Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Duas Abordagens de Redes Neurais Convolucionais em FPGA: Algoritmo Espacial e Algoritmo de Winograd

Artur Hugo Cunha Pereira

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. Marcus Vinicius Lamar

Brasília
2023

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Curso de Engenharia da Computação

Coordenador: Prof. Dr. João Luiz Azevedo de Carvalho

Banca examinadora composta por:

Prof. Dr. Marcus Vinicius Lamar (Orientador) — CIC/UnB
Prof. Dr. Ricardo Pezzuol Jacobi — CIC/UnB
Prof. Dr. Marcelo Grandi Mandelli — CIC/UnB

CIP — Catalogação Internacional na Publicação

Pereira, Artur Hugo Cunha.

Duas Abordagens de Redes Neurais Convolucionais em FPGA: Algoritmo Espacial e Algoritmo de Winograd / Artur Hugo Cunha Pereira. Brasília : UnB, 2023.
56 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2023.

1. Redes Neurais, 2. FPGA, 3. Redes Neurais Convolucionais,
4. Algoritmo de Winograd

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Duas Abordagens de Redes Neurais Convolucionais em FPGA: Algoritmo Espacial e Algoritmo de Winograd

Artur Hugo Cunha Pereira

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. Dr. Marcus Vinicius Lamar (Orientador)
CIC/UnB

Prof. Dr. Ricardo Pezzuol Jacobi
CIC/UnB

Prof. Dr. Marcelo Grandi Mandelli
CIC/UnB

Prof. Dr. João Luiz Azevedo de Carvalho
Coordenador do Curso de Engenharia da Computação

Brasília, 26 de julho de 2023

Dedicatória

Eu dedico esse trabalho aos meus pais, Carmen Lúcia e Nelson José, que me criaram e me fizeram a pessoa que sou hoje. Sem o amor e o suporte deles eu não teria chegado até aqui. Muito obrigado, amo vocês e Deus abençoe.

Agradecimentos

Eu gostaria de agradecer a todos os educadores que tive a honra de conhecer ao longo da minha jornada acadêmica na Universidade de Brasília. Todos tiveram um papel fundamental na minha formação e me inspiraram a chegar até onde cheguei. Em especial, agradeço ao meu orientador Marcus Lamar por ter me guiado durante a reta final do curso. Seu suporte foi indispensável para enfrentar os desafios que encontrei ao longo do projeto final. Quero agradecer também aos meus amigos, que sempre estiveram ao meu lado, para descontrair em momentos de lazer e trocar desabafos em momentos difíceis.

Resumo

O campo de inteligência artificial tem evoluído muito recentemente. Com o advento das Redes Neurais Profundas (do inglês, *Deep Neural Networks*) (DNNs), a demanda por desempenho computacional se torna cada vez maior. Apesar da popularidade do uso de Unidades de Processamento Gráficas (do inglês *Graphics Processing Units*) (GPUs) para aceleração do processamento, outras tecnologias podem apresentar potencial e diferentes vantagens para resolver esse problema. Uma dessas alternativas são os Arranjos de Portas Programáveis em Campo (do inglês *Field-Programmable Gate Arrays*) (FPGAs), que apresentam grande flexibilidade para desenvolvimento de *hardware* dedicado. Este trabalho se propõe a explorar a implementação em FPGA de Redes Neurais Convolucionais (do inglês, *Convolutional Neural Networks*) (CNNs), um dos tipos mais populares de DNNs. Para tal, dois circuitos são propostos: um que se baseia no algoritmo tradicional, ou espacial, de convolução e outro que se baseia no algoritmo de Winograd para convolução. O algoritmo de Winograd visa simplificar o processo de convolução através de transformações lineares que reduzem o número de multiplicações necessárias. Entretanto, há um aumento na complexidade do circuito devido à lógica adicional para realizar as transformações. As duas implementações são comparadas em um estudo de caso de uma arquitetura simples para resolver o problema de classificação de dígitos escritos à mão da base MNIST. A conclusão obtida a partir dessa comparação é de que, embora o circuito de Winograd tenha maiores requisitos espaciais, seu tempo de inferência é consideravelmente menor devido à natureza do algoritmo que permite calcular quatro saídas da convolução paralelamente.

Palavras-chave: Redes Neurais, FPGA, Redes Neurais Convolucionais, Algoritmo de Winograd

Abstract

The field of artificial intelligence has been rapidly evolving in recent times. With the advent of Deep Neural Networks (DNNs), the demand for computational performance is ever-increasing. Even though Graphics Processing Units (GPUs) for accelerating processing are the most popular choice for accelerating DNNs, other technologies can offer potential and different advantages to solve this problem. One of these alternatives is Field-Programmable Gate Arrays (FPGAs), which provide great flexibility for dedicated hardware development. This work aims to explore the implementation of Convolutional Neural Networks (CNNs), one of the most popular types of DNNs, on FPGAs. To this end, two circuits are proposed: one based on the traditional, or spatial, convolution algorithm and another based on the Winograd algorithm for convolution. The Winograd algorithm aims to simplify the convolution process through linear transformations that reduce the number of multiplications required. However, there is an increase in circuit complexity due to the additional logic to perform the transformations. The two implementations are compared in a case study of a simple architecture to solve the handwritten digits classification problem in the MNIST dataset. The conclusion drawn from this comparison is that, although the Winograd circuit has higher spatial requirements, its inference time is considerably lower due to the nature of the algorithm, which allows for the parallel calculation of four convolution outputs.

Keywords: Neural Networks, FPGA, Convolutional Neural Networks, Winograd's Algorithm

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

ALMs Módulos Lógicos Adaptativos (do inglês *Adaptive Logic Modules*).

ALUTs Tabelas de Pesquisa Adaptativas (do inglês *Adaptive Lookup Tables*).

ANN Rede Neural Artificial (do inglês, *Artificial Neural Network*).

ANNs Redes Neurais Artificiais (do inglês, *Artificial Neural Networks*).

ASIC Circuito Integrado de Aplicação Específica (do inglês *Application Specific Integrated Circuit*).

CNN Rede Neural Convolucional (do inglês, *Convolutional Neural Network*).

CNNs Redes Neurais Convolucionais (do inglês, *Convolutional Neural Networks*).

DL Aprendizado Profundo (do inglês *Deep Learning*).

DNNs Redes Neurais Profundas (do inglês, *Deep Neural Networks*).

DSPs Processadores de Sinal Digital (do inglês *Digital Signal Processors*).

FC Totalmente Conectada (do inglês, *Fully Connected*).

FPGA Arranjo de Portas Programável em Campo (do inglês *Field-Programmable Gate Array*).

FPGAs Arranjos de Portas Programáveis em Campo (do inglês *Field-Programmable Gate Arrays*).

GPU Unidade de Processamento Gráfica (do inglês *Graphics Processing Unit*).

GPUs Unidades de Processamento Gráficas (do inglês *Graphics Processing Units*).

HDL Linguagem de Descrição de *Hardware* (do inglês *Hardware Description Language*).

HLS Síntese de Alto Nível (do inglês *High Level Synthesis*).

IA Inteligência Artificial.

MIF Arquivo de Inicialização de Memória (do inglês *Memory Initialization File*).

MIFs Arquivos de Inicialização de Memória (do inglês *Memory Initialization Files*).

ML Aprendizado de Máquina (do inglês *Machine Learning*).

MLP Perceptron de Múltiplas Camadas (do inglês *Multi-Layer Perceptron*).

MNIST Banco de dados da Organização Nacional de Benchmarks e Inovação Alterado (do inglês *Changed National Organization of Benchmarks and Innovation database*).

RAM Memória de Acesso Aleatório (do inglês *Random Access Memory*).

ReLU Unidade Retificadora Linear (do inglês *Rectifying Linear Unit*).

RGB Vermelho, Verde, Azul (do inglês *Red, Green, Blue*).

RNN Redes Neural Recorrente (do inglês, *Recurrent Neural Network*).

Capítulo 1

Introdução

A Inteligência Artificial (IA) é um campo de pesquisa que tem impactado vastamente a sociedade moderna. Aplicações habilitadas por IA estão presentes em diversos dispositivos e contextos, desde grandes servidores processando *Big Data* aos aparelhos celulares que as pessoas levam em seus bolsos.

Uma das vertentes da IA mais utilizada atualmente é a do Aprendizado de Máquina (do inglês *Machine Learning*) (ML), que tem como princípio a definição de modelos capazes de aprender a identificar padrões a partir de um conjunto de dados de treinamento. Conforme a complexidade dos padrões aumentam, modelos mais robustos são necessários. Atualmente, esses modelos robustos são as Redes Neurais Profundas (do inglês, *Deep Neural Networks*) (DNNs), que compõem a subárea de Aprendizado Profundo (do inglês *Deep Learning*) (DL).

As DNNs tem como precursoras as Redes Neurais Artificiais (do inglês, *Artificial Neural Networks*) (ANNs), um tipo de modelo inspirado no sistema nervoso humano. Uma das primeiras implementações de ANN foi o Perceptron [1], que seria o modelo matemático análogo a um neurônio real, que recebe estímulos de entrada, processa esses estímulos e os combina em um saída. Com o tempo, essa ideia foi evoluindo a partir de arquiteturas como o Perceptron de Múltiplas Camadas (do inglês *Multi-Layer Perceptron*) (MLP), usando cada vez mais neurônios e encadeando múltiplas camadas. Eventualmente, arquiteturas mais elaboradas surgiram para resolver problemas mais específicos. Hoje em dia, o estado da arte para tarefas de visão computacionais são modelos de Redes Neurais Convolucionais (do inglês, *Convolutional Neural Networks*) (CNNs). Porém, a eficácia desses modelos avançados vem com um elevado custo computacional.

1.1 Justificativa

Redes Neurais Convolucionais (do inglês, *Convolutional Neural Networks*) (CNNs) são uma classe de redes neurais artificiais profundas que é projetada para preservar a relação espacial entre suas entradas. Dessa forma, elas são bastante adequadas e eficazes para o processamento e análise de imagens em tarefas de visão computacional. Porém, como modelos de aprendizado profundo, CNNs apresentam um custo computacional elevado. Isso torna interessante, se não necessário em alguns casos, a utilização de *hardware* dedicado para acelerar o processamento dos modelos.

