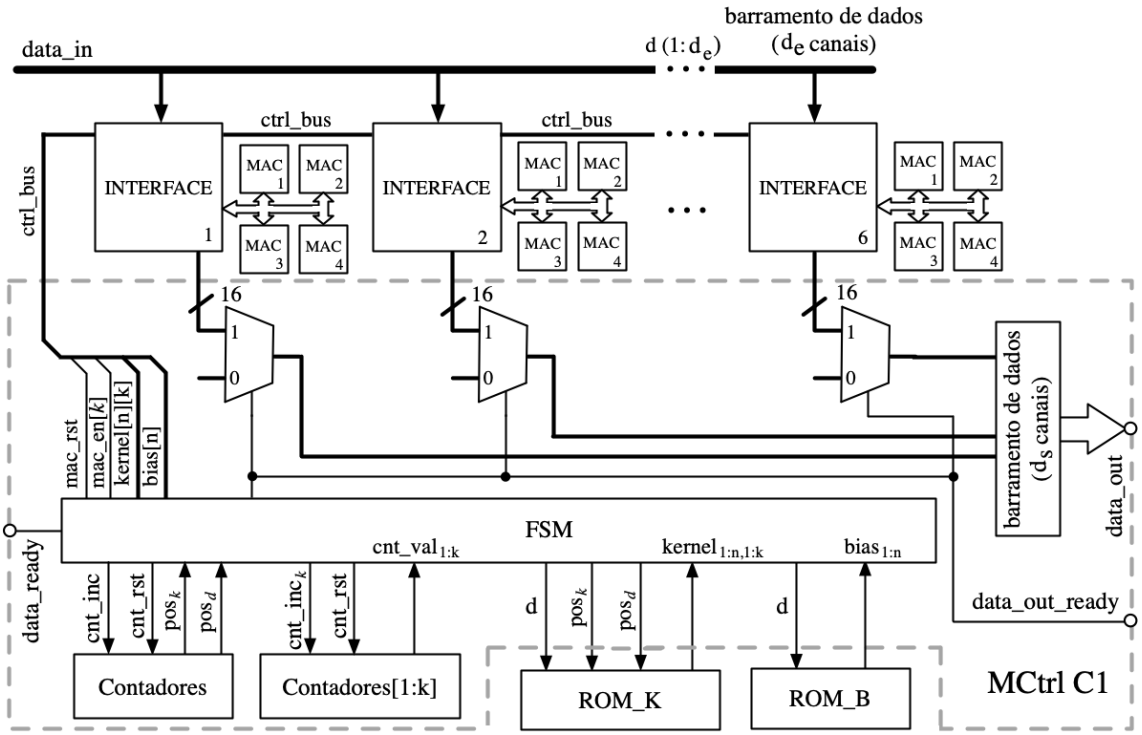


Figura 5.7: Microarquitetura da camada C1 com Múltiplos MACs.

5.3 Resultados

A Tabela 5.1 apresenta os principais dados extraídos da síntese e simulação da MLA e MMA para os FPGAs Cyclone II 2C70 e Zynq UltraScale+ XCZU28DR. Conforme esperado, não é utilizada nenhuma memória pois os dados intermediários são armazenados em registradores (apenas um número é armazenado em cada canal). Estes registradores são integrantes dos componentes MAC e MAX apresentados na Seção 4.1. Conforme descrito no Algoritmo 4, a imagem de entrada é reordenada para um vetor de 250.000 bytes que é transferido para a MLA byte a byte. Consequentemente, é esperado um grande número de ciclos de clock para extrair as características da imagem de entrada e, pelo resultado da simulação, a MLA requer 270.000 ciclos de clock. Com a frequência de operação alcançada no dispositivo Cyclone II de 55,48 MHz, a MLA é capaz de processar e extrair características em 4,866 ms, que corresponde a aproximadamente 205 imagens por segundo. A MMA, com uso similar de elementos lógicos e de registradores, aprimora o desempenho em 3×, extraindo as características em 90.000 ciclos de clock, que corresponde a aproximadamente 612 imagens por segundo, com frequência de operação de 55,15 MHz.

Ao utilizar um dispositivo moderno, da família Zynq UltraScale+, é evidenciado que as implementações das arquiteturas são bastante eficientes, com a MLA requerendo menos de 7% dos 4.272 DSPs disponíveis, por exemplo. Embora o total de ciclos de clock para extrair as features não é alterado, como a frequência má-

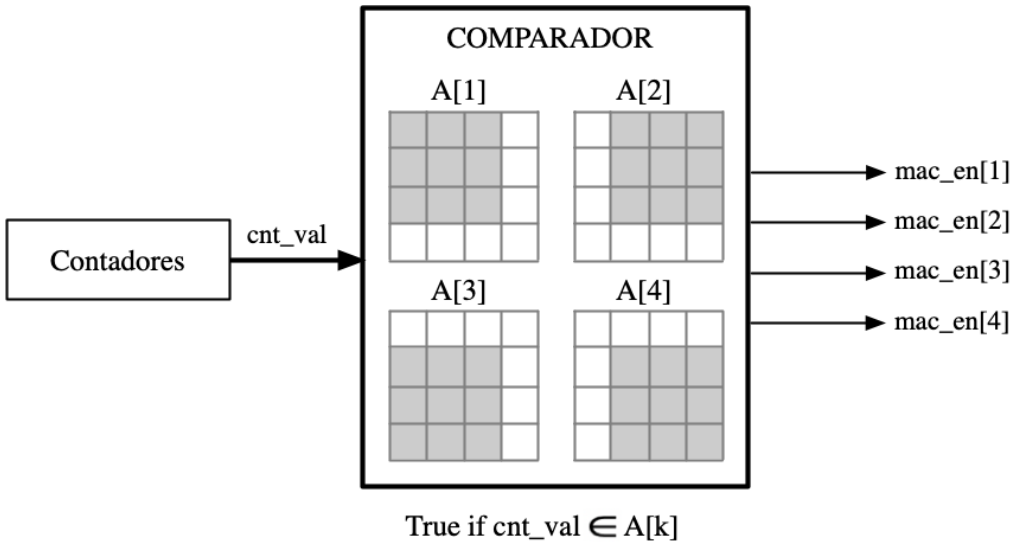


Figura 5.8: Controle dos produtos internos para definição do MAC.

Tabela 5.1: Recursos usados nas sínteses para os FPGAs Cyclone II 2C70 e Zynq UltraScale+ XCZU28DR.

	Cyclone II 2C70		Zynq UltraScale+	
Métrica	MLA	MMA	MLA	MMA
Elementos Lógicos/LUT	42.036	43.628	8.521	9.122
Memória (bits)	0	0	0	0
DSP	278	296	142	160
Registradores	5.898	6.348	7.459	8.125
Frequência (MHz)	55,48	55,15	97,48	117,72
Ciclos de clock	270.000	90.000	270.000	90.000
Tempo (ms)	4,866	1,632	2,770	0,765
Consumo (W)	0,303	0,323	0,076	0,089

xima de operação é superior, as arquiteturas MLA e MMA processam cerca de 361 e 1.307 imagens por segundo, respectivamente. A redução na potência dinâmica consumida é significativa em relação ao dispositivo de baixo custo.

A Tabela 5.2 apresenta o resumo para cada camada. Convém observar que o total de recursos utilizados nas camadas não, necessariamente, correspondem nas duas tabelas, isto deve-se ao fato das implementações contarem com outros circuitos de controle, além do componente de serialização dos dados de saída. Pode-se notar que o consumo de recursos do FPGA aumenta à medida que o índice da camada também aumenta. Isso ocorre devido ao aumento no número de canais à medida que se percorre as camadas na rede neural.

Tabela 5.2: Recursos usados, por camada, em sínteses para o dispositivo Zynq UltraScale+ XCZU28DR.

	MLA					MMA				
Métrica	C_1	C_3	C_5	P_2	P_4	C_1	C_3	C_5	P_2	P_4
LUT	94	608	7.088	56	141	673	610	7.050	111	141
Registradores	200	618	4.338	103	263	900	618	4.304	103	263
DSP	6	16	120	0	0	24	16	120	0	0
BRAM	0	0	0	0	0	0	0	0	0	0

Na síntese da MLA e da sua variação, a MMA, todas as restrições de tempo são atendidas de forma que os tempos de atraso são todos positivos. A principal restrição de tempo imposta foi o uso de 20 ns para o período de clock para que o projeto opere na frequência de 50 MHz. A Tabela 5.3 apresenta os caminhos críticos da MLA e MMA nos dois FPGAs utilizados. Como as modificações na MMA, com relação à MLA, concentram-se na camada C_1 e o caminho crítico é verificado na camada C_5 da MLA, é de se esperar que os tempos de folga sejam parecidos na MMA. O caminho crítico nas implementações, encontra-se na transferência e uso de um contador cujo valor é definido na FSM e utilizado no MAC da camada C_5. Nesta tabela também são apresentadas as frequências de operação calculadas pelos valores dos WNS na Equação 4.1.

Tabela 5.3: Caminhos críticos nas implementações da MLA.

	MLA		MMA	
Métrica	Cyclone II 2C70	Zynq XCZU28DR	Cyclone II 2C70	Zynq XCZU28DR
De	C5:FSM	C5:FSM	C5:FSM	C5:FSM
Para	C5:MAC	C5:MAC	C5:MAC	C5:MAC
WNS (ns)	1,977	9,741	1,869	11,505
F_{max} (MHz)	55,48	97,48	55,15	117,72

5.4 Considerações Finais

Conforme apresentado na descrição da MLA, a arquitetura requer que alguns produtos internos e operações de *max pooling* sejam recalculados. Consequentemente, existem computações redundantes nas camadas da rede neural. Estes cálculos redundantes resultam em um tempo de processamento mais longo da MLA em comparação com o obtido pela MFA. O desempenho alcançado pela MLA, utilizando o FPGA da família Cyclone II, permite o processamento de cerca de 205 imagens por segundo. Este resultado ainda é comparável com trabalhos encontrados na literatura, porém é cerca de 3,5x menor que o conseguido pela MFA. O consumo de energia é ligeiramente inferior para utilização de aproximadamente 4% menos elementos lógicos e sem o emprego de memória.

A eficiência de transferência de dados entre as camadas e a operação em pipeline são características bastante interessantes apresentadas pela MLA. Para aprimorar o tempo de processamento, a experimentação no uso de alguns elementos de memória se apresenta como uma boa alternativa de investigação, posicionando uma nova arquitetura entre a MFA e a MLA. Com esse objetivo em mente, é proposta no Capítulo 6, a arquitetura com memória cache.

Capítulo 6

Arquitetura com Memória Cache

A proposta da arquitetura com memória cache (CMA, de *cache-memory architecture*) é usar o mesmo padrão de leitura da MLA, enquanto melhora o desempenho do processamento ao armazenar, em uma memória cache, parte dos produtos internos previamente calculados. Assim como na MLA, na CMA a imagem de entrada é recebida externamente e não é armazenada. A memória cache é adicionada à camada C_1, por causa da grande quantidade de operações repetidas na MLA, conforme ilustrado na Figura 5.3. Outra memória cache pode ser incluída na camada P_2 para eliminar completamente qualquer recálculo, porém esta abordagem não será tratada neste trabalho, podendo ser um tópico para futuras investigações.