Nos dias de hoje, a abordagem mais amplamente adotada para aceleração de CNNs é o uso de Unidades de Processamento Gráficas (do inglês *Graphics Processing Units*) (GPUs). Porém, GPUs apresentam um custo elevado em termos de consumo de energia, tornando a solução não ideal para cenários mais limitados como em dispositivos embarcados. Outras tecnologias que podem ser exploradas para aceleração de CNNs são as de Arranjo de Portas Programável em Campo (do inglês *Field-Programmable Gate Array*) (FPGA) e de Circuito Integrado de Aplicação Específica (do inglês *Application Specific Integrated Circuit*) (ASIC). A implementação de CNNs em FPGA tem sido estudada com certa constância em tempos recentes, devido à maior flexibilidade que essa abordagem fornece graças à facilidade de reconfigurar o circuito projetado. A fim de aproveitar essa flexibilidade, esse trabalho visa explorar técnicas de otimização. Dentre as técnicas utilizadas, destaca-se a implementação do circuito de convolução baseado no algoritmo de Winograd. Para ter uma referência de comparação, também foi implementado um circuito baseado na convolução espacial. Com esse contexto em mente, a hipótese de pesquisa aqui apresentada é de que é possível acelerar o processamento da inferência de CNNs em FPGA com o uso da convolução de Winograd.

1.2 Objetivos

O objetivo desse projeto é verificar a possibilidade de implementar uma CNN em FPGA e comparar o desempenho de duas abordagens adotadas: uma baseada no algoritmo convecional, ou espacial, de convolução e outra baseada no algoritmo de Winograd. O escopo da implementação proposta foi limitado apenas ao processo de inferência de uma rede pré-treinada. Para avaliar o desempenho do circuito projetado, ele é testado sobre um conjunto de dados adequado ao modelo implementado. Os objetivos específicos do projeto podem ser elencados em:

1. Implementar módulos em *SystemVerilog* que possam ser combinados para implementar arquiteturas de *CNNs*;

2. Implementar uma arquitetura simples como prova de conceito;
3. Testar o circuito na mesma base de dados usada para avaliar o modelo implementado em *software*;
4. Implementar o circuito fisicamente em uma placa de desenvolvimento e extrair métricas de desempenho;
5. Analisar e comparar resultados obtidos com os resultados de referência.

1.3 Organização da Monografia

Nesse capítulo introdutório, foi apresentado um breve panorama do contexto de IA e CNNs, tal como uma justificativa para a implementação de redes convolucionais em FPGA. Em seguida foram elencados os objetivos do projeto desenvolvido a fim de verificar a hipótese de pesquisa.

Os demais capítulos da monografia estão organizado da seguinte forma: no Capítulo 2, é apresentada a fundamentação teórica sobre CNNs, algoritmos de convolução rápida e técnicas otimização de implementações em *hardware*; no Capítulo 3, é apresentada a metodologia proposta para implementação do circuito de inferência em FPGA; no Capítulo 4, são apresentados os resultados obtidos a partir dos experimentos realizados com uma implementação de prova de conceito; por fim, o Capítulo 5 finaliza a obra sumarizando a análise dos resultados, apresentando considerações finais e possíveis trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Neste capítulo, são apresentados os fundamentos teóricos necessários para para o projeto. Na Seção 2.1, são explicados os fundamentos de CNNs, cada parte sendo abordada em maior detalhe em suas respectivas subseções. A Seção 2.2 apresenta a base para entender a aplicação do algoritmo de Winograd a convoluções e, por fim, a Seção 2.3 discorre sobre a literatura relacionada ao trabalho apresentado.

2.1 Redes Neurais Convolucionais

A primeira arquitetura precursora das CNNs modernas foi a rede Neocognitron proposta por Fukushima em 1980 [2], inspirado na estrutura hierárquica do sistema visual humano. A estrutura do modelo consiste de camadas encadeadas que usam filtros deslizantes para processar suas entradas bidimensionais. Essas camadas eram intercaladas com módulos que realizavam uma subamostragem dos dados filtrados. Por fim, a saída da última subamostragem é conectada a 10 neurônios de forma que, após o treinamento não supervisionado do modelo, cada neurônio responda a apenas um padrão de estímulo.

Algum tempo depois, LeCun e outros [3] apresentam uma nova arquitetura, que ainda usa as ideias fundamentais propostas por Fukushima, aplicada a um problema prático de reconhecimento de dígitos escritos à mão. Um dos grandes diferenciais, porém, foi a proposta de utilizar aprendizado supervisionado (ao invés de não supervisionado) para obter os valores, ou pesos, dos filtros usados em cada camada convolucional. Esse veio a ser um dos principais paradigmas para o treinamento de CNNs.

Embora as CNNs tenham evoluído bastante desde esses trabalhos seminais, a estrutura básica de uma rede para classificação de imagens continua a mesma: camadas convolucionais, função de ativação, subamostragem e uma camada Totalmente Conectada (do inglês, *Fully Connected*) (FC). A seguir, cada um desses blocos de base são explicados em maior detalhe.

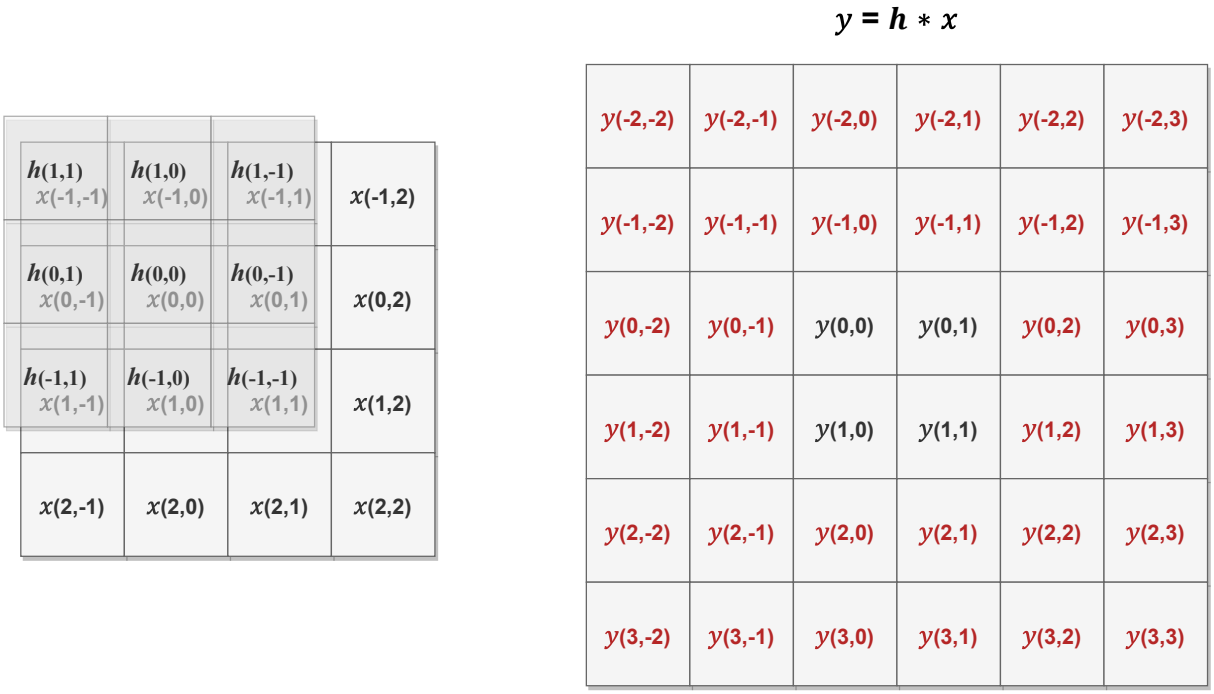


Figura 2.1: Visualização de convolução entre filtro 3×3 e imagem 4×4. Os valores em vermelho destacam os resultados da convolução em que valores do filtro foram multiplicados por valores que não foram definidos pela imagem.

Fonte: próprio autor

2.1.1 Convolução

A operação de convolução discreta em duas dimensões [4] pode ser expressa por

$$y[n, m] = (x * h)[n, m] = \sum_i \sum_j x[n - i, m - j] \cdot h[i, j], \tag{2.1}$$

onde x representa o sinal bidimensional que se deseja convolver com o filtro h.

Dessa forma, computar todos os valores de y seria análogo a deslizar o filtro h espelhado sobre a entrada x. A Figura 2.1 ilustra um exemplo de como pode ser visualizada a convolução de uma imagem 4×4 e um filtro 3×3. É importante observar que, nesse caso, a imagem de entrada x está definida apenas no intervalo $([-1, 2], [-1, 2])$. Para obter os valores destacados em vermelhos, seria necessário extrapolar os valores da imagem além de suas bordas. Isso pode ser obtido com o preenchimento de valores (do inglês *padding*). Entretanto, em muitos casos, como em CNNs, esses valores são ignorados e trabalha-se com a dimensão reduzida da saída (no caso do exemplo, 2×2).

Além disso, na prática, muitas implementações modernas não espelham o filtro antes de fazer a operação, caracterizando assim uma correlação-cruzada e não uma convolução. Isso pode ser feito, pois os pesos dos filtros são aprendidos. Ou seja, ao se treinar uma CNN

usando a operação de correlação-cruzada e outra usando convolução, os filtros aprendidos por uma seriam os filtros espelhados da outra.

Assim, considerando que a entrada x_l de uma camada convolucional $l+1$ possui K_l canais, o canal k_{l+1} da saída y_{l+1} pode ser computado de acordo com

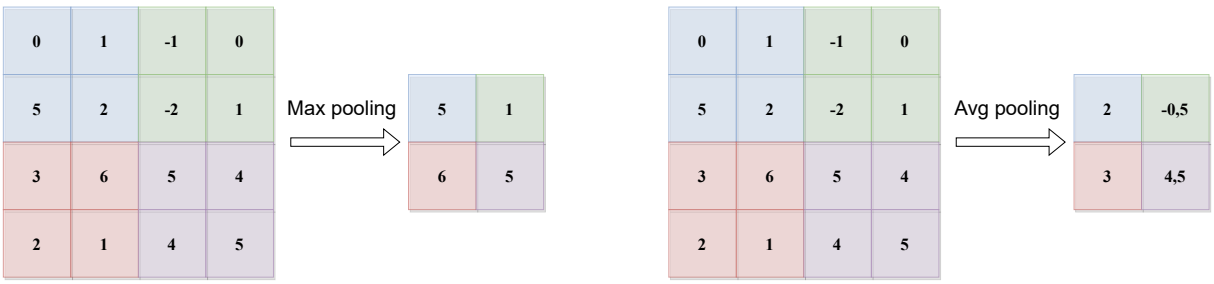
$$y_{l+1}[n, m, k_{l+1}] = \sum_{k_l=0}^{K_l-1} \sum_{i=0}^{N_{l+1}-1} \sum_{j=0}^{M_{l+1}-1} x_l[n+i, m+j, k_l] \cdot h_{k_{l+1}}(i, j, k_l), \tag{2.2}$$

onde o filtro $h_{k_{l+1}}$ tem dimensões $N_{l+1} \times M_{l+1}$.

2.1.2 Subamostragem

A subamostragem é uma etapa da inferência que acontece após a convolução da entrada de uma camada. Ao invés da saída de uma camada convolucional ser simplesmente o resultado da convolução, algum critério é utilizado para reduzir a dimensionalidade da saída. Esse critério geralmente é aplicado sobre regiões menores da imagem, chamados de agrupamentos (do inglês *pools*), e resulta em valores únicos.

A mesma analogia de uma janela deslizante pode ser usada para compreender o processo de subamostragem. Na Figura 2.2, são ilustrados exemplos de subamostragem em agrupamentos de dimensões 2×2 com o passo da janela de agrupamento igual a 2. Na Figura 2.2a, é exemplificado o tipo de subamostragem mais comum em CNNs, a subamostragem máxima (do inglês *max pooling*). A Figura 2.2b exemplifica a subamostragem média (do inglês *average pooling*).



(a) Subamostragem máxima.

(b) Subamostragem média.

Figura 2.2: Ilustrações de subamostragens com agrupamentos 2×2 e passo 2. Os resultados de uma cor correspondem à aplicação do critério de subamostragem no agrupamento da mesma cor.

Fonte: próprio autor

O uso de subamostragem é importante em CNNs, pois auxilia na invariância espacial do modelo ao agrupar informações próximas em uma representação única e de dimensionalidade menor.

2.1.3 Camada Totalmente Conectada

A camada Totalmente Conectada (do inglês, *Fully Connected*) (FC) é composta por neurônios artificiais [5]. A Figura 2.3 apresenta a estrutura de um neurônio artificial.

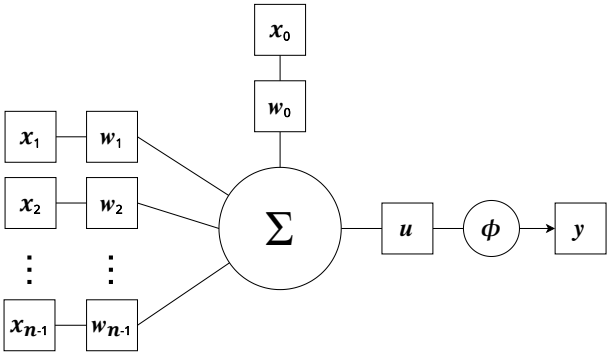


Figura 2.3: Representação do modelo de neurônio utilizado em camadas FC.

Fonte: próprio autor

O neurônio recebe vários sinais de entrada x_i da camada anterior, multiplica-os por seus respectivos pesos w_i, acumula seus valores no potencial de ativação u dado por

$$u(x_1, x_2, ..., x_i) = \sum_{n=0}^{i} w_n x_n \tag{2.3}$$

e, por fim, passa o potencial pela função de ativação ϕ, obtendo a saída $y = \phi(u)$. O peso w_0 é chamado de viés (do inglês *bias*) e a entrada x_0 é sempre unitária.

2.1.4 Funções de Ativação

As funções de ativação são outro componente importante na arquitetura de ANNs. Elas servem ao propósito de introduzir não linearidades na saída de uma camada da rede. Caso não houvesse essa não linearidade a rede neural seria uma simples combinação linear das entradas e pesos, perdendo muito seu poder de generalização. Embora diversas funções satisfaçam esse requisito de não linearidade, uma das mais utilizada em camadas convolucionais é a Unidade Retificadora Linear (do inglês *Rectifying Linear Unit*) (ReLU), pois proporciona menor tempo de convergência para a rede no processo de treinamento [6]. Essa função é dada por

$$\text{ReLU}(u) = \max(u, 0)\,. \tag{2.4}$$

Outras funções comumente encontradas em ANNs são a tangente hiperbólica, a *sigmoid* e a *softmax*, mostradas respectivamente nas Equações 2.5 a 2.7.

$$\tanh(u) = \frac{e^u - e^{-u}}{e^u + e^{-u}}\,, \tag{2.5}$$

$$\text{sigmoid}(u) = \frac{1}{1 + e^{-u}}\,, \tag{2.6}$$

$$\text{softmax}(\vec{u})_i = \frac{e^{u_i}}{\sum_{j=1}^{|K|} e^{u_j}}\,. \tag{2.7}$$

É importante observar que na Equação 2.7 $\vec{u}$ é um vetor. A função softmax geralmente é utilizada para problemas de classificação com múltiplas classes devido ao fato de gerar uma distribuição de probabilidade da classificação [7]. Na camada FC de CNNs, o vetor $(\vec{u})$ é composto pelas K saídas u_i de cada neurônio i. Essa função transforma as saídas u_i, em probabilidades, de forma que $\sum_i \text{softmax}(u)_i = 1$.

2.2 Algoritmo de Winograd

Com o panorama previamente apresentado sobre o processo de convolução, é possível notar que, seguindo a abordagem tradicional expressa pela Equação 2.1, é realizado um número de multiplicações igual ao número de coeficientes do filtro para cada valor de saída (considerando-se um único canal de entrada). Para uma convolução unidimensional, pode-se denotar esse processo de obter m saídas de uma convolução com um filtro de r coeficientes como $F(m, r)$. Assim, tem-se que o algoritmo tradicional, ou espacial, da convolução usa $m \cdot r$ multiplicações.

Existem, porém formas de computar $F(m, r)$ de maneira mais eficiente. Uma dessas formas é o algoritmo de Winograd, proposto em [8] e revisitado em [9] como alternativa para aceleração de CNNs. O algoritmo consiste em realizar uma transformação linear sobre os dados de entrada e sobre os coeficientes do filtro de forma que apenas uma multiplicação precise ser realizada por elemento da entrada, isto é, $m+r-1$ multiplicações. Esse resultado pode então ser transformado de volta para o domínio original.

A Equação 2.8 sumariza esse processo, sendo d os dados de entrada, g o filtro, B^T e G suas respectivas transformações e A^T a transformação inversa para obter o resultado final Y no domínio espacial.

$$Y = A^T[(Gg) \odot (B^T d)]\,. \tag{2.8}$$

onde $\odot$ corresponde ao produto de Hadamard, isto é multiplicação elemento a elemento. Em seu trabalho, Winograd documenta o exemplo a seguir para $F(2, 3)$.

$$F(2,3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix} \tag{2.9}$$

onde
$m_1 = (d_0 - d_2)g_0$,
$m_2 = (d_1 + d_2)\frac{g_0+g_1+g_2}{2}$,
$m_4 = (d_1 - d_3)g_2$ e
$m_3 = (d_2 - d_1)\frac{g_0-g_1+g_2}{2}$.

Note que as únicas multiplicações por valores não constantes necessárias são as feitas para obter os valores m_i. Dessa forma, como esperado, tem-se $m + r - 1 = 4$ multiplicações, ao invés de 6.

O algoritmo para convolução em uma dimensão, apresentado na Equação 2.8, pode ser aninhado em si mesmo para realizar uma convolução bidimensional $F(m \times m, r \times r)$ de acordo com a expressão:

$$Y = A^T[(GgG^T) \odot (B^T dB)]A\,. \tag{2.10}$$

No caso de $F(2 \times 2, 3 \times 3)$, as matrizes das transformações necessárias são dadas por

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}, \tag{2.11}$$

$$G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix} \tag{2.12}$$

e

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}. \tag{2.13}$$

2.3 Artigos Relacionados

A aceleração de ANNs em FPGA é uma área que vem sendo pesquisada há um tempo. A natureza reconfigurável dos FPGAs é um atrativo para o projeto de *hardware* dedicado

a um tipo de tarefa específico. Antes da popularização da aceleração via GPU, pesquisadores exploravam FPGAs como uma plataforma para otimização do processamento de ANNs [10], [11], [12], [13]. Em 2009, Raina e outros introduzem o uso de GPUs para o treinamento de modelos de aprendizado não supervisionado [14]. Logo em seguida, em 2010, Cireşan e outros aplicam o uso de GPUs para o problema de classificação de dígitos da base de dados Banco de dados da Organização Nacional de Benchmarks e Inovação Alterado (do inglês *Changed National Organization of Benchmarks and Innovation database*) (MNIST) [15]. Finalmente, a aceleração via GPU é popularizada por Krizhevsky e outros em seu artigo [6] publicado em 2012. Desde então, GPUs têm sido a tecnologia dominante para aceleração de DNNs.

Porém, a alternativa da aceleração em FPGA ainda tem seus méritos e continua sendo pesquisada em dias mais recentes. Em 2021, Yang e outros [16] apresentam uma implementação para DL por reforço. Em 2022, Jovanović e outros [17] fazem uma revisão da literatura acerca da implementação de mapas auto-organizáveis, técnica de aprendizado não supervisionado, enquanto Wang e outros [18] apresentam uma implementação de uma arquitetura de Redes Neural Recorrente (do inglês, *Recurrent Neural Network*) (RNN).

A revisão de literatura [19] de 2021 feita por Wu e outros elenca as diferentes tecnologias que podem ser usadas para aceleração de inferência de ANNs, discute as vantagens e desvantagens de cada uma e discorre sobre diferentes técnicas de otimização. Em sua discussão, um ponto que destaca o potencial de FPGA como plataforma de aceleração é a flexibilidade da tecnologia, o que promove maior compatibilidade com diferentes técnicas de otimização.

Tratando-se mais especificamente de CNNs, em 2017, Song e outros apresentam um acelerador de convolução baseado em FPGA [20]. Em 2018, Venieris e outros propõem um fluxo automatizado para o mapeamento de várias CNNs em uma FPGA [21]. Em 2020, Bai e outros desenvolvem uma arquitetura unificada para convolução e deconvolução em *hardware* [22]. A revisão de literatura [23], publicada em 2018 por Shawahna e outros, investiga a tendências de aceleradores de CNN em FPGA. Diversos trabalhos são listados nesses artigos e é apresentada uma relação entre cada trabalho e as técnicas utilizadas. Algumas das técnicas mais populares são também utilizadas nesse projeto, como a precisão de ponto fixo, o *buffer* de linha e o desenrolamento de laço (do inglês *loop unrolling*). Embora menos utilizado, o algoritmo de Winograd é implementado por dois dos trabalhos apresentados, [24] e [25]. Uma implementação proposta na mesma época que também utiliza o algoritmo de Winograd é a PipeCNN proposta por Wang e outros em [26]. Algumas implementações publicadas mais recentemente, [27] em 2020 e [28] em 2022, também se baseiam no algoritmo de Winograd, mostrando que essa técnica ainda possui aplicabilidade em *hardware* moderno.