6.1 Componentes Básicos

A CMA faz uso dos mesmos blocos básicos MAC e MAX apresentados na MFA. Adicionalmente é empregado um componente para controle da memória cache. A Figura 6.1 apresenta a arquitetura utilizada para a memória cache. Cada *slot* da cache é composto por um endereço de 8 bits ($addr\_z$), um bit de validade ($valid$) e quatro palavras (16-bits $\times$ 4), para armazenar os quatro produtos internos computados, resultando, assim, em um total de 73 bits. Como existe uma memória cache em cada um dos seis canais da camada C_1, temos $73 \times 6 = 438$ bits para cada *slot* de memória. A memória cache usada na CMA é totalmente associativa [65, 66] (um padrão em Z pode ocupar qualquer um dos slots da memória cache), organizada em um único conjunto de cache com múltiplos *slots*.

Durante uma consulta à memória cache, conforme detalhado no Algoritmo 7, o valor do contador cnt_z é inicialmente comparado com o z para definir a linha do padrão em Z correspondente. Para gravar os resultados do produto interno correspondentes a um padrão em Z na memória cache, o bit de validade deve ser definido com o valor 1, e os registradores $word$[0..3] devem ser atualizados sequencialmente com os resultados dos produtos internos indicados pelo contador cnt_p. Para ler os

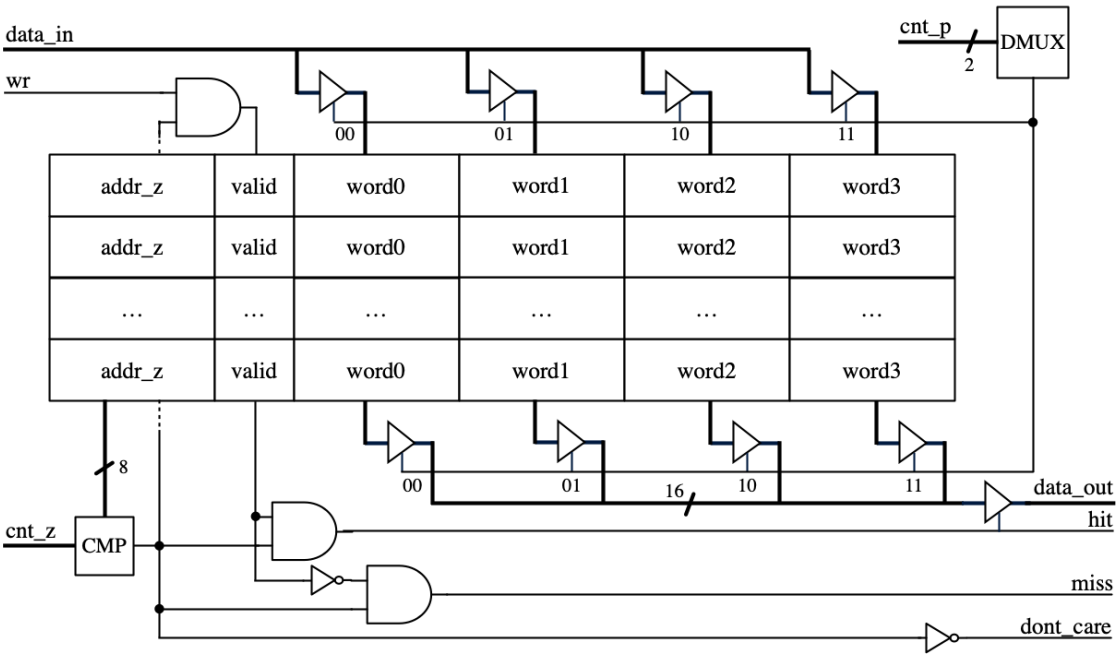


Figura 6.1: Diagrama para controle dos *slots* da cache.

produtos internos de um determinado padrão em Z da memória cache, o bit de validade é primeiro verificado para definir os sinais de *hit* e *miss*. Ao receber um sinal de *hit*, o controlador da cache carrega os produtos internos da memória. No caso da consulta resultar em *miss*, os produtos internos são calculados e seus resultados são armazenados.

Algorithm 7 Leitura/Escrita da cache para endereços cnt_z

```
if cache.addr_z(row) = cnt_z then
    if wr = 1 then
        cache.valid(row) = 1
        for cnt_p = 0 to 3 do
            cache.word(row, cnt_p) = data_in
        end for
    else
        if cache.valid(row) = 1 then
            cache.hit = 1
            for cnt_p = 0 to 3 do
                data_out = cache.word(row, cnt_p)
            end for
        else
            cache.miss = 1
        end if
    end if
end if
```

6.2 Arquitetura

O conceito desta arquitetura é empregar uma memória cache para armazenar os resultados de cálculos anteriores e utilizá-los, sempre que necessário, para reduzir

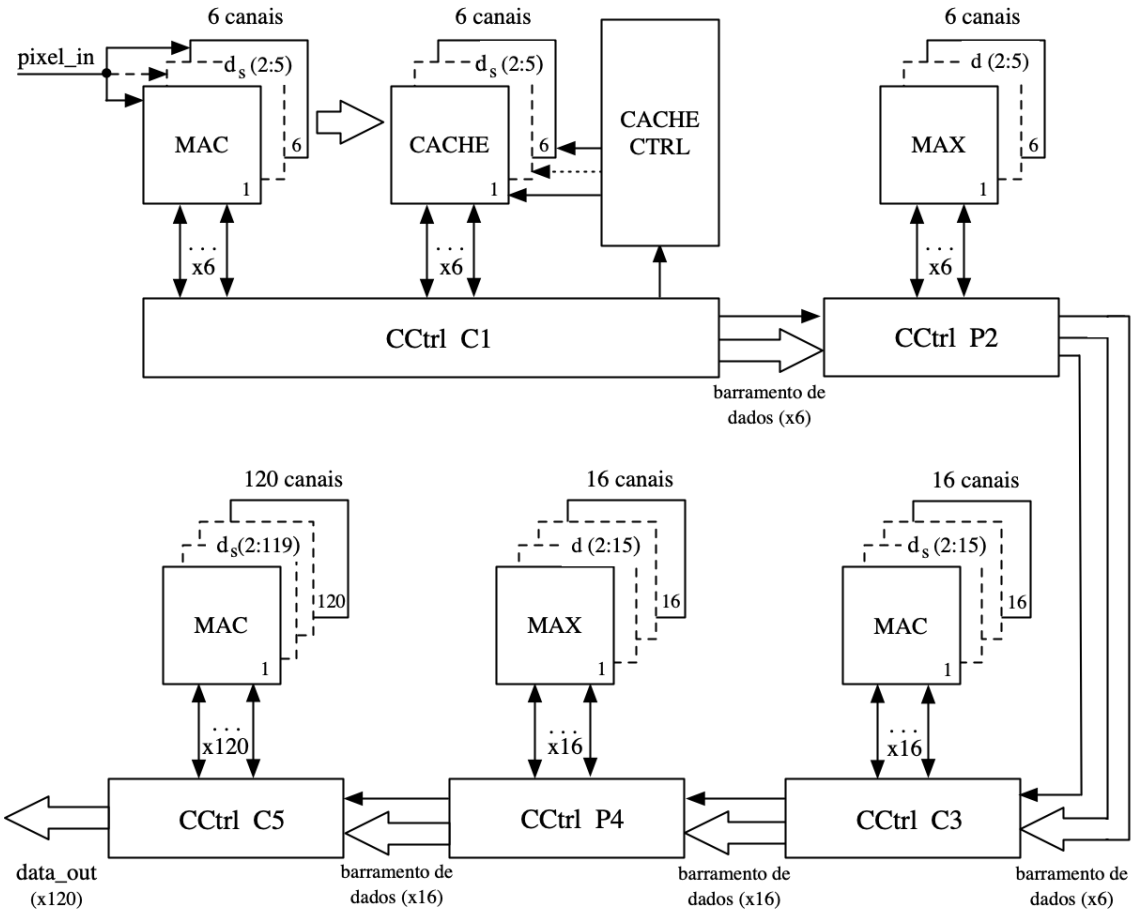


Figura 6.2: Diagrama de blocos da CMA.

o número de computações repetidas. Em um sistema com CPU, a memória é organizada hierarquicamente desde os registradores até a memória interna, passando pelas memórias cache L_1, L_2 e L_3. Por analogia, dada a proximidade do processador (a memória está diretamente conectada ao MAC, que representa a unidade de processamento em uma camada convolucional), tamanho, compartilhamento (utilizada apenas pela camada C_1) e a latência alcançada, a memória cache empregada na CMA é equivalente à memória cache L_1 [67–70]. Esta equivalência com um sistema que emprega uma CPU é a motivação para a denominação desta arquitetura como CMA.

O diagrama de blocos da CMA é apresentado na Figura 6.2. Comparando com a Figura 5.4, o diagrama da CMA inclui a memória CACHE e o controlador CACHE CTRL. Existem seis blocos de memória cache, um para cada canal, que armazenam os resultados dos MACs. Para gerar os sinais de escrita/leitura para a memória cache, o bloco CACHE CTRL utiliza uma tabela de consulta interna (LUT).

A Figura 6.3 apresenta a microarquitetura da camada convolucional C_1 na CMA. Os multiplexadores na parte superior atuam como um seletor de entrada de dados para as memórias cache. O sinal de escrita (wr) determina se o resultado, obtido pelos componentes MAC, deve ser armazenado na memória cache. Os MUXs na parte

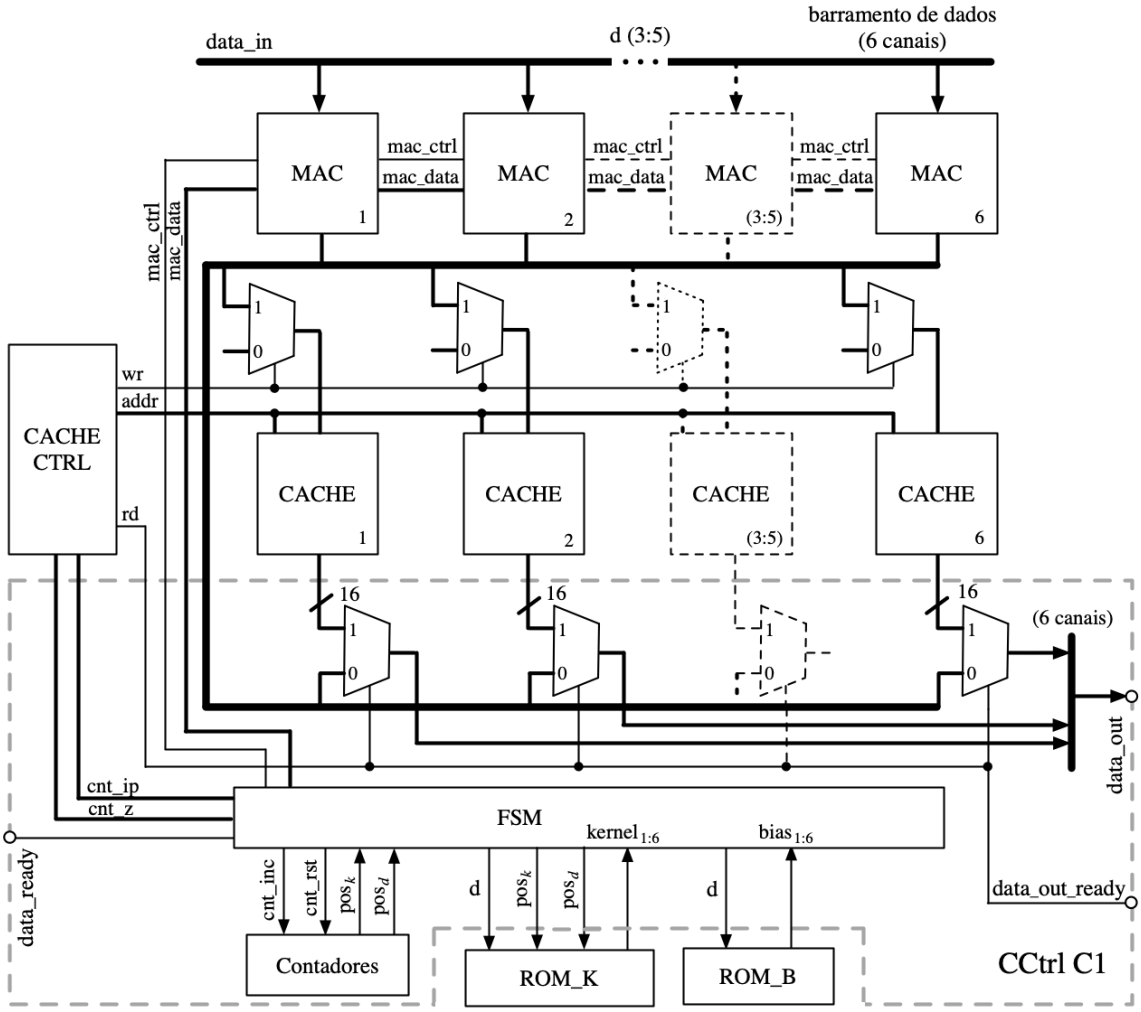


Figura 6.3: Microarquitetura da camada C_1 da CMA.

inferior atuam como um seletor de saída de dados. O sinal de leitura (rd) determina se os valores utilizados para computações na camada seguinte são originados pelos MACs, ou se são originados da memória cache. Além de acessar a LUT e gerar os sinais de leitura/gravação para a memória cache, o bloco CACHE CTRL também implementa contadores para indexar o padrão em Z.

A LUT incluída no controlador de cache compreende uma memória somente de leitura composta por um vetor de bits. A tabela possui N_Z entradas, que correspondem ao número total de possíveis padrões em Z. A Equação 6.1 permite o cálculo desse número de entradas.

$$N_Z = \left(\frac{I_s - \alpha}{2} \right)^2, \tag{6.1}$$

onde I_s representa o tamanho da imagem e α o tamanho (largura) do kernel, menos 1. Esta expressão para N_Z é válida para o caso em que a camada P_2 usa janelas de tamanho 2×2.

Os elementos das extremidades (1,1), (1,14), (14,1) e (14,14) referem-se a padrões em Z que são calculados apenas uma vez. Portanto, considera-se um total de 192 linhas ($14^2 - 4$) para a LUT.

No início do cálculo de um produto interno, a linha correspondente ao padrão em Z é lida. Se contiver '0', não se espera que o resultado do MAC seja armazenado na memória cache, portanto, a computação continua e o resultado do MAC é propagado para a saída. Caso contrário, a memória cache é verificada a fim de determinar se os produtos internos já foram calculados. O Algoritmo 8 descreve simplificadamente uma consulta à LUT.

Algorithm 8 Busca na LUT pelo padrão Z com índice igual a z

```
if LUT(z) = 1 then
    cache query of inner products of the z-th Z-pattern
else
    compute the inner product
end if
```

As microarquiteturas das camadas convolucionais C_3 e C_5 são as mesmas implementadas na MLA. A Figura 6.4 apresenta o diagrama das camadas P_2 e P_4, onde d_s indica o número de barramentos, que é igual ao número de canais. Ou seja, $d_s = 6$ para a camada P_2 e $d_s = 16$ para P_4.

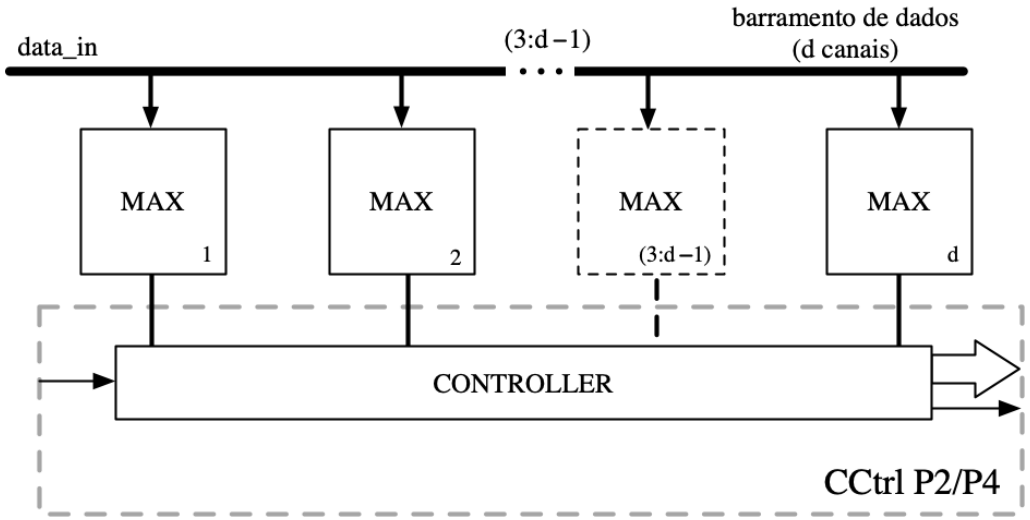


Figura 6.4: Microarquitetura resumida das camadas P_2 e P_4.

A proposta da CMA é compensar a perda de desempenho da MLA pela característica de não armazenar os resultados intermediários (consequentemente, a MLA precisa recalcular estes resultados toda vez que eles fizerem parte de uma convolução subsequente). Como o tamanho da memória cache na CMA pode ser configurado, é possível avaliar o balanceamento entre o uso de memória e o tempo de processamento. Para esta avaliação é necessário conhecer a quantidade de recursos

necessários para implementar cada *slot* da memória cache, e a melhoria alcançada, em número de ciclos de clock, quando um padrão em Z é armazenado.

A Tabela 6.1 apresenta a contagem do padrão em Z, classificado de acordo com o número de ocorrências. Como apresentado no Capítulo 5, o número de ocorrências pode ser calculado por meio das Equações (5.2) e (5.3), e o resultado é exibido na Figura 5.3. A Tabela 6.1 está ordenada pelo número de ocorrências de cada padrão em Z e apresenta quantos padrões distintos existem. Nesta tabela também é apresentado o incremento no desempenho, simulado em ciclos de clock, quando todos os padrões em Z de uma determinada classe (uma *classe* é um conjunto de padrões em Z que ocorrem, cada um, o mesmo número de vezes) são armazenados em cache. Esta simulação é realizada modificando o arquivo de *testbench* para permitir o armazenamento de apenas um padrão em Z de uma determinada classe e, usando um contador específico, determinando a melhoria no número de ciclos de clock. Este processo é repetido removendo sucessivamente algumas restrições para que toda a melhoria da classe possa ser computada.

Tabela 6.1: Melhoria em ciclos de clock de acordo com o número de padrões em Z armazenados na memória cache.

Classe	Número de padrões Z	Ocorrências de padrão Z	Melhoria por padrão Z	Melhoria por classe
1	36	25	2.592	93.312
2	24	20	2.052	49.248
3	4	16	1.620	6.480
4	24	15	1.512	36.288
5	8	12	1.188	9.504
6	24	10	972	23.328
7	4	9	864	3.456
8	8	8	756	6.048
9	8	6	540	4.320
10	24	5	432	10.368
11	12	4	324	3.888
12	8	3	216	1.728
13	8	2	108	864

Por exemplo, na Tabela 6.1, a classe 1 tem 36 padrões em Z únicos que ocorrem 25 vezes cada (entre os quais, 24 são repetições). Para cada um desses padrões em Z armazenados na memória cache, as simulações apresentam uma melhoria de 2.592 ciclos de clock. Se todos os 36 padrões em Z da classe forem armazenados, obteremos uma melhoria de 93.312 ciclos de clock. Desta forma, ao incluir na memória cache todos os padrões em Z da classe 10 que ocorrem 5 vezes cada, obtemos um ganho de desempenho maior ($24 \times 432 = 10.368$) do que armazenando os elementos da classe 3 ($4 \times 1.620 = 6.480$), por exemplo.

O tamanho total da memória cache compreende a quantidade de memória necessária para armazenar cada um dos *slots* (438 bits para os seis canais da C_1) e o número total de padrões em Z. Assim, armazenar a classe 10 requer $438 \times 24 = 10.512$ bits de memória cache, enquanto a classe 3 requer apenas $438 \times 4 = 1.752$ bits. Portanto,

podemos melhorar o desempenho em 60% ao armazenar padrões em Z da classe 10, em vez da classe 3 (conforme a coluna "Melhoria por classe" da Tabela 6.1, embora isso multiplique o uso de memória cache por um fator de 6 (de 1.752 bits para 10.512 bits). Para definir estratégias ou melhores abordagens para o uso da memória cache, esse *trade-off* deve ser avaliado.

A estrutura da arquitetura CMA tem origem no projeto da MLA e, conforme o conceito apresentado na Figura 5.1, ao término de um determinado número de computações de uma camada, o processamento da camada seguinte pode ser iniciado. Desta forma, o fluxo de dados é propagado ao longo de toda a rede de forma quase que contínua. O fluxo não é contínuo pois existe diferença no tempo de execução das operações entre as camadas. Da mesma forma que opera a MLA, a CMA também implementa um pipeline de cinco estágios.

6.3 Resultados

O projeto da CMA é implementado, simulado e sintetizado em VHDL utilizando os dispositivos Altera Cyclone II e Xilinx Zynq UltraScale+. A Tabela 6.2 apresenta os principais dados extraídos da síntese e simulação da CMA para os dois dispositivos. A coluna CMA_{min} apresenta os resultados obtidos ao usar memória cache para armazenar apenas um padrão em Z da classe 1, enquanto a coluna CMA_{max} contém os resultados obtidos quando todos os 192 padrões em Z são armazenados. A sequência de armazenamento desses padrões em Z na memória cache segue a mesma ordenação indicada na Figura 5.3. Os resultados da síntese para o dispositivo Zynq UltraScale+ demonstram ganho significativo de desempenho representado pela frequência de operação e pelo menor consumo de energia.

Tabela 6.2: Recursos usados nas sínteses para os FPGAs Cyclone II 2C70 e Zynq UltraScale+ XCZU28DR.