Alguns estudos mais recentes indicam que a aceleração em FPGA ainda tem sua relevância e apresenta potencial. Em 2020, Li e outros [29] apresentam uma nova implementação para processamento em tempo real e baixo consumo de energia, enquanto Dinelli e outros [30] apresentam um acelerador que não faz uso de recursos externos ao *chip* da FPGA e analisam suas vantagens e desvantagens. El-Maksoud e outros [31] apresentam um estudo de caso de sua implementação de alta eficiência energética. Outra implementação focada no consumo de energia é apresentada em 2021 por Sharma e outros [32]. Dois trabalhos de 2022 trazem as propostas de processamento em tempo real [33] e flexibilidade em relação às dimensões da convolução [34]. Em 2023, Bai e outros publicam o artigo [35], apresentando acelerador em FPGA aplicado ao caso de uso de computação em borda. No mesmo ano, Cruz e outros [36] propõem uma abordagem mista de *hardware* e *software* para acelerar a inferência de modelos programados com a biblioteca TensorFlow Lite, que usa parâmetros de 8 *bits*. E, por fim, Srilakshmi e outros [37] apresentam um estudo comparativo entre o uso de Síntese de Alto Nível (do inglês *High Level Synthesis*) (HLS) e o uso de Linguagem de Descrição de *Hardware* (do inglês *Hardware Description Language*) (HDL) para projetar aceleradores em FPGA.

Com esse panorama da literatura em mente, o capítulo seguinte apresenta a metodologia proposta nesse trabalho, passando pela visão geral do sistema desenvolvido e entrando em maior detalhe sobre cada aspecto e módulo da implementação.

Capítulo 3

Metodologia Proposta

Este capítulo apresenta a metodologia proposta para o desenvolvimento organizada da seguinte forma: a Seção 3.1 apresenta uma visão geral do sistema desenvolvido e uma estrutura em alto nível do circuito projetado; a Seção 3.2 trata da arimética de ponto fixo utilizada; a Seção 3.3 discorre sobre os módulos comuns de leitura de pesos e a organização interna das memórias de pesos; a Seção 3.4 trata dos módulos específicos do circuito de convolução, começando pelos módulos comuns às duas abordagens e entrando em específicos do algoritmo espacial e do algoritmo de Winograd na Subseção 3.4.1 e na Subseção 3.4.2 respectivamente; e, por fim, a Seção 3.5 trata da implementação do módulo comum de camada FC.

3.1 Visão Geral

A Figura 3.1 apresenta o fluxo seguido para geração do circuito proposto e obtenção de suas saídas.

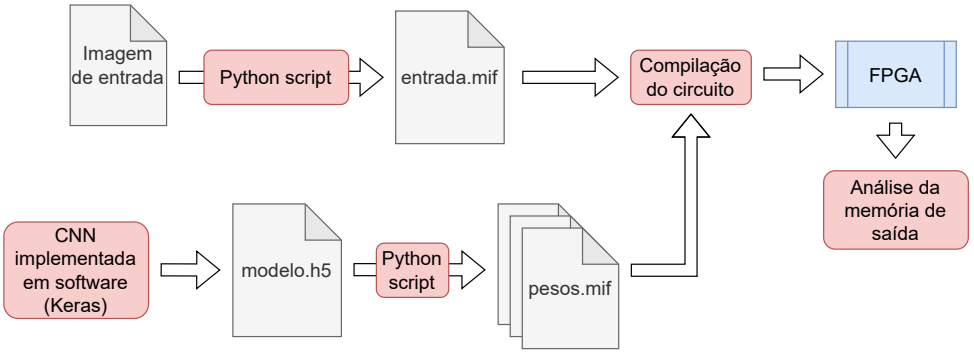


Figura 3.1: Fluxo do sistema proposto. Em vermelho são destacados processos, em azul é destacado o dispositivo de teste do circuito gerado. Os demais blocos são arquivos intermediários usados na compilação do circuito.

Fonte: próprio autor

Esse processo tem início com a implementação e o treinamento de um modelo de CNN em *software*. Essa etapa foi feita usando a linguagem Python com a biblioteca Keras [38]. Essa biblioteca permite salvar o modelo em um arquivo de formato HDF5 [39] a partir do qual são gerados os Arquivos de Inicialização de Memória (do inglês *Memory Initialization Files*) (MIFs) para cada camada que será implementada em *hardware*. De forma análoga, a imagem de entrada que se deseja processar também é convertida em arquivo MIF e fornecida para a compilação do circuito. Uma vez compilado, o circuito é carregado no FPGA. Por fim, os resultados obtidos são escritos em uma memória de saída, que pode ser analisada depois do tempo necessário para o circuito realizar a inferência.

Para cada camada implementada no circuito, é instanciado um módulo de memória interna para armazenar seus pesos. A imagem de entrada do circuito também é carregada em um módulo de memória e, para avaliar o resultado, os valores dos neurônios da camada FC também são escritos em uma memória de saída. A Figura 3.2 mostra o esquemático de como fica a visão geral do circuito desenvolvido, incluindo a relação entre camadas e memórias de peso e usando como exemplo uma arquitetura com duas camadas convolucionais.

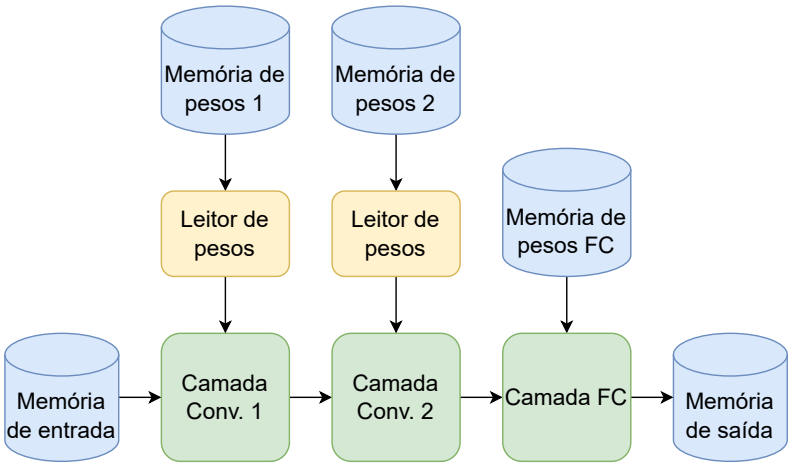


Figura 3.2: Diagrama da estrutura geral do circuito usando duas camadas convolucionais para exemplificar. Em azul são representados os elementos de memória interna, em amarelo, as interfaces entre camada convolucional e memória e em verde, os circuitos responsáveis pelo processamento de cada camada.

Fonte: próprio autor

A interface entre memória e camada convolucional é feita por um módulo leitor de pesos, explicado em maior detalhe na Seção 3.3. A camada FC acessa diretamente a memória, pois não é necessária lógica adicional para estruturar os pesos em canais de filtros. Essa camada é explicada em detalhe na Seção 3.5.

3.2 Representação Numérica

As operações básicas para a implementar uma convolução são a soma e a multiplicação de valores. Deste modo, a primeira decisão do projeto foi relativa à representação numérica utilizada: a representação em ponto fixo. Como mostrado no artigo de revisão de literatura [23], a representação em ponto fixo é amplamente utilizada na aceleração de CNNs em *hardware*. Nessa representação, uma palavra de N bits possui M bits que representam valores fracionários, conforme ilustrado na Figura 3.3 com o exemplo de uma palavra de 8 bits com 4 bits fracionários. Nesse caso, diz-se que essa é uma representação de 8 bits Q4.

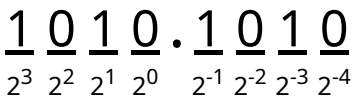


Figura 3.3: Representação de um número de 8 bits em ponto fixo Q4 (quatro bits fracionários representando expoentes negativos).

Fonte: próprio autor

Uma das vantagens dessa representação é o fato de que suas operações aritméticas são implementadas da mesma forma que operações com números inteiros. Dessa forma, o único cuidado necessário foi com a multiplicação, pois a multiplicação de duas palavras de N bits resulta em uma palavra de $2N$ bits. Esse resultado possui o dobro de bits inteiros e o dobro de bits fracionários. Assim, caso deseje-se manter o tamanho da palavra, é necessário descartar a metade mais significativa dos bits inteiros e a metade menos significativa dos bits fracionários. A Figura 3.4 exemplifica essa situação com um resultado de 16 bits Q8.

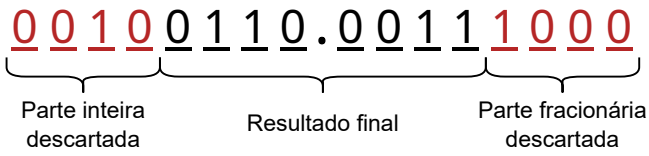


Figura 3.4: Resultado de uma multiplicação de dois números de 8 bits Q4 resultando em um número de 16 bits Q8. Os bits destacados em vermelho seriam descartados caso deseje-se manter o resultado em 8 bits Q4.

Fonte: próprio autor

É importante notar que, na Figura 3.4, existem bits não nulos nas partes descartadas. No caso de bits fracionários descartados, há uma perda de precisão, que pode ser mitigada realizando-se um arredondamento para o número mais próximo dentro da faixa

de valores da representação desejada. Isso pode ser feito somando-se ao resultado final o mais significativo dos bits fracionários descartados. Já no caso de bits não nulos na parte inteira descartada, o único caso em que isso não representa um *overflow* é quando todos os bits são 1 ou 0, de acordo com o bit mais significativo do resultado final (considerando representação em complemento a 2). Nos demais casos, o resultado da multiplicação dos dois números não está na faixa de valores da representação desejada sendo impossível representá-lo. Com isso em mente, um módulo de multiplicação de números em ponto fixo foi implementado para tratar desses casos de arredondamento e sinalização de *overflow*.

3.3 Organização de Memória

Uma das operações mais custosas na implementação de CNNs em FPGA é a leitura dos pesos da memória. As arquiteturas mais básicas de CNN possuem um número de pesos que torna impraticável o armazenamento simultâneo de todos esses valores em registradores. Dessa forma, todos os pesos precisam ser lidos múltiplas vezes. Além de técnicas de reutilização de dados para diminuir o número total de leituras necessárias, uma preocupação válida é minimizar a latência da operação de leitura em si.

Este projeto optou pela utilização de blocos de memória internos ao FPGA. Essa decisão vem com o compromisso de que a memória interna geralmente tem uma capacidade bem menor que dispositivos externos, o que limita altamente a complexidade da arquitetura de CNN implementada. Porém, essa escolha permite criar diversos módulos de memória interna dedicados ao armazenamento dos pesos de uma camada específica, possivelmente favorecendo o paralelismo entre camadas.

Para controlar a leitura dos pesos de uma memória, foi desenvolvido um módulo leitor com interface ilustrada na Figura 3.5.

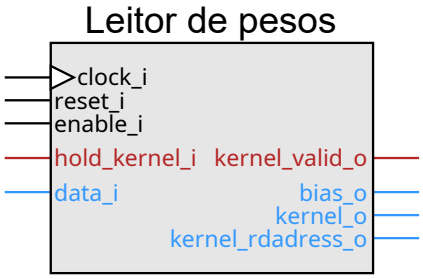


Figura 3.5: Interface do módulo de leitura de pesos. Portas de entrada à esquerda e portas de saída à direita. Portas destacadas em azul representam o caminho de dados, enquanto as em vermelho são sinais de controle.