	Cyclone II 2C70		Zynq UltraScale+	
Métrica	CMA_{min}	CMA_{max}	CMA_{min}	CMA_{max}
Elementos lógicos/LUT	42.184	52.628	8.529	8.550
Memória (bits)	438	84.096	$48^{(*)}$	$3^{(**)}$
DSP	278	278	142	142
Registradores	5.970	6.348	7.559	7.561
Frequência (MHz)	56,2	56,2	116,95	116,95
Ciclos de clock	267.408	21.168	267.408	21.168
Tempo (ms)	4,758	0,376	2,286	0,181
Consumo (W)	0,308	0,310	0,077	0,081

$^{(*)}$ LUTRAM; $^{(**)}$ BRAM.

Na síntese da CMA, aplicando a restrição de tempo para o período de clock de 20 ns, o *worst negative slack* resultante é positivo, indicando que todas as restrições de tempo são atendidas e o projeto consegue operar acima da frequência mínima

definida em 50 MHz. A Tabela 6.3 apresenta os caminhos críticos da CMA nos dois FPGAs utilizados e as frequências de operação calculadas pelos valores dos WNS.

O caminho crítico é verificado na camada C_5 da MLA e, como as modificações na CMA concentram-se na camada C_1, os tempos de folga são parecidos nas duas implementações. O caminho crítico corresponde ao incremente de um contador, ao seu controle pela máquina de estados da camada, e à habilitação de uma operação do MAC.

Tabela 6.3: Caminhos críticos nas implementações da CMA.

Métrica	Cyclone II 2C70	Zynq XCZU28DR
De	C5:FSM	C5:FSM
Para	C5:MAC	C5:MAC
WNS (ns)	2,206	11,449
F_{max} (MHz)	56,20	116,95

6.4 Considerações Finais

Na simulação da implementação da CMA destaca-se o desempenho da extração de características de uma imagem, com o tempo de 0,376 ms, correspondendo a aproximadamente 2.660 imagens por segundo com consumo de potência dinâmica de 310 mW. A implementação da CMA confirma que o padrão de sequenciamento em padrão em Z pode ser bastante eficiente. A CMA inclui memória cache na camada C_1 para reduzir o número de repetições computacionais da MLA. A partir dos resultados da CMA, podemos apontar e analisar algumas compensações entre tempo de processamento, uso de elementos lógicos e de memória. A implementação da CMA, na configuração CMA_{max}, requer cerca de 20% mais recursos lógicos do que a implementação da MFA. Porém, a melhora no desempenho no tempo de extração das características, é superior a 3×. Também é relevante destacar que o consumo de memória na CMA é consideravelmente menor, totalizando aproximadamente 84 kb, em comparação com os 214 kb da MFA.

Os resultados referentes a precisão decorrente das features (características) extraídas pela CMA, em FPGA, são apresentados no Capítulo 7, juntamente com os resultados das demais arquiteturas propostas. Também são apresentados os resultados da síntese e simulação da CMA utilizando as duas abordagens quanto ao uso da memória cache, CMA_{min} e CMA_{max}.

Capítulo 7

Análise dos Resultados

A CNN LeNet-5 foi treinada em software com o conjunto de dados MNIST com 60.000 imagens [25]. As quatro arquiteturas de hardware são implementadas, simuladas e sintetizadas em VHDL. A partir de uma imagem de entrada 32×32, as arquiteturas MLA, MFA e CMA são simuladas de forma a extrair um vetor com 120 características. O tempo necessário para isso, entre outras características de desempenho (consumo, por exemplo), é medido. Por fim, a inferência se completa em software, com a execução de uma rede densa (*fully-connected*) com topologia 120-84-10, que classifica o dígito a partir das 120 features. O tempo gasto nesta etapa em software não é contabilizado.

7.1 Acurácia

Os parâmetros para a CNN treinada foram obtidos em [40] e convertidos de números reais para números de ponto fixo usando o formato (16,8) para kernels e (32,16) para biases. O formato do bias e também dos resultados intermediários, ou seja, ponto fixo de 32 bits foi empregado para não perder a precisão dos resultados das multiplicações antes do truncamento para 16 bits. A Figura 4.1 apresenta a multiplicação de um número do kernel por outro número de 16 bits e, na sequência, a adição do resultado da multiplicação a um número de 32 bits. O truncamento é aplicado somente após a última operação MAC.

Os testes consistem em avaliar os resultados obtidos com o processamento de 10 mil imagens de teste (que não foram usadas para treinamento) na implementação em software, a ser usada como referência, e os resultados proporcionados pelas arquiteturas propostas. A implementação em software obtém uma precisão de 98,08% (9.808 acertos e 192 erros) e sua matriz de confusão é apresentada na Tabela 7.1. Todas as arquiteturas de hardware geram features (características), a partir das quais o classificador em software alcança precisão igual a 98,17%. Como esperado, todas as arquiteturas geram características que proporcionam a mesma

precisão, uma vez que suas unidades de processamento utilizam o mesmo formato para a representação numérica e truncamento. A matriz de confusão é apresentada na Tabela 7.2, comprovando que as arquiteturas propostas geram features que proporcionam precisão de 98,17%, similar àquela obtida com a implementação de software (98,08%).

Tabela 7.1: Matriz de confusão obtida pela implementação em software da LeNet-5.

Classe Real \ Classe Prevista	0	1	2	3	4	5	6	7	8	9
0	978	0	0	0	0	0	1	1	0	0
1	0	1133	0	0	0	0	0	0	2	0
2	9	6	986	0	5	0	2	9	13	2
3	4	2	1	972	0	13	0	2	11	5
4	1	0	0	0	976	0	0	1	1	3
5	5	1	0	2	0	874	4	1	5	0
6	7	2	0	0	1	2	943	0	3	0
7	1	7	2	2	2	0	0	1005	1	8
8	12	1	0	0	2	1	1	1	952	4
9	7	3	0	0	7	1	0	1	1	989

Tabela 7.2: Matriz de confusão obtida a partir das features (características) extraídas pelas implementações da MFA, MLA, MMA e CMA.

Classe Real \ Classe Prevista	0	1	2	3	4	5	6	7	8	9
0	978	0	0	0	0	0	1	1	0	0
1	1	1132	0	0	0	0	0	0	2	0
2	9	7	989	0	4	0	2	10	9	2
3	3	2	1	970	0	17	0	2	8	7
4	1	0	0	0	976	0	0	1	1	3
5	4	1	0	1	0	879	4	1	2	0
6	6	2	0	0	1	2	944	0	3	0
7	1	7	2	2	2	0	0	1005	1	8
8	11	1	0	0	1	2	1	1	951	6
9	5	2	0	0	8	0	0	1	0	993

7.2 Simulação

A Tabela 7.3 apresenta os requisitos de área (elementos lógicos, memória, DSPs e registradores), tempo e energia das implementações das arquiteturas propostas. Conforme apresentado no Capítulo 6, a coluna CMA_{min} apresenta os resultados obtidos ao usar memória cache para armazenar apenas um padrão em Z da classe 1, enquanto a coluna CMA_{max} contém os resultados obtidos quando todos os 192 padrões em Z são armazenados.

Em termos de elementos lógicos, a Tabela 7.3 indica que as implementações da MFA, MLA, MMA e CMA_{min} usam um número semelhante de recursos, com cerca de 4% de diferença. Porém, quando a CMA é configurada para armazenar todos os padrões em Z (CMA_{max}), ela apresenta um aumento significativo de elementos

Tabela 7.3: Recursos utilizados nas implementações para o FPGA Cyclone II 2C70.

Métrica	MFA	MLA	MMA	CMA_{min}	CMA_{max}
Elementos lógicos	43.876	42.036	43.628	42.184	52.628
Memória (bits)	214.344	0	0	438	84.096
DSP	278	278	296	278	278
Registradores	6.198	5.898	6.348	5.970	6.348
Frequência (MHz)	47,15	55,48	55,15	56,2	56,2
Ciclos de clock	64.650	270.000	90.000	267.408	21.168
Tempo (ms)	1,375	4,866	1,632	4,758	0,376
Speedup	-	0,3	0,8	0,3	3,7
Consumo (W)	0,326	0,303	0.323	0,308	0,310

lógicos, cerca de 25%. Como a MLA não utiliza memória, então ela deve requerer menos área do FPGA. A MFA utiliza tamanho fixo de memória de 214 kbits, dos quais 8 kbits são empregados para armazenar a imagem de entrada. Mesmo no caso da CMA_{max}, o uso de memória é inferior a 40% do total utilizado pela MFA (84.096 e 214.344, respectivamente). A CMA não usa memória para armazenar a imagem de entrada. A Figura 7.1 apresenta o uso de memória e elementos lógicos, de acordo com o número de padrões em Z armazenados na memória cache. Com zero padrão em Z armazenado na memória cache, o uso de elementos lógicos pela CMA é próximo da MLA, sendo indicado pelas curvas verde (CMA) e azul (MLA). Quando a CMA armazena 32 padrões em Z na cache, a utilização de elementos lógicos é semelhante à da MFA, conforme indicado pela interseção das curvas sólidas azul (CMA) e vermelha (MFA) no gráfico. A Figura 7.1 também apresenta a utilização da memória. Nas implementações MFA e MLA, o tamanho requerido é constante: 214.344 bits na MFA (linha tracejada vermelha) e zero na MLA (linha tracejada verde). Na CMA, o tamanho da memória requerida varia de 0 a 84 kbits (linha tracejada azul).

O consumo de energia das arquiteturas propostas é estimado utilizando o software Altera Quartus II. Esta estimativa refere-se ao consumo dinâmico de energia de um FPGA Cyclone II 2C70. O software indica um consumo de energia semelhante para todas as arquiteturas, com pequenas variações na estimativa. Esses números semelhantes, em torno de 310 mW, contribuem a favor da possibilidade de aplicação desses projetos de CNN em sistemas embarcados de baixa potência.

A MLA requer cerca de 4× mais ciclos de clock que a MFA, ou seja, 270.000 *versus* 64.650 ciclos de clock, verificando assim que, para essas duas arquiteturas, o uso de memória está inversamente relacionado ao tempo de processamento. Por armazenar resultados de operações que, na MLA, teriam que ser recalculados, a CMA é capaz de acelerar o tempo de processamento em até 13× (reduzindo assim o número de ciclos de clock de 270.000 para 21.168). O tempo de processamento é calculado usando o produto entre o período de clock (inverso da frequência máxima) e o número total de ciclos de clock necessários para realizar a parte da inferência referente às camadas C_1, P_2, C_3, P_4 e C_5 (ou seja, a parte da inferência referente

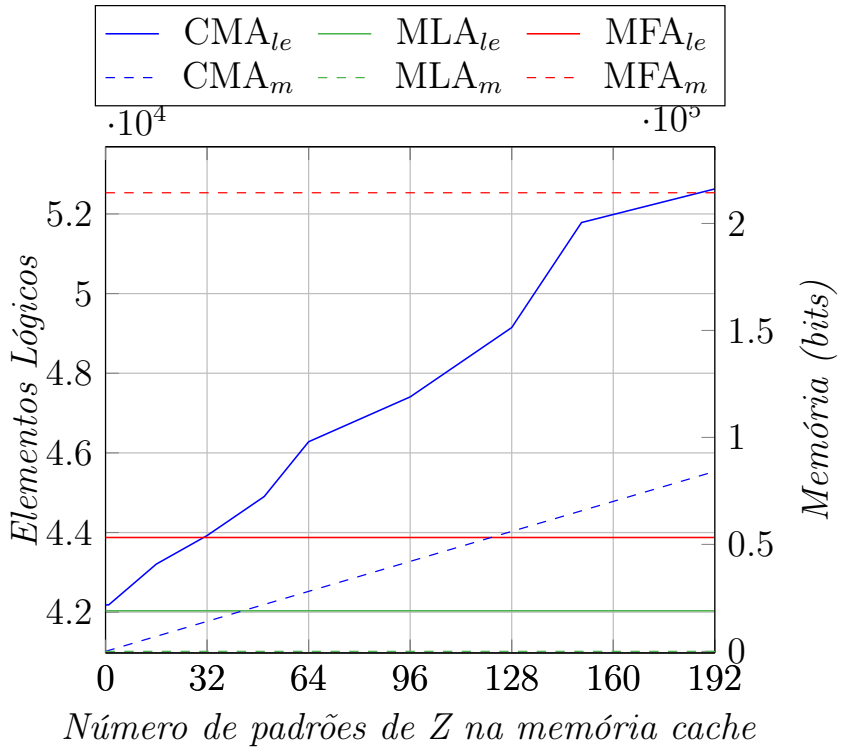


Figura 7.1: Número de elementos lógicos e uso de memória (10^5 bits).