Fonte: próprio autor

Esse módulo recebe um sinal de controle hold_kernel_i para cada canal de entrada do módulo de convolução da camada correspondente. Esse sinal indica que o leitor deve segurar os valores do canal correspondente de kernel_o. Esse sinal de saída é um arranjo de portas bidimensional de forma que cada kernel_o[i] é um arranjo de portas com os valores dos pesos para o i-ésimo canal do filtro. O sinal de controle kernel_valid_o existe para cada canal do filtro e indica se aquele canal está válido para ser utilizado pelo núcleo de convolução. O sinal bias_o é o peso de *bias* de um filtro, que o leitor atualiza quando começa a carregar os valores do primeiro canal desse filtro.

A porta de entrada data_i recebe os valores lidos da memória para que sejam colocados na porta de saída correspondente. O módulo de leitura utiliza o sinal kernel_rdaddress_o para indicar qual o endereço do valor a ser lido.

Como o leitor de pesos controla o endereço de leitura, é importante que a memória de pesos esteja estruturada de acordo com a ordem em que o leitor espera encontrar os pesos. A lógica desenvolvida para o leitor de pesos requer que a memória esteja organizada em blocos adjacentes do seguinte formato: valor do *bias*, valores da primeira linha do primeiro canal, segunda linha do primeiro canal e assim por diante.

3.4 Convolução

Para implementar as camadas convolucionais, um módulo de núcleo convolucional foi projetado. As implementações dos núcleos de convolução espacial possuem algumas características comuns que serão tratadas a seguir. Os pontos em que as duas implementações divergem são detalhados nas Subseções 3.4.1 e 3.4.2 respectivamente.

Um núcleo convolucional possui um conjunto de portas de dados e um conjunto de portas de controle. Essa interface é ilustrada na Figura 3.6.

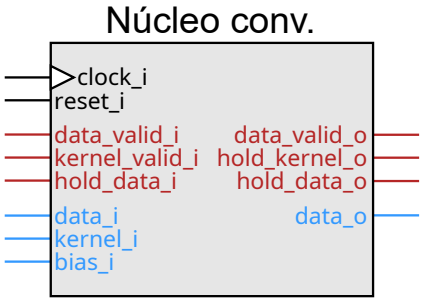


Figura 3.6: Interface do módulo de núcleo de convolução. Portas de entrada à esquerda e portas de saída à direita. Portas destacadas em azul representam o caminho de dados, enquanto as em vermelho são sinais de controle.

Fonte: próprio autor

À esquerda, estão as portas de entrada e, à direita, as portas de saída. As portas destacadas em azul são as portas de dados enquanto as destacadas em vermelho são as portas de controle da convolução e sincronização de camadas. As demais portas são de controle geral, sendo elas o sinal de clock_i de entrada, o sinal de reset_i para voltar o circuito a um estado inicial e a saída que indica ocorrência de overflow_o em alguma multiplicação.

Todos os sinais do caminho de dados são palavras de tamanho parametrizável representadas em ponto fixo. Para cada canal de entrada da camada, existe uma porta data_i para receber os dados de entrada. O arranjo de portas kernel_i recebe simultaneamente todos os canais do filtro fornecido pelo leitor de pesos. A porta data_o existe para cada canal de saída da camada, ou seja, uma porta de saída para cada filtro. Dessa forma, o módulo é parametrizável por número de canais de entrada e número de filtros que geram dados para diferentes canais de saída.

Os sinais de controle de entrada data_valid_i e kernel_valid_i existem para cada canal de entrada. Um sinal data_valid_i de um canal indica se o valor na porta de entrada data_i desse canal é válido e deve ser processado. De maneira análoga, o kernel_valid_i de um canal de entrada indica se todos os valores do arranjo kernel_i do canal estão válidos, além do valor de bias_i que é compartilhado por todos os canais de um mesmo filtro. Já o sinal de entrada hold_data_i existe para cada filtro e indica quando a camada seguinte está solicitando que o processamento do canal de saída correspondente seja interrompido para que ela possa concluir seu processamento necessário com o valor atual.

Os sinais de controle de saída data_valid_o e hold_data_o são conectados às portas de entrada correspondentes do núcleo da camada seguinte e do núcleo da camada anterior respectivamente, conforme ilustrado na Figura 3.7. Já o sinal hold_kernel_o do núcleo de uma camada é conectado ao sinal hold_kernel_i do módulo de leitura de pesos responsável pelos filtros da mesma camada.

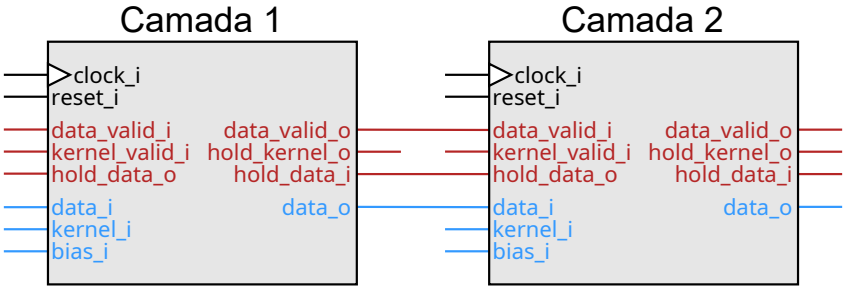


Figura 3.7: Exemplo de encadeamento de núcleos convolucionais responsáveis por duas camadas adjacentes.

Fonte: próprio autor

O módulo de leitura de pesos é responsável por fazer a interface entre o núcleo convolucional e a memória de pesos. A Figura 3.8 mostra como se dá a conexão entre núcleo convolucional e o leitor de pesos.

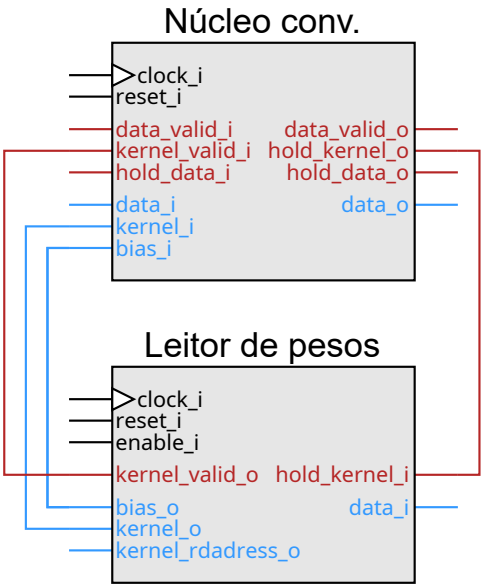


Figura 3.8: Exemplo de encadeamento de núcleos convolucionais responsáveis por duas camadas adjacentes.

Fonte: próprio autor

O leitor de pesos é responsável por controlar qual endereço da memória será lido através de seu sinal de saída kernel_rdaddress_o. Com isso, a memória de pesos alimenta o leitor com o valor presente naquele endereço através da porta de entrada data_i do leitor. Conforme o leitor recebe os valores da memória, esses valores são escritos na porta adequada, bias_o ou alguma porta do arranjo kernel_o. Quando todos os valores de um canal de um filtro estão escritos nas portas adequadas, o sinal kernel_valid_o avisa ao núcleo convolucional que os pesos que ele enxerga em suas entradas podem ser utilizados. Quando isso ocorre, o núcleo sinaliza que os pesos estão sendo utilizados através da porta hold_kernel_i para que o módulo de leitura interrompa seu funcionamento e não sobrescreva os valores atuais.

Outro módulo comum às duas implementações é o *buffer* de janela. Esse módulo abstrai a estrutura dos *buffers* de linha que são uma técnica de otimização por meio de reutilização de dados. A ideia de um *buffer* de linha é guardar valores de uma mesma linha em uma fila conforme eles são lidos. Como mostrado na Figura 3.9, conforme os dados são recebido sequencialmente na entrada, eles são colocados na última linha de registradores da janela de valores que será usada no cômputo da convolução.

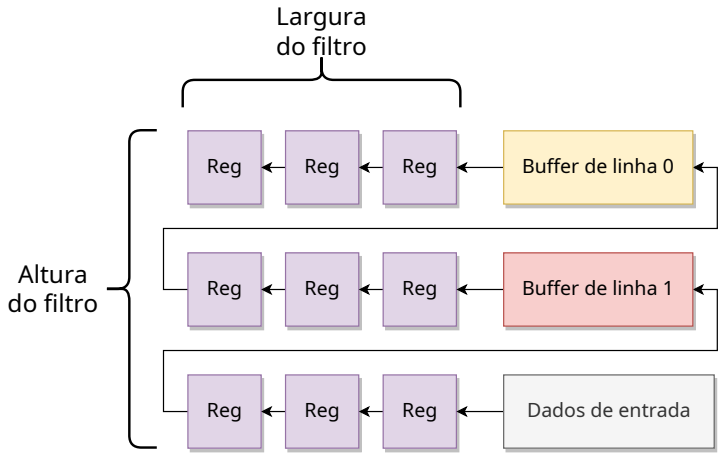


Figura 3.9: Diagrama de dois *buffers* de linha sendo usados para formar uma janela deslizante para convolução com filtro 3×3.

Fonte: próprio autor

Conforme um valor entra nessa janela, os valores que entraram anteriormente são passados para o registrador adjacente. Caso não haja um registrador adjacente, o valor é guardado no *buffer* da linha de cima. O *buffer* de linha tem um tamanho igual ao número de colunas da entrada, menos o número de colunas da janela. Dessa forma, os valores da janela de registradores passam pela imagem conforme a ilustração de exemplo na Figura 3.10.

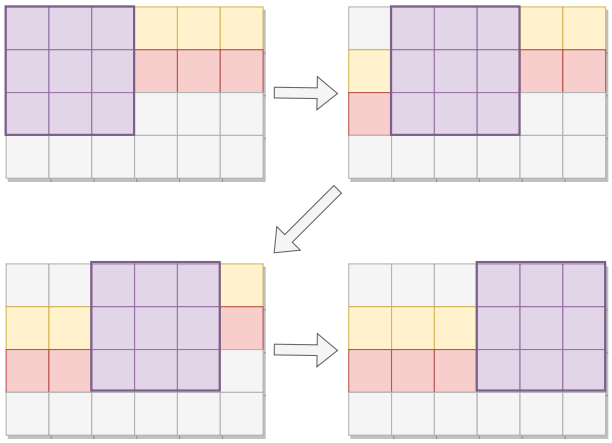


Figura 3.10: Ilustração de como os valores da janela de registradores deslizam sobre uma imagem de entrada reaproveitando os valores dos *buffers* de linha. Em roxo estão os valores expostos pela janela de registradores e em amarelo e vermelho estão os valores guardados nos dois *buffers* utilizados.

Fonte: próprio autor

O módulo implementado de *buffer* de janela é parametrizável pelas dimensões desejadas da janela de registradores e pelo tamanho da linha. Dessa forma, esse módulo cria os

registradores e *buffers* necessários e faz a devidas conexões. É importante notar que, na Figura 3.10, quando a janela chega no final de uma linha, o próximo valor lido da entrada resultará em uma janela inválida para a convolução. Esse caso é chamado, nesse trabalho, de desalinhamento de janela e a responsabilidade de lidar com essa situação é delegada ao circuito que faz uso do *buffer*.

Por fim, o último aspecto comum às duas implementações é a lógica de sequenciamento de canais e filtros. Esse componente da lógica consiste em uma máquina de dois estados, ilustrada na Figura 3.11, que alterna entre esperar valores válidos para um canal de entrada no estado `S1` e atualizar os valores de sinais internos de controle para processar o canal válido encontrado no estado `S2`. Na Figura 3.11, o sinal channel_valid é igual a 1 quando data_valid_i e kernel_valid_i são ambos iguais a 1 para o canal atual, representado por curr_channel.

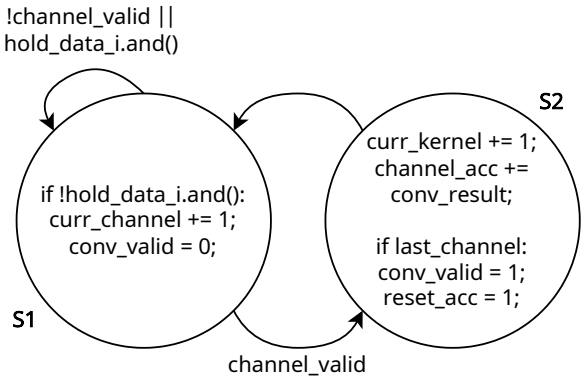


Figura 3.11: Máquina de estados para sequenciamento de canais e filtros do núcleo convolucional.

Fonte: próprio autor

A expressão hold_data_i.and() representa a conjunção lógica do sinal hold_data_i para todos os canais de entrada. Se todos os sinais estiverem sendo segurados pela camada seguinte, o canal atual não é sequenciado e os sinais de convolução válida, conv_valid, são mantidos. Caso contrário, a máquina de estados passa a verificar a validade do canal seguinte e abaixa todos os sinais de convolução válida.

No estado `S2`, todos os dados de entrada do canal atual estão na posição correta do *buffer* de janela e todos os pesos desse canal do filtro atual estão corretos. Então o filtro atual do canal atual, representado por curr_kernel, é sequenciado para o filtro seguinte. Nesse ponto, o sinal de hold_kernel_o para esse canal é abaixado, o que permite que a leitura do canal atual do próximo filtro se inicie.

Nesse estado, o resultado parcial da convolução do canal atual é válido e, portanto, é acumulado no registrador channel_acc. Caso o canal atual tenha sido o último canal

de entrada, a convolução entre a janela atual e o filtro está completa, então o sinal conv_valid é acionado para o canal atual e o acumulador de canais channel_acc tem seu valor reiniciado de acordo com o valor do *bias* do filtro.

3.4.1 Convolução Espacial

O esquemático dos componentes internos do núcleo de convolução espacial é mostrado na Figura 3.12.

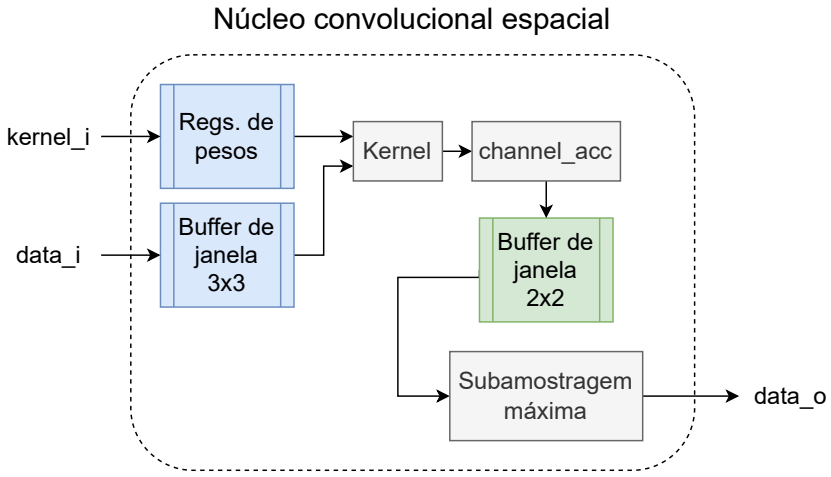


Figura 3.12: Diagrama dos componentes internos de um núcleo convolucional espacial. Elementos em azul são replicados para cada canal de entrada. Elementos em verde são replicados para cada filtro da camada, ou seja, canais de saída.

Fonte: próprio autor

Esse circuito utiliza um módulo chamado de *kernel*, que implementa a multiplicação entre os valores da janela de entrada de um canal e os valores de um canal do filtro de entrada. O módulo *kernel* realiza a multiplicação dos elementos de entrada com os valores do filtro de maneira paralela. Os resultados dessas multiplicações são somados entre si e acumulados no registrador channel_acc.

Controle de Entrada

A ideia do controle de entrada do núcleo convolucional é ter um *buffer* de janela para cada canal e, conforme os elementos são recebidos na entrada, determinar quando uma janela de um canal está válida para o processamento. Essa verificação da validade das janelas para um canal específico faz uso de dois contadores: um para o número de linhas de saída que faltam para concluir a convolução e outro para o número de elementos que foram inseridos no *buffer*.

O controle de entrada primeiramente verifica se resta alguma linha de saída para ser calculada. Caso não exista, o controle de entrada cessa seu funcionamento em um estado ocioso. Do contrário, verifica-se o sinal hold_data_o do canal. Se o sinal não estiver ativo, verifica-se o sinal data_valid_i do canal. Sendo o dado de entrada válido, o contador de elementos adicionados ao *buffer* é incrementado. Caso o número de elementos colocados no *buffer* atinja um determinado valor que indica que a primeira janela válida foi formada, o sinal de hold_data_o é ativado para indicar que não devem ser colocados novos valores na entrada enquanto a convolução da janela atual é calculada. Esse sinal é mantido ativo até que a lógica de sequenciamento de filtros o abaixe após o processamento do último filtro com essa janela.

Enquanto hold_data_o está ativo, o valor do contador de elementos do *buffer* de entrada é verificado para ver se a janela atual é a última da linha. Caso seja, significa que o próximo elemento adicionado ao *buffer* colocará a janela em desalinhamento. Nesse caso, o valor do contador de elementos no *buffer* é mudado para o valor que indica a primeira janela válida, menos o tamanho de uma linha do filtro. Dessa forma, o sinal hold_data_o só é levantado novamente após a leitura de elementos suficientes para recuperar o alinhamento da janela.

Controle de Saída

Após os valores das convoluções com cada janela de entrada serem calculados, é necessário realizar o processo de subamostragem. Para tal, um módulo de subamostragem máxima foi desenvolvido. Esse módulo recebe como entrada uma janela 2×2 de valores e expõe na saída o maior deles. Com essa simples interface, é possível usar o *buffer* de janela para controlar as entradas da subamostragem.

Um *buffer* de janela 2×2 é instanciado para cada filtro da camada. Conforme os valores de convolução são computados, eles são colocados no *buffer* e um contador de elementos é incrementado. A lógica de contar até um valor que corresponde à primeira janela válida e verificar quando a janela chega ao final da linha é feita de forma análoga à lógica do controle de entrada. A diferença aqui está no uso de sinais internos adicionais para controlar a sinalização de uma saída válida da subamostragem. Esses sinais garantem que uma saída válida é sinalizada por data_valid_o apenas a cada dois passos da janela de subamostragem. Para garantir o passo de dois no sentido vertical, esse contador também é zerado quando a janela chega ao final de uma linha. Além disso, é feito o controle para ignorar a última coluna quando a saída da convolução tem largura ímpar.

3.4.2 Convolução de Winograd

A implementação da convolução de Winograd é um pouco mais restrita, uma vez que é preciso estabelecer quais serão as matrizes de transformação utilizadas. Como tratado na Seção 2.2 do Capítulo 2, o algoritmo $F(2 \times 2, 3 \times 3)$ foi implementado, ou seja, convolução de filtro 3×3 com janela de entrada 4×4 para produzir janela de saída 2×2. Por conta disso, o módulo *kernel* do núcleo convolucional de Winograd não tem dimensões de entrada parametrizáveis. Esse módulo recebe uma janela de entrada 4×4 no domínio espacial, realiza a transformação linear para obter os valores no domínio de Winograd, realiza a multiplicação com os valores do filtro em paralelo e realiza a transformação inversa para obter o resultado 2×2 no domínio espacial. Esses valores são armazenados e acumulados em quatro registradores channel_acc conforme apresentado na Figura 3.13.

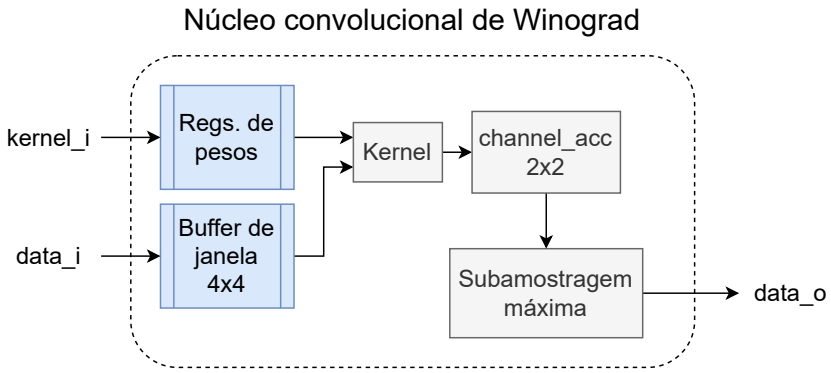


Figura 3.13: Diagrama dos componentes internos de um núcleo convolucional de Winograd. Elementos em azul são replicados para cada canal de entrada.

Fonte: próprio autor

Na implementação desse módulo, considerou-se que os pesos lidos da memória são referentes aos coeficientes dos filtros 4×4 já devidamente transformados. Isso aumenta o espaço de memória necessário para armazenar os pesos, mas diminui a complexidade do circuito.

Controle de Entrada

O controle de entrada do núcleo de Winograd é bastante similar ao controle de entrada do núcleo de convolução espacial. A principal diferença é a dimensão das janelas dos *buffers*, que são 4×4. Além disso, quando o contador de elementos do *buffer* é incrementado ao ser recebido um valor válido na entrada, o sinal hold_data_o correspondente só é acionado quando a diferença entre o valor do contador e o valor que indica a primeira janela válida é par. Dessa forma, o passo horizontal de 2 é garantido. Por fim, para garantir um passo vertical de 2, quando a janela chega ao final da linha, o contador de elementos do *buffer*

é decrementado em duas vezes o tamanho da linha. Dessa forma, a próxima janela válida só é alcançada após a leitura de duas linhas, resultando no passo vertical desejado.