à extração de features, ou características). A Figura 7.2 apresenta o número de ciclos de clock e o tempo de processamento como funções do número de padrões em Z armazenados em memória cache. A ordem segundo a qual os padrões em Z são armazenados na memória cache é a mesma mostrada na Tabela 6.1, ou seja, da classe 1 à classe 13. Para simular esta sequência é necessário incluir alguns circuitos de controle no arquivo de *testbench*. Como no caso dos elementos lógicos, o tempo de processamento e os ciclos de clock também se aproximam dos da MLA quando o número de padrões em Z na memória cache é próximo de zero. Como esperado, o tempo de processamento da CMA é minimizado quando todos os 192 padrões em Z são armazenados na memória cache, resultando em um *speedup* de 3,7 em relação à arquitetura de referência, a MFA. O cálculo do *speedup* pode ser expresso como a razão entre o tempo de execução da arquitetura de referência (MFA) e o tempo de execução da arquitetura proposta. Quanto maior o valor do *speedup*, maior é a melhoria no desempenho. O *speedup* é uma métrica importante pois auxilia na comparação de diferentes implementações e otimizações, indicando o ganho de desempenho alcançado com as mudanças implementadas.

A escolha de quais classes de padrões em Z serão armazenadas na memória cache é importante, porque o número de ocorrências influencia a redução no número total de ciclos de clock. A Figura 7.3 apresenta a CMA com 36 slots de memória cache para três estratégias denotadas por S_k, com k=1, 2 ou 3. A estratégia S_1 armazena na memória cache somente os padrões em Z com maior ocorrência, ou

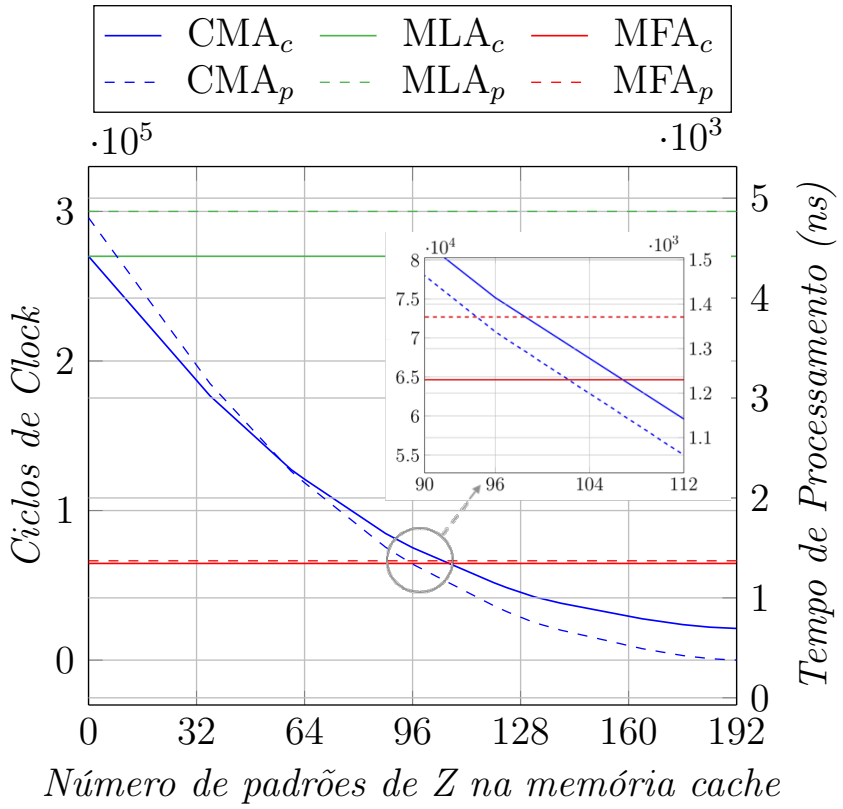


Figura 7.2: Comparação do número total de ciclos de clock e do tempo total de processamento, com destaque para a região de interseção entre CMA e MFA.

seja, apenas a classe 1 indicada na linha 1 da Tabela 6.1, conseguindo ganho de 2.592 ciclos de clock para cada padrão em Z armazenado (ao todo, a economia, ou o ganho, é de 2.592×36 ciclos de clock). Na estratégia S_2, os padrões em Z são armazenados de acordo com a sequência definida pelos índices A das entradas na Figura 5.3, e o ganho de desempenho varia dependendo da classe que está sendo armazenada no momento. Com a estratégia S_2, é possível alcançar uma melhoria global de 39.744 ciclos de clock. A estratégia S_3 armazena os padrões em Z com as menores ocorrências, ou seja, todos os padrões em Z das classes 13, 12, 11 e oito padrões em Z da classe 10, correspondendo a 36 padrões em Z armazenados na memória cache. A estratégia S_3 de priorizar os padrões em Z de menor ocorrência resulta em uma aceleração geral de apenas 9.936 ciclos.

A Figura 7.4 apresenta como o tempo de processamento é distribuído entre as camadas C_1, P_2, C_3, P_4 e C_5. O tempo de processamento está concentrado principalmente na camada C_1, nas arquiteturas MFA e MLA. Na MFA, essa concentração é atribuída ao tempo de acesso à memória, enquanto na MLA a concentração é resultado do grande número de repetição de operações. Por outro lado, a CMA apresenta uma distribuição mais equilibrada no tempo de processamento, por eliminar repetições na camada C_1.

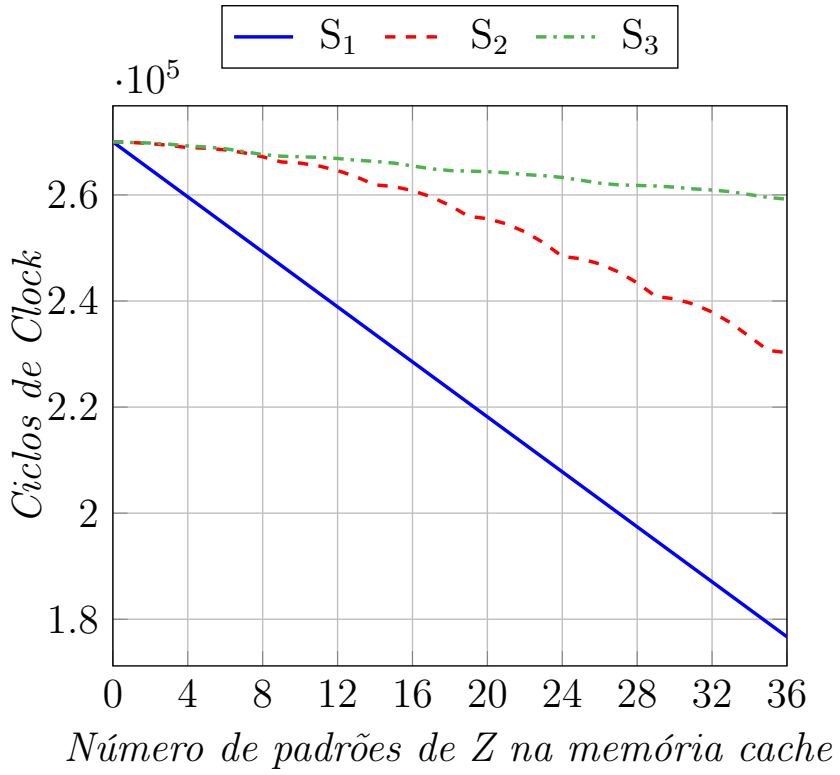


Figura 7.3: Comparação do número total de ciclos de clock e padrões em Z armazenados em cache.

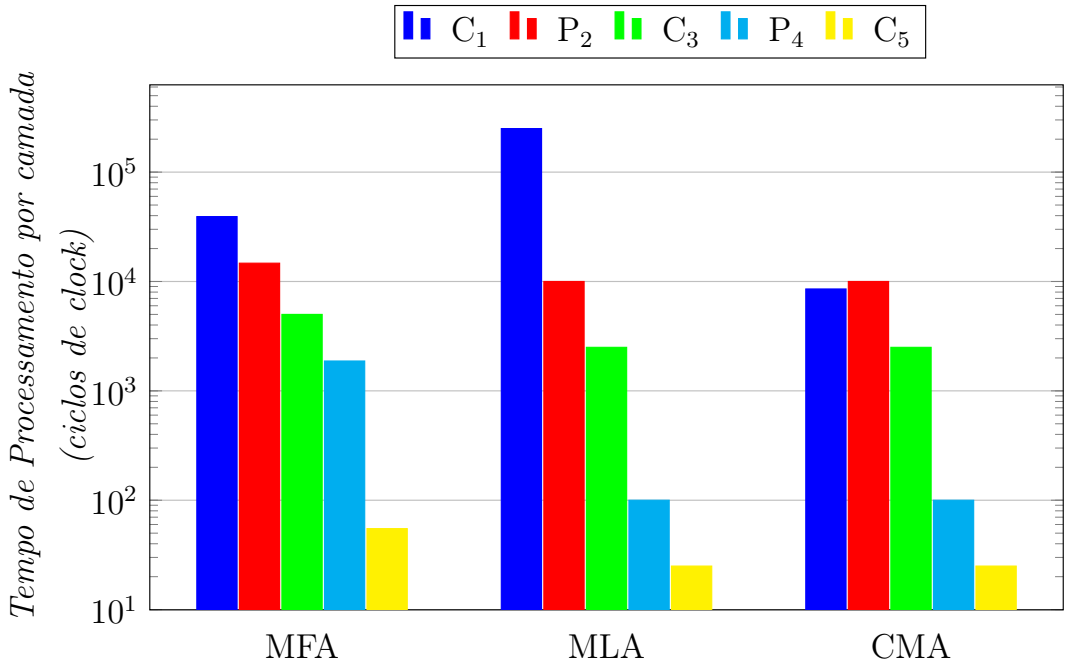


Figura 7.4: Tempo de processamento por camada para as arquiteturas MFA, MLA e CMA.