Controle de Saída

Uma vez que a saída do módulo *kernel* de Winograd já é uma janela 2×2, não é necessário usar *buffers* para controlar a entrada do módulo de subamostragem. Portanto, o controle do sinal data_valid_o é feito na própria lógica de sequenciamento de canais e filtros apresentada na Figura 3.11. No núcleo de Winograd, o channel_acc é um arranjo 2×2 de registradores para acumular os resultados parciais. Assim, quando o resultado do último canal é adicionado, o data_valid_o correspondente é acionado no estado `S2` e desativado no estado seguinte.

3.5 Camada Totalmente Conectada

Após a última camada convolucional, tem-se a camada Totalmente Conectada (do inglês, *Fully Connected*). Como pode ser visto em sua interface apresentada na Figura 3.14.

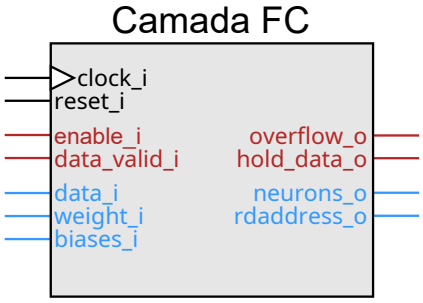


Figura 3.14: Interface do módulo de camada FC. Portas de entrada à esquerda e portas de saída à direita. Portas destacadas em azul representam o caminho de dados, enquanto as em vermelho são sinais de controle.

Fonte: próprio autor

Este módulo possui apenas dois sinais de controle para se comunicar com a camada anterior: data_valid_i e hold_data_o. O primeiro sinal para verificar quando o valor de saída da camada anterior é válido. Quando esse sinal é ativado, o sinal a camada FC levanta o sinal hold_data_o para indicar que o valor atual não deve ser sobrescrito.

O módulo de camada FC comunica-se diretamente com sua memória de pesos usando o sinal de rdaddress_o para indicar o endereço do valor atual, que é lido pela porta weight_i. Para cada neurônio, uma porta bias_i recebe o valor de seu peso de *bias*. Esses valores são usados para inicializar os registradores de saída de cada neurônio, representados pelo sinal neurons_o.

Quando esse módulo recebe um valor de entrada válido, ele sinaliza que esse valor deve ser mantido e prossegue com o seu processamento. Para cada neurônio, um peso correspondente é lido da memória, multiplicado pelo valor de entrada e acumulado no registrador correspondente do arranjo neurons_o. Uma vez que todos os registradores são atualizados dessa forma, o sinal hold_data_o é abaixado para que um novo valor possa ser recebido.

Uma outra possibilidade considerada foi a implementação da camada FC de maneira combinacional. Com essa implementação, os valores de saída da segunda camada convolucional seriam entregues como entradas da rede FC que realizaria a multiplicação de cada entrada pelos pesos correspondentes (diretamente implementados em hardware, sem necessidade de registradores) e acumulados para a obtenção da saída de cada neurônio. As vantagens e limitações dessa abordagem são avaliadas no capítulo seguinte, que apresenta os resultados obtidos no projeto.

Capítulo 4

Resultados Obtidos

Este capítulo apresenta os resultados obtidos e é estruturado da seguinte forma: a Seção 4.1 trata da caracterização espacial dos circuitos de convolução projetados, a Seção 4.2 trata da caracterização das duas abordagens de camada FC e, por fim, a Seção 4.3 apresenta os resultados obtidos a partir da implementação de uma rede simples usada como estudo de caso.

A placa de desenvolvimento utilizada para implementar os circuitos do projeto foi a DE1-SoC, que possui um FPGA Intel Cyclone V modelo 5CSEMA5F31C6 com 32.070 Módulos Lógicos Adaptativos (do inglês *Adaptive Logic Modules*) (ALMs), 85.000 Tabelas de Pesquisa Adaptativas (do inglês *Adaptive Lookup Tables*) (ALUTs) e 87 Processadores de Sinal Digital (do inglês *Digital Signal Processors*) (DSPs). Por DSPs entende-se um circuito especializado em fazer multiplicações em hardware. A precisão de representação numérica adotada na implementação dos circuitos foi de ponto fixo de 32 *bits* Q16, ou seja, 16 *bits* fracionários. Alguns resultados de caracterização espacial não puderam ser compilados para carregamento na placa e portanto apresentam uso de Elementos Lógicos em ALUTs ao invés de ALMs. Os códigos fonte desse projeto estão disponíveis publicamente no repositório do GitHub[1].

O projeto foi desenvolvido usando a ferramenta Quartus Prime Lite Edition, versão 21.1, para compilação dos circuitos e programação do FPGA. A ferramenta *In-System Memory Content Editor* do Quartus foi utilizada para fazer a análise da memória de saída. A ferramenta Modelsim, versão 20.1, foi usada para realizar simulações de forma de onda. A linguagem Python foi utilizada para desenvolver programas auxiliares para pré-processamento de arquivos entrada e análise dos resultados. A máquina utilizada para o desenvolvimento do projeto foi um computador com processador AMD Ryzen 5 5600G, 16GiB de RAM e sistema operacional Ubuntu 22.04.2 LTS.

[1] `https://github.com/ArturHugo/cnn_fpga.git`

4.1 Caracterização do Núcleo Convolucional

Para verificar como variam os requisitos espaciais dos circuitos implementados em função de seus parâmetros, cada módulo foi sintetizado separadamente. Para os módulos de convolução, os parâmetros variados foram o número de canais de entrada e o número de filtros. Outros parâmetros que podem influenciar nos requerimentos espaciais do circuito são a dimensão horizontal da entrada e as dimensões dos filtros. Para simplificar a análise, esses parâmetros não foram variados. A dimensão horizontal da entrada influencia apenas no tamanho dos *buffers* de linha, além de ser bastante limitada para a implementação física no FPGA utilizado. Já as dimensões dos filtros só poderiam ser variadas para o circuito da convolução espacial. Portanto, a caracterização aqui apresentada é feita considerando largura de entrada 28 e dimensões de filtro 3×3.

A Tabela 4.1 apresenta os requerimentos de ALUTs, registradores, *bits* de memória e DSPs para os circuitos com três canais de entrada e diferentes números de filtros.

Tabela 4.1: Requisitos espaciais para os módulos de convolução para 3 canais de entrada e diferentes quantidades de filtros.

Requisitos	**Conv. Espacial (3 canais)**			**Conv. Winograd (3 canais)**		
	32 filtros	**48 filtros**	**64 filtros**	**32 filtros**	**48 filtros**	**64 filtros**
ALUTs	5.648	7.517	9.360	5.243	5.767	6.293
Regs	4.704	6.646	8.582	1.533	1.555	1.571
Bits	30.592	43.392	56.192	7.200	7.200	7.200
DSPs	27	27	27	48	48	48

O circuito com três canais seria utilizado na prática para implementar a primeira camada convolucional de muitas arquiteturas. Os valores de filtros testados foram de 32, 48 e 64. Essas mesmas quantidades de filtros foram avaliadas para os circuitos com 32 canais de entrada e com 64 canais de entrada, conforme apresentado na Tabela 4.2 e na Tabela 4.3 respectivamente.

Tabela 4.2: Requisitos espaciais para os módulos de convolução para 32 canais de entrada e diferentes quantidades de filtros.

Requisitos	**Conv. Espacial (32 canais)**			**Conv. Winograd (32 canais)**		
	32 filtros	**48 filtros**	**64 filtros**	**32 filtros**	**48 filtros**	**64 filtros**
ALUTs	11.406	12.680	13.841	15.748	15.826	15.871
Regs	11.918	13.870	15.758	14.062	14.142	14.158
Bits	32.064	37.184	42.304	29.760	29.760	29.760
DSPs	27	27	27	48	48	48

Tabela 4.3: Requisitos espaciais para os módulos de convolução para 64 canais de entrada e diferentes quantidades de filtros.

Requisitos	**Conv. Espacial (64 canais)**			**Conv. Winograd (64 canais)**		
	32 filtros	**48 filtros**	**64 filtros**	**32 filtros**	**48 filtros**	**64 filtros**
ALUTs	18.853	20.252	21.423	27.882	28.060	28.088
Regs	20.016	22.032	23.920	27.952	28.096	28.112
Bits	53.888	59.008	64.128	59.520	59.520	59.520
DSPs	27	27	27	48	48	48

É possível observar que, para um mesmo número de canais, o circuito de convolução espacial tem um aumento mais substancial de ALUTs, registradores e *bits* de memória conforme os números de filtros aumenta. Já a convolução de Winograd, embora seus requisitos não variem muito em função do número de filtros, seus recursos aumentam bem mais para um maior número de canais.

Um ponto interessante de se observar são os valores de *bits* de memória, que variam em função do número de filtros apenas para o circuito de convolução espacial. Isso é esperado, uma vez que esse circuito implementa um *buffer* de janela por filtro para fazer o controle de subamostragem.

O último ponto que vale destacar é o uso de DSPs. A quantidade de DSPs utilizada por cada circuito não varia com o número de canais ou o número de filtros, pois os DSPs são usados apenas para implementar os multiplicadores necessários para calcular os produtos de elementos de entrada por pesos dos filtros. Como o produto de um canal do filtro é feito em paralelo, o número de DSPs usados depende apenas das dimensões de altura e largura dos filtros. Evidentemente, o circuito de Winograd necessita de um número maior de multiplicadores, pois o filtro 3×3 transformado torna-se 4×4.

4.2 Caracterização da Camada FC

A camada FC também foi analisada e sintetizada separadamente para fazer sua caracterização. Para a abordagem sequencial, considerando uma camada de 10 neurônios, o circuito usa 1.966 ALUTs, 664 registradores dedicados e 30 DSPs. Na Seção 4.3 a seguir, os requisitos temporais desse módulo são levados em conta pelas frequências máximas obtidas por cada circuito implementado no estudo de caso.

Para a análise da abordagem combinacional, foram caracterizados tanto os requisitos físicos quanto o requisito temporal de atraso de propagação t_{pd}. Esses valores são apresentados na Tabela 4.4 e foram obtidos a partir da compilação do circuito de 10 neurônios,

considerando números de entrada de 2 a 15. A partir desses valores pode ser possível extrapolar os valores necessários para o estudo de caso.

Tabela 4.4: Requisitos temporais e espaciais para o circuito de camada FC combinacional.

Número de entradas	**ALMs**	**ALUTs**	**DSPs**	t_{pd} **(ns)**
2	743 (2%)	460	40 (46%)	15,535
3	992 (3%)	600	60 (69%)	20,270
4	1.415 (4%)	1.100	80 (92%)	21,451
5	1.661 (5%)	1.240	87 (100%)	22,971
6	2.758 (9%)	2.946	87 (100%)	27,570
7	7.697 (24%)	11.122	87 (100%)	32,213
8	12.499 (39%)	19.018	87 (100%)	27,822
9	16.673 (52%)	26.033	87 (100%)	34,974
10	19.369 (60%)	30.484	87 (100%)	40,315
11	22.061 (69%)	35.076	87 (100%)	35,097
12	24.747 (77%)	39.486	87 (100%)	42,663
13	27.236 (85%)	43.980	87 (100%)	32,497
14	29.958 (93%)	48.511	87 (100%)	37,070
15	32.070 (100%)	53.023	87 (100%)	38,549

Vale já observar que, para um número bem reduzido de entradas (15), o circuito já esgota os recursos físicos do FPGA. A partir de 5 entradas a quantidade de DSP é esgotada, sendo então o restante dos multiplicadores necessários implementados usando ALMs.

4.3 Estudo de Caso

Por fim, um problema simples, de classificação de dígitos escritos à mão, foi escolhido para verificar o erro introduzido pela precisão de ponto fixo e pelas transformações de Winograd. A escolha desse problema se deve ao fato de que a complexidade da CNN necessária para obter bons resultados é relativamente baixa em relação a problemas de classificação mais gerais e problemas de detecção e segmentação.

A rede implementada foi treinada com o conjunto de dados MNIST [40] e sua arquitetura é ilustrada na Figura 4.1.

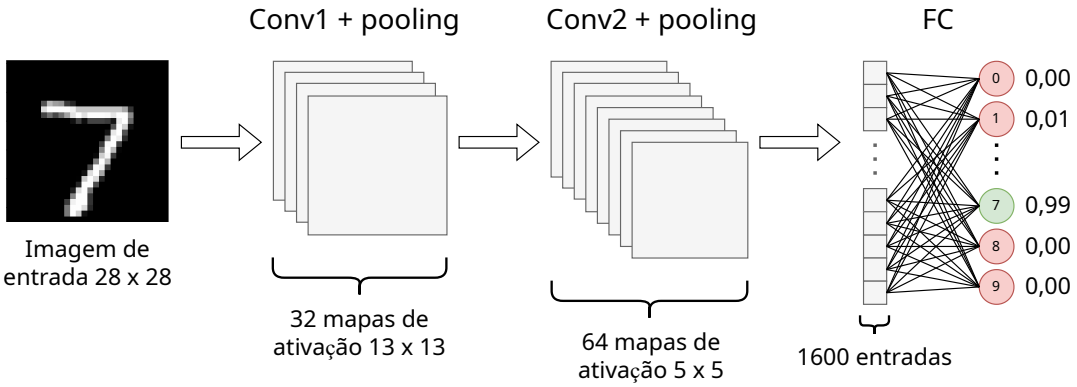


Figura 4.1: Diagrama da CNN implementada. Os neurônios da camada FC passam pela função de ativação *softmax* para retornarem as probabilidades da classificação da entrada.

Fonte: próprio autor

A rede consiste em duas camadas convolucionais de 32 e 64 filtros respectivamente e uma camada FC com 10 neurônios (um neurônio para cada dígito de 0 a 9). Essa arquitetura é um exemplo apresentado na documentação da biblioteca Keras e foi escolhida como estudo de caso por atingir acurácia de aproximadamente 99% no conjunto de teste da base de dados utilizada [41].

Devido a uma limitação da implementação desenvolvida, as camadas convolucionais não funcionam para entradas de apenas um canal. Portanto, foi decidido treinar a rede de exemplo para aceitar como entradas imagens de três canais (sistema de cor RGB).

4.3.1 Análise do Erro

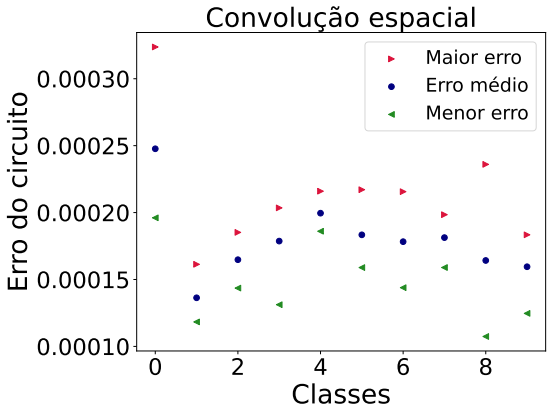
Para testar as implementações da CNN de exemplo, foi feita uma subamostragem aleatória de 30 imagens do conjunto de teste, sendo 3 imagens de cada classe. Esse subconjunto de teste foi utilizado para avaliar a magnitude dos erros introduzidos pela precisão de ponto fixo e pelas transformações do algoritmo de Winograd. Como o circuito não implementa a ativação *softmax* da camada FC, essa função de ativação foi removida da implementação em *software* para obter os valores de referência dos neurônios.

A Tabela 4.5 apresenta os erros médios e os desvios padrão de cada implementação em *hardware* para cada classe de dígito. A média e o desvio padrão dos erros para cada classe foram obtidos a partir das três amostras testadas daquela classe. Os dados apresentados demonstram que a diferença média entre os neurônios da implementação em *software* e os neurônios implementados em *hardware* é de menos de um centésimo de unidade. Considerando que a menor diferença encontrada entre os dois maiores valores de neurônios foi de mais de duas unidades para uma amostra da classe 3, é razoável concluir que a chance do erro introduzido pelos circuitos gerar um erro de classificação é bem baixa.

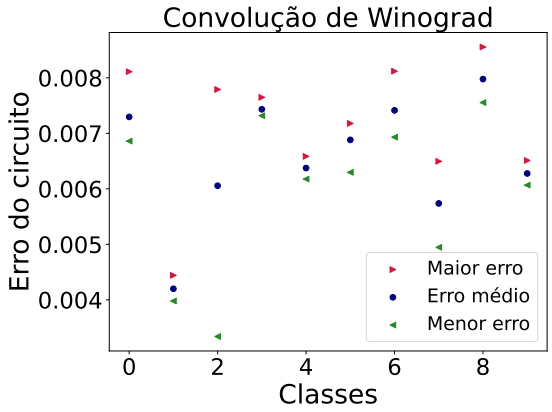
Tabela 4.5: Erros médios e desvios padrão dos resultados de cada implementação para cada classe de dígito.

Classe	Conv. Espacial (10^{-4})	Conv. Winograd (10^{-3})
0	$2{,}5 \pm 1{,}6$	$7{,}2 \pm 4{,}4$
1	$1{,}4 \pm 0{,}9$	$4{,}2 \pm 2{,}0$
2	$1{,}6 \pm 1{,}5$	$6{,}1 \pm 4{,}4$
3	$1{,}8 \pm 1{,}4$	$7{,}4 \pm 4{,}3$
4	$2{,}0 \pm 1{,}6$	$6{,}4 \pm 3{,}2$
5	$1{,}8 \pm 1{,}2$	$6{,}9 \pm 4{,}1$
6	$1{,}8 \pm 1{,}3$	$7{,}4 \pm 4{,}5$
7	$1{,}8 \pm 1{,}4$	$5{,}7 \pm 3{,}7$
8	$1{,}6 \pm 1{,}1$	$8{,}0 \pm 4{,}4$
9	$1{,}6 \pm 0{,}9$	$6{,}3 \pm 4{,}4$

Os gráficos apresentados na Figura 4.2 mostram a proporção dos erros em relação ao menor e ao maior erro obtidos para cada classe. Na Figura 4.2a é possível observar que o maior erro obtido com a implementação do algoritmo de convolução espacial não passou de $3{,}5 \times 10^{-4}$. Já na Figura 4.2b, vê-se que o maior erro do circuito de Winograd não passa de 9×10^{-3}. Com isso, pode-se notar que o erro introduzido pelas transformações da implementação de Winograd são uma ordem de grandeza maiores que os erros introduzidos apenas pela precisão de ponto fixo.



(a) Erros médio, mínimo e máximo para cada classe obtidos pelo circuito de convolução espacial.



(b) Erros médio, mínimo e máximo para cada classe obtidos pelo circuito de convolução de Winograd.

Figura 4.2: Erros nas saídas da rede FC calculados pelas implementações em *hardware* em relação aos valores de referência da implementação em *software*.

Fonte:próprio autor

4.3.2 Análise Espacial

A Tabela 4.6 apresenta os requisitos espaciais obtidos a partir da síntese do estudo de caso para circuito espacial e para circuito de Winograd. A partir dos dados apresentados, nota-se imediatamente que a abordagem de Winograd gasta um pouco mais de recursos que a abordagem espacial, especialmente em termos de DSPs, que ultrapassam o limite de 87 do dispositivo alvo.

Tabela 4.6: Requisitos espaciais obtidos pela síntese da rede de exemplo para as implementações espacial e de Winograd com e sem camada FC.

Requisitos	**Conv. Espacial**		**Conv. Winograd**	
	Sem FC	**Com FC**	**Sem FC**	**Com FC**
ALUTs	22.495	21.841	24.512	23.895
Regs	32.619	32.399	35.688	35.468
Bits	2.628.800 (65%)	3.153.088 (78%)	2.592.864 (64%)	3.117.152 (77%)
DSPs	56 (64%)	86 (99%)	98 (113%)	128 (147%)

Já a Tabela 4.7 apresenta os requisitos físicos após o mapeamento do circuito no dispositivo alvo. Dessa forma, é possível ter uma noção da proporção dos recursos utilizados, uma vez que o mapeamento usa os Módulos Lógicos Adaptativos (do inglês *Adaptive Logic Modules*) (ALMs) do FPGA, que são limitados a 32.070 unidades.

Tabela 4.7: Requisitos espaciais físicos obtidos pelo mapeamento na FPGA da rede de exemplo para as implementações espacial e de Winograd com e sem camada FC.

Requisitos	**Conv. Espacial**		**Conv. Winograd**	
	Sem FC	**Com FC**	**Sem FC**	**Com FC**
ALMs	21.887 (68%)	22.831 (71%)	22.554 (70%)	23.359 (73%)
Regs	34.656	34.455	37.029	36.822
Bits	2.589.424 (64%)	3.091.552 (76%)	2.590.394 (64%)	3.091.392 (76%)
DSPs	56 (64%)	86 (99%)	87 (100%)	87 (100%)

Cada uma das abordagens foi sintetizada e mapeada com e sem a camada FC sequencial. Dessa forma, é possível ver quantos recursos são necessários apenas para as camadas convolucionais. Assim, pode-se extrapolar os requisitos físicos necessários para uma FC combinacional de 1.600 entradas