7.3 Comparação dos Resultados

A precisão de 98,17%, alcançada a partir das features (características) extraídas via MFA, MLA, MMA e CMA, utilizando ponto fixo de 16 bits, é consistente com a literatura, que indica precisão variando de 96% a 99,6% conforme apresentado na Tabela 7.4. Em termos de utilização de memória, a MFA requer até 25% menos memória em comparação com [9][1]. A CMA usa cerca de 29% da memória de [9], mesmo usando sua configuração de máxima memória, que requer 84 kbits. A Tabela 7.4 apresenta a memória total e os elementos de DSP usados em diferentes implementações da CNN LeNet-5.

O desempenho depende do FPGA utilizado, mas mesmo utilizando um FPGA de baixo custo, o desempenho das arquiteturas propostas é comparável ao encontrado na literatura. O trabalho apresentado em [71], que utiliza Xilinx Zedboard equipado com um FPGA Zynq XC7Z020, classifica uma imagem em 6,8 ms. A MLA, por exemplo, extrai, em 4,8 ms, as features de uma imagem. Ao comparar implementações que possuem consumo de energia semelhante (na ordem de mW), as arquiteturas propostas alcançam desempenho para a extração das características até 2,5× maior, p.ex, 0,376 ms na CMA_{max} (Tabela 7.3) e 0,960 ms apresentado em [72]. As implementações mais recentes [73–75] apresentadas na tabela utilizam a plataforma PYNQ-Z2. Uma vantagem notável da placa PYNQ-Z2 é sua capacidade de permitir o controle de um dispositivo Zynq a partir de programas Python executados no sistema de processamento baseado em ARM. Para reduzir a influência da tecnologia utilizada em cada um dos FPGAs, uma medida de eficiência é incluída na Tabela 7.4. Ela é calculada, a partir do consumo de energia e do tempo de processamento.

A Tabela 7.5 apresenta algumas implementações que usam SoC com GPU embarcada. As implementações de GPUs se beneficiam da disponibilidade de *frameworks* e APIs (*application programming interface*). Em geral, os tempos de inferência apresentados pelas GPUs são menores, mas exigem alto consumo de energia. O tempo que a CMA gasta para extrair features (características) de uma imagem é comparável ao tempo total (features e classificação) gasto pelas GPUs, mas o consumo de energia da CMA é mais baixo.

7.4 Escalabilidade

Com o FPGA Zynq UltraScale+ XCZU28DR, que utiliza tecnologia mais moderna, de 16 nm, é possível obter um ganho significativo de desempenho. A Tabela 7.6

[1]Para esta comparação foi usada como referência apenas a quantidade de memória utilizada na implementação da CNN (285 kb).

Tabela 7.4: Comparação com outras implementações de Lenet-5 em FPGA.

Métrica	[7]	[9]	[72]	[73]	[74]	[71]	[75]	MFA	MLA	MMA	CMA
Acurácia (%)	97,6	96,0	-	98,3	97,1	97,0	98,2	98,1	98,1	98.1	98.1
Dados (bits)	12	13	8	32	8	-	32	16	16	16	16
Mem. (kb)	976	371	-	-	-	-	-	214	0	84	0
DSP	158	77	574	62	-	188	23	278	278	296	278
Freq (MHz)	100	143	-	100	100	-	100	47,15	55,48	55,15	56,20
Tempo (ms)	142,8	1,61	0,96	170,0	929,0	6,80	653,0	1,38	4,86	1,62	0,38
Cons. (W)	0,12	-	0,47	-	-	4,35	-	0,33	0,30	0,32	0,31
Eficiência	0,003	-	2,04	-	-	2,6	-	1	0,26	-	3,4
Ano	2020	2019	2017	2022	2022	2019	2022	2021	2021	2023	2023

(*) Dados = quantidade de bits na representação em ponto fixo, Mem. = memória, Cons. = Consumo.
(**) Os tempos registrados pelos trabalhos das referências são relacionados à inferência.
(***) Os valores de consumo de energia referem-se ao consumo dinâmico ou, então, não foram informados nas referências.

Tabela 7.5: Comparação com implementações contendo GPU embarcada.

Métrica	[76]	[77]	[77]	[78]	[79]	MFA	MLA	MMA	CMA
Dispositivo	T860	Jetson Tx2	Jetson Tx2	Jetson Nano	Jetson Tx1	Cyclone II			
Tempo (ms)	606	0,074	0,098	0,008	0,910	1,370	4,860	1,632	0,376
Consumo (W)	-	6,87	24,6	6.12	-	0,33	0,30	0,32	0,31
Ano	2019	2022	2022	2020	2018	2021	2021	2023	2023

apresenta o ganho de desempenho alcançado nas arquiteturas propostas ao utilizar esse FPGA mais moderno. O ganho é apresentado pela frequência máxima de clock e tempo necessário para extrair features de uma imagem. O consumo dinâmico de energia do projeto também é significativamente menor.

Tabela 7.6: Recursos utilizados nas implementações para o FPGA Zynq UltraScale+ XCZU28DR.

Métrica	MFA	MLA	MMA	CMA_{min}	CMA_{max}	Scaled MLA
LUT	9.399	8.521	9.122	8.529	8.550	9.716
Registradores	7.590	7.459	8.125	7.559	7.561	10.264
DSP	142	142	160	142	142	226
BRAM	22	0	0	$0^{(*)}$	3	0
Frequência (MHz)	88,07	97,48	117,72	116,95	116,95	115,22
Tempo (ms)	0,734	2,770	0,765	2,286	0,181	17,358
Consumo (W)	0,143	0,076	0,089	0,077	0,081	0,087

(*) 48 LUTRAM é utilizada ao invés de BRAM

A Tabela 7.6 também apresenta um projeto baseado na MLA escalonada para 20 camadas, com 16 camadas convolucionais e 4 camadas de *pooling*. A imagem de entrada possui 256×256 pixels no formato RGB e, portanto, as primeiras quatro camadas convolucionais possuem 3 canais, as dez seguintes possuem 6 canais e as duas últimas possuem 16 e 120, respectivamente. Esta configuração é usada para determinar o tempo para cálculo de features apresentado na Tabela 7.6.

7.5 Considerações Finais

Neste capítulo, os resultados da classificação realizada em software, utilizando as características extraídas por hardware, foram comparados com os resultados obti-

dos com a rede neural implementada totalmente em software. Nessa comparação, observamos que a acurácia alcançada pelas implementações em hardware é equivalente àquela conseguida no software. Quanto ao uso de recursos do FPGA, as arquiteturas propostas têm custos semelhantes para alguns recursos lógicos. Uma das maiores diferenças reside na quantidade de memória utilizada pela MFA, que, mesmo na comparação com a CMA_{max}, é aproximadamente $2,5\times$ maior. Por outro lado, na mesma configuração, a CMA requer 20% mais elementos lógicos do que a MFA. No que diz respeito ao desempenho na extração de características, a MLA, ao aplicar o padrão em Z e reorganizar a imagem de entrada, requer um número de ciclos de clock consideravelmente superior ao da MFA (cerca de $4\times$). Por outro lado, a CMA, ao utilizar elementos de memória cache e adotar critérios para o armazenamento de dados intermediários, consegue aprimorar o desempenho em relação à MLA em quase 13 vezes.

Ao implementar as arquiteturas em um dispositivo mais moderno, com tecnologia de fabricação superior, é esperado um desempenho aprimorado. Os resultados da síntese para um dispositivo Zynq UltraScale+ confirmam as expectativas de melhoria alcançadas nas arquiteturas propostas, resultando em uma redução no tempo de processamento para a extração de características, aumento na frequência máxima de operação, baixo consumo de potência e utilização eficiente dos recursos do FPGA. Para complementar estas considerações, no Capítulo 8 são apresentadas as conclusões da tese e alguns trabalhos futuros.

Capítulo 8

Conclusão

Este trabalho descreve arquiteturas de hardware para implementação de uma CNN. A MFA armazena os mapas de características de todos os canais nas camadas da CNN. Embora a execução dos canais seja totalmente em paralelo, a estrutura inter-camadas é sequencial. O projeto da MLA é proposto para eliminar totalmente o uso de memória, aplicando o padrão em Z à entrada. Esta arquitetura implementa um pipeline de 5 estágios para as camadas da rede. Para melhorar principalmente o desempenho da MLA, a MMA utiliza quatro componentes MACs para calcular os quatro produtos internos simultaneamente na primeira camada da rede neural. A proposta da CMA introduz o uso de memória cache, de tamanho personalizado, para equilibrar os requisitos de memória e melhorias de desempenho. A CMA mantém a mesma estrutura de pipeline para as camadas implementada na MLA. A estrutura intra-camada de todas as arquiteturas é massivamente paralela.

8.1 Considerações

Nas arquiteturas propostas nesta tese, fazemos uso dos blocos de memória disponíveis dentro do próprio FPGA, levando em consideração a latência de um ciclo de clock para cada operação relacionada à memória. Uma das arquiteturas apresentadas dispensa completamente o uso de memória, enquanto outra abordagem se destaca ao incorporar uma memória cache para armazenar e reutilizar os resultados intermediários. Para representar numericamente esses resultados intermediários, bem como os valores dos *biases*, são empregados formatos de ponto fixo de 32 bits, enquanto todos os outros sinais nas arquiteturas são representados por valores de ponto fixo de 16 bits. Esse enfoque permite alcançar a precisão de classificação desejada, otimizar a utilização da área disponível no FPGA e eliminar a necessidade de recursos como FPU. A acurácia na classificação foi avaliada utilizando uma rede neural *fully-connected* implementada em software, que se utilizou das características extraídas pelas arquiteturas de hardware.

A MFA representa a concepção de uma arquitetura de hardware de uma CNN que foi previamente treinada em ambiente de software. Os resultados obtidos com a implementação desta arquitetura demonstram que o tempo de processamento e o consumo de energia estão em conformidade com os resultados apresentados na literatura. A principal desvantagem da MFA reside no processamento sequencial das camadas. Considerando a premissa de uso de um FPGA de baixo custo, a utilização de 214 kb de memória também pode ser considerada uma desvantagem. Para otimizar o uso de memória RAM, o projeto da MLA é proposto.

A premissa da MLA é eliminar por completo o uso de memória RAM. Para alcançar esse objetivo, é proposta uma modificação na leitura da imagem de entrada, chamada de "padrão em Z". A utilização desse padrão personalizado garante que as operações aritméticas sejam executadas na sequência necessária para a correta propagação dos dados para as camadas subsequentes, sem a necessidade de utilizar memória para armazenar os mapas de características. A principal vantagem desta arquitetura, além de não fazer uso de memória RAM, é o processamento das camadas em pipeline. No entanto, a eliminação completa do uso de memória não se mostrou eficaz em relação ao tempo de processamento, resultando em um aumento significativo no número de ciclos de clock necessários para a extração de características, conforme demonstrado pela MFA. Para equilibrar melhor o custo em relação ao desempenho, é apresentada outra proposta de arquitetura, a MMA.

A MMA inclui quatro componentes MACs na primeira camada convolucional para reduzir a repetição dos dados de entrada e das operações computadas. Isso é alcançado através da implementação de um pipeline para a execução das quatro operações de produto interno. Esta arquitetura, que utiliza um pipeline dentro da camada C_1, consegue melhorar o desempenho da rede neural em até três vezes em comparação com a MLA.

A CMA utiliza o mesmo padrão de leitura e sequenciamento usado na MLA, mas tem como objetivo melhorar o desempenho do processamento ao armazenar parte dos resultados intermediários previamente calculados em uma memória cache. A melhora no tempo de processamento alcançada pela CMA em relação à MLA é bastante significativa. Ainda em relação à MLA, a implementação da CMA aumenta o custo dos recursos de hardwares requeridos, cerca de 25% para elementos lógicos e 7% para registradores.

O projeto da MFA apresenta um tempo de processamento menor (cerca de 1,4 ms) que a MLA (4,8 ms). Porém, na MLA, as repetições ocorrem principalmente nas camadas C_1 e P_2, e a propagação do resultado de uma camada para outra também requer menos tempo, uma vez que não é necessário o cálculo e armazenamento dos mapas de características completos, reduzindo assim o tempo ocioso do componentes da camada e eliminando o uso de memória RAM.

O objetivo da CMA é equilibrar o uso da memória e o tempo de processamento. A CMA inclui memória cache na camada C_1, a fim de reduzir o número de repetições computacionais da MLA. Dessa forma, reduzimos o número de ciclos de clock necessários para a extração de features (características). A partir dos resultados da CMA, podemos apontar e analisar algumas compensações entre tempo de processamento, uso de elementos lógicos e uso de memória. Por exemplo, o menor tempo de processamento é obtido usando o maior tamanho de memória e o menor número de elementos lógicos. Esta vantagem é alcançada na configuração máxima de memória CMA_{max}, que requer 84 kbits de memória e 52.628 elementos lógicos, resultando assim em tempo para extração de características igual a 0,376 ms. Ao comparar a CMA com a MFA, vemos que, mesmo com o menor tempo de processamento, o uso de memória na CMA (84 kbits) é significativamente menor do que o uso de memória na MFA (214 kbits). A CMA atinge o mesmo número de ciclos de clock que a MFA (64.650 ciclos de clock) quando 107 padrões em Z são armazenados na memória cache. Nesse ponto de interseção, são necessários aproximadamente 50 kbits de memória para a CMA, enquanto 214 kbits são necessários para a MFA.

Por meio de simulações e sínteses, a CMA apresenta uma boa flexibilidade na utilização dos recursos disponíveis no dispositivo FPGA, permitindo assim uma utilização equilibrada de tais recursos. O uso de memória cache na camada C_1 também fornece o menor tempo de computação (0,376 ms) em comparação com as arquiteturas MFA e MLA (1,375 ms e 4,866 ms, respectivamente).

8.2 Trabalhos Futuros

A classificação realizada em software alcançou 9.808 acertos, enquanto que com as features (características) extraídas por MFA, MLA e CMA, 9.817 acertos. As matrizes de confusão mostram que existem mais diferenças de classificação do que em apenas 9 imagens. É provável que mudanças, de precisão numérica e de métodos de truncamento, tenham impacto também nas camadas *fully-connected* das CNNs e, portanto, impacto nos resultados de classificação. O estudo da implementação das camadas *fully-connected* que, neste trabalho foram implementadas em ambiente numérico (software), é um tópico importante para investigação futura.

O padrão em Z é usado na MLA e CMA para otimizar o uso de recursos como memória e tempo de processamento. O desempenho da MFA pode ser potencialmente aprimorado ao implementar uma variação do padrão em Z, eliminando a necessidade da memória RAM_S1 utilizada para armazenar a imagem de entrada. Além disso, antecipar o início do processamento de cada camada subsequente e incorporar um pipeline também deverá aumentar o desempenho da MFA. O benefício de alguns

métodos especializados para camadas convolucionais, como os algoritmos im2col e Winograd, é um tópico para investigação futura.

Alguns trabalhos relacionados [51, 72, 74] exploram o emprego de representações numéricas em ponto fixo de 8 bits para pesos. Investigar a otimização dos recursos lógicos do FPGA e a taxa de erro, por meio da redução da quantidade de bits utilizada nas representações numéricas, é um tópico de pesquisa relevante. Ao escalar as arquiteturas propostas para implementar CNNs maiores como VGG e ResNet, pode-se verificar o desempenho e os requisitos de recursos. Adicionalmente, pode-se avaliar o impacto de técnicas como convolução dilatada e *padding*, *skipping connections*, *channel shuffling*, e módulo *inception*.

Outro tema de trabalho futuro seria investigar o uso de memória cache na camada P_2, visto que repetições de operações também estão presentes nesta camada. Também é possível implementar algum algoritmo de substituição de memória cache, como *Least Frequently Used*, *Last Recently Used* ou algumas de suas variações. Isso reduziria o tamanho da memória cache, por meio da reutilização de seus blocos para armazenar diferentes padrões em Z. Uma alternativa para reduzir os requisitos de memória cache é a implementação de um algoritmo de análise dinâmica de padrões em Z. Isso permite a construção de uma tabela dinâmica para o uso da memória cache, substituindo a *lookup table* empregada na CMA.

Referências Bibliográficas

[1] PÉREZ, I., FIGUEROA, M. "A Heterogeneous Hardware Accelerator for Image Classification in Embedded Systems", *Sensors*, v. 21, n. 8, 2021. ISSN: 1424-8220. doi: 10.3390/s21082637.

[2] DE FRANÇA, A. B. Z., OLIVEIRA, F. D. V. R., GOMES, J. G. R. C., et al. "Non-Memoryless vs. Memoryless Hardware Architectures for Convolutional Neural Networks". In: *2021 IEEE 12th Latin America Symposium on Circuits and System (LASCAS)*, pp. 1–4, 2021. doi: 10.1109/LASCAS51355.2021.9459115.

[3] DE FRANÇA, A. B. Z., OLIVEIRA, F. D. V. R., GOMES, J. R. C., et al. "Hardware designs for convolutional neural networks: Memoryful, memoryless and cached", *Integration*, v. 94, pp. 102074, 2024. doi: 10.1016/j.vlsi.2023.102074.

[4] SIDDIQUI, F., AMIRI, S., MINHAS, U. I., et al. "FPGA-Based Processor Acceleration for Image Processing Applications", *Journal of Imaging*, v. 5, n. 1, 2019. ISSN: 2313-433X. doi: 10.3390/jimaging5010016.

[5] LI, J., CHI, Y., CONG, J. "HeteroHalide: From Image Processing DSL to Efficient FPGA Acceleration". In: *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '20, pp. 51–57, New York, NY, USA, 2020. Association for Computing Machinery. ISBN: 9781450370998. doi: 10.1145/3373087.3375320.

[6] TSIKTSIRIS, D., ZIOUZIOS, D., DASYGENIS, M. "A High-Level Synthesis Implementation and Evaluation of an Image Processing Accelerator", *Technologies*, v. 7, pp. 4, 12 2018. doi: 10.3390/technologies7010004.

[7] GONZALEZ, E., LUNA, W., FAJARDO, C. "A Hardware Accelerator for The Inference of a Convolutional Neural Network", *Ciencia e Ingeniería Neogranadina*, v. 30, pp. 107–116, 2020. doi: 10.18359/rcin.4194.

[8] LOPES, I. D. C. *Convolutional Neural Network Reliability on an APSoC platform - a traffic-sign recognition case study.* Tese de Mestrado, Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Microeletrônica, 2017.

[9] SOLOVYEV, R., KUSTOV, A., TELPUKHOV, D., et al. "Fixed-Point Convolutional Neural Network for Real-Time Video Processing in FPGA". In: *2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, pp. 1605–1611, 2019. doi: 10.1109/EIConRus.2019.8656778.

[10] HAN, S., KANG, J., MAO, H., et al. "ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pp. 75–84, New York, NY, USA, 2017. Association for Computing Machinery. ISBN: 9781450343541. doi: 10.1145/3020078.3021745.

[11] LEE, M., HWANG, K., PARK, J., et al. "FPGA-Based Low-Power Speech Recognition with Recurrent Neural Networks". In: *2016 IEEE International Workshop on Signal Processing Systems (SiPS)*, pp. 230–235, 2016. doi: 10.1109/SiPS.2016.48.

[12] PAN, S.-T., LAN, M.-L. "An efficient hybrid learning algorithm for neural network–based speech recognition systems on FPGA chip", *Neural Computing and Applications*, v. 24, n. 7, pp. 1879–1885, 2014. doi: 10.1007/s00521-013-1428-5.

[13] MOTAMEDI, M., GYSEL, P., AKELLA, V., et al. "Design space exploration of FPGA-based Deep Convolutional Neural Networks". In: *21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 575–580, 2016. doi: 10.1109/ASPDAC.2016.7428073.

[14] WEI, X., YU, C. H., ZHANG, P., et al. "Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs". In: *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, New York, NY, USA, 2017. Association for Computing Machinery. ISBN: 9781450349277. doi: 10.1145/3061639.3062207.

[15] PEREIRA, P. M. C. *Efficient hardware implementation of 2D convolutions, optimized for point cloud and adjustable to the 3D model requirements for object detection and classification.* Tese de Mestrado, Universidade do Minho, 2021.

[16] GANAHL, M., BEALL, J., HAURU, M., et al. "Density Matrix Renormalization Group with Tensor Processing Units", *PRX Quantum*, v. 4, n. 1, pp. 010317–, 02 2023. doi: 10.1103/PRXQuantum.4.010317. Disponível em: <https://link.aps.org/doi/10.1103/PRXQuantum.4.010317>.

[17] PESERICO, N., SHASTRI, B. J., SORGER, V. J. "Integrated Photonic Tensor Processing Unit for a Matrix Multiply: A Review", *Journal of Lightwave Technology*, v. 41, n. 12, pp. 3704–3716, jun 2023. doi: 10.1109/jlt.2023.3269957.

[18] XILINX. *Introduction to FPGA Design with Vivado High-Level Synthesis*. Relatório Técnico UG998 (v1.1), Xilinx, 2019.

[19] KAESLIN, H. *Digital Integrated Circuit Design - From VLSI Architectures to CMOS Fabrication*. Cambridge University Press, 2008.

[20] XILINX. *Vitis High-Level Synthesis User Guide*. Relatório Técnico UG1399, Xilinx, 2022.

[21] XILINX. *Vivado Design Suite User Guide: High-Level Synthesis*. Relatório Técnico UG902 (v2020.1), Xilinx, 2021.

[22] FINGEROFF, M., BOLLAERT, T. *High-Level Synthesis Blue Book*. Mentor Graphics Corporation, 2010.

[23] Disponível em: <https://www.xilinx.com/products/design-tools/vivado.html>.

[24] Disponível em: <https://www.intel.com.br/content/www/br/pt/software/programmable/quartus-prime/hls-compiler.html>.

[25] LECUN, Y., BOTTOU, L., BENGIO, Y., et al. "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, v. 86, n. 11, pp. 2278–2324, 1998. doi: 10.1109/5.726791.

[26] SHAHSHAHANI, M., GOSWAMI, P., BHATIA, D. "Memory Optimization Techniques for FPGA based CNN Implementations". In: *2018 IEEE 13th Dallas Circuits and Systems Conference (DCAS)*, pp. 1–6. IEEE, 2018. doi: 10.1109/DCAS.2018.8620112.

[27] ARMENIAKOS, G., ZERVAKIS, G., SOUDRIS, D., et al. "Hardware Approximate Techniques for Deep Neural Network Accelerators: A Survey", *ACM Computing Surveys*, v. 55, n. 4, pp. 1–36, nov 2022. doi: 10.1145/3527156.

[28] BABU, A., SAIKIRAN, R., S, S. "Design of floating point multiplier for signal processing applications", *International Journal of Applied Engineering Research*, v. 8, pp. 715–722, 01 2013.

[29] PITSIS, A. G. *Design and Implementation of an FPGA-Based Convolutional Neural Network Accelerator*. Tese de Mestrado, Technical University of Crete, 2018.

[30] LECUN, Y., CORTES, C., BURGES, C. J. "The MNIST Database of handwritten digits". Disponível em: <`http://yann.lecun.com/exdb/mnist`>.

[31] BOUGUEZZI, S., FREDJ, H. B., BELABED, T., et al. "An Efficient FPGA-Based Convolutional Neural Network for Classification: Ad-MobileNet", *MDPI, Electronics*, 2021.

[32] SUDA, N., CHANDRA, V., DASIKA, G., et al. "Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks", *International Symposium on Field-Programmable Gate Arrays - FPGA '16*, 2016.

[33] TAPIADOR, R., NAVARRO, A. R., BARRANCO, A. L. "Comprehensive Evaluation of OpenCL-Based Convolutional Neural Network Accelerators in Xilinx and Altera FPGAs", *preprint arXiv:1609.09296*, 2016.

[34] SAINI, D., M'DIA, B. "Floating Point Unit Implementation on FPGA", *International Journal of Computational Engineering Research*, 2012.

[35] IEEE. *IEEE Standard for Floating-Point Arithmetic*. Relatório técnico, IEEE Computer Society, 2008.

[36] SZE, V., CHEN, Y.-H., YANG, T.-J., et al. "Efficient Processing of Deep Neural Networks: A Tutorial and Survey", *Proceedings of the IEEE*, 2017.

[37] ZHANG, C., WU, D., SUN, J., et al. "Energy-Efficient CNN Implementation on a Deeply Pipelined FPGA Cluster", *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, 2016.

[38] FARABET, C., MARTINI, B., AKSELROD, P., et al. "Hardware Accelerated Convolutional Neural Networks for Synthetic Vision Systems", *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, 2010.

[39] ARIF, S., LAL, R. K. "Design and Performance Analysis of Various Adder and Multiplier circuits using VHDL", *International Journal of Applied Engineering Research*, 2015.

[40] SAMPAIO, O. A. B. "pythonLearning". Disponível em: <https://github.com/olavosamp/pythonLearning>.

[41] *Cyclone II Device Handbook*. Altera Corporation, 2008.

[42] *DE2-70 Development and Education Board User Manual*. Terasic Technologies, 2009.

[43] LU, L., XIE, J., HUANG, R., et al. "An Efficient Hardware Accelerator for Sparse Convolutional Neural Networks on FPGAs", *IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019.

[44] PEEMEN, M., SETIO, A. A. A., MESMAN, B., et al. "Memory-Centric Accelerator Design for Convolutional Neural Networks", *IEEE 31th International Conference on Computer Design (ICCD)*, 2013.

[45] ALWANI, M., CHEN, H., FERDMAN, M., et al. "Fused-layer CNN accelerators". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, 2016. doi: 10.1109/MICRO.2016.7783725.

[46] LI, N., TAKAKI, S., TOMIOKAY, Y., et al. "A multistage dataflow implementation of a Deep Convolutional Neural Network based on FPGA for high-speed object recognition". In: *2016 IEEE Southwest Symposium on Image Analysis and Interpretation (SSIAI)*, pp. 165–168, 2016. doi: 10.1109/SSIAI.2016.7459201.

[47] QIU, J., WANG, J., YAO, S., et al. "Going Deeper with Embedded FPGA Platform for Convolutional Neural Network". In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pp. 26–35, New York, NY, USA, 2016. Association for Computing Machinery. ISBN: 9781450338561. doi: 10.1145/2847263.2847265.

[48] PARK, J., SUNG, W. "FPGA based implementation of deep neural networks using on-chip memory only". pp. 1011–1015, 03 2016. doi: 10.1109/ICASSP.2016.7471828.

[49] GYSEL, P., MOTAMEDI, M., GHIASI, S. "Hardware-oriented Approximation of Convolutional Neural Networks", 2016. doi: 10.48550/ARXIV.1604.03168.

[50] JANG, S., LIU, W., PARK, S., et al. "Automatic RTL Generation Tool of FPGAs for DNNs", *MDPI, Electronics*, 2022.

[51] KANEDA, N., CHUANG, C.-Y., ZHU, Z., et al. "Fixed-Point Analysis and FPGA Implementation of Deep Neural Network Based Equalizers for High-Speed PON", *Journal of Lightwave Technology*, v. 40, n. 7, 2022.

[52] NICHOLS, K. R., MOUSSA, M. A., AREIBI, S. M. "Feasibility of Floating-Point Arithmetic in FPGA based Artificial Neural Networks", 2002.

[53] SANKARADAS, M., JAKKULA, V., CADAMBI, S., et al. "A Massively Parallel Coprocessor for Convolutional Neural Networks", *20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, pp. 53–60, 2009. doi: 10.1109/ASAP.2009.25.

[54] GUPTA, S., AGRAWAL, A., GOPALAKRISHNAN, K., et al. "Deep Learning with Limited Numerical Precision". 2015.

[55] WU, C., WANG, M., CHU, X., et al. "Low Precision Floating-point Arithmetic for High Performance FPGA-based CNN Acceleration", *ACM, Transactions on Reconfigurable Technology and Systems*, 2022.

[56] IRMAK, H., CORRADI, F., DETTERER, P., et al. "A Dynamic Reconfigurable Architecture for Hybrid Spiking and Convolutional FPGA-Based Neural Network Designs", *Journal of Low Power Electronics and Applications*, v. 11, n. 3, 2021. ISSN: 2079-9268. doi: 10.3390/jlpea11030032.

[57] YANG, G., LEI, J., XIE, W., et al. "Algorithm/Hardware Co-Design for Real-Time On-Satellite CNN based Ship Detection in SAR Imagery", *IEEE Transactions of Geoscience and Remote Sensing*, 2021.

[58] ZHANG, X., WANG, J., ZHU, C., et al. "DNNBuilder: an Automated Tool for Building High-Performance DNN Hardware Accelerators for FPGAs", *International Conference on Computer Aided Design (ICCAD), San Diego*, 2018.

[59] CHAKRADHAR, S., SANKARADAS, M., JAKKULA, V., et al. "A Dynamically Configurable Coprocessor for Convolutional Neural Networks", *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*, 2010.

[60] MINAKOVA, S., STEFANOV, T. "Memory-Throughput Trade-off for CNN-based Applications at the Edge", *ACM Transactions on Design Automation of Electronic Systems*, 2022.

[61] ZHANG, C., LI, P., SUN, G., et al. "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, pp. 161–170, New York, NY, USA, 2015. Association for Computing Machinery. ISBN: 9781450333153. doi: 10.1145/2684746.2689060.

[62] FARABET, C., POULET, C., HAN, J. Y., et al. "CNP: An FPGA-based processor for Convolutional Networks". In: *2009 International Conference on Field Programmable Logic and Applications*, pp. 32–37, 2009. doi: 10.1109/FPL.2009.5272559.

[63] LI, H., FAN, X., JIAO, L., et al. "A high performance FPGA-based accelerator for large-scale convolutional neural networks". pp. 1–9, 08 2016. doi: 10.1109/FPL.2016.7577308.

[64] OBERSTAR, E. L. *Fixed-Point Representation & Fractional Math.* Relatório técnico, University of Wisconsin–Madison, 2007.

[65] GUPTA, M., SINGH, J. "A Comparative Study of Cache Optimization Techniques and Cache Mapping Techniques", *International Journal of Engineering Research & Technology (IJERT)*, v. 6, n. 05, 2017. doi: 10.17577/IJERTV6IS050569.

[66] SHESHAPPA, S. N., RAMAKRISHNAN, K., RAO, G. A. "Enhancing Cache Performance Based on Improved Average Access Time", *International Journal of Scientific and Research Publications. ISSN 2250-3153*, v. 2, n. 11, 2012.

[67] KHAN, A. "Brief Overview of Cache Memory", 2020. Unpublished.

[68] *Improving Real-Time Performance by Utilizing Cache Allocation Technology.* Relatório técnico, Intel, 2015.

[69] PETROT, F., GREINER, A., GOMEZ, P. "On Cache Coherency and Memory Consistency Issues in NoC Based Shared Memory Multiprocessor SoC Architectures". In: *9th EUROMICRO Conference on Digital System Design (DSD'06)*, pp. 53–60. IEEE, 2006. doi: 10.1109/DSD.2006.73.

[70] AKULA, R., JAIN, K., KOTECHA, D. J. "System Performance with varying L1 Instruction and Data Cache Sizes: An Empirical Analysis", *University of Central Florida.* , 2019. doi: 10.48550/arXiv.1911.11642.

[71] ZHAI, S., QIU, C., YANG, Y., et al. "Design of Convolutional Neural Network Based on FPGA", *Journal of Physics: Conference Series*, v. 1168, pp. 062016, 2019. doi: 10.1088/1742-6596/1168/6/062016.

[72] LI, Z., WANG, L., GUO, S., et al. "Laius: An 8-bit fixed-point CNN hardware inference engine". In: *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*, pp. 143–150. IEEE, 2017. doi: 10.1109/ISPA/IUCC.2017.00030.

[73] XIE, Y., MAJOROS, T., ONIGA, S. "FPGA-Based Hardware Accelerator on Portable Equipment for EEG Signal Patterns Recognition", *Electronics, 2022*, v. 11, n. 15, 2022. doi: 10.3390/electronics11152410.

[74] HUYNH, T. V. "FPGA-based Acceleration for Convolutional Neural Networks on PYNQ-Z2", *International Journal of Computing and Digital Systems, 2022,*, v. 11, n. 1, 2022. doi: 10.12785/ijcds/110136.

[75] YIN, Z.-Y., XU, G.-Y., ZHANG, F.-G., et al. "Design and implementation of Convolution Neural Network Unit Based on Zynq Platform", *Journal of Chinese Computer Systems, 2022*, v. 43, n. 231-235, 2022.

[76] LEE, S., LEE, J. "Compressed Learning of Deep Neural Networks for OpenCL-Capable Embedded Systems", *Applied Sciences*, v. 9, n. 8, 2019. doi: 10.3390/app9081669.

[77] PARK, S.-S., CHUNG, K.-S. "Regularized Convolutional Neural Network for Highly Effective Parallel Processing", *Journal of Computing Science and Engineering*, v. 16, n. 2, pp. 105–112, 2022.

[78] PETTERSSON, L. *Convolutional Neural Networks on FPGA and GPU on the Edge: A Comparison.* Tese de Doutorado, Uppsala Universitet, 2020.

[79] LIU, X. *Resource and data optimization for hardware implementation of Deep Neural Networks Targeting FPGA-based Edge Devices.* Tese de Doutorado, University of Illinois, 2018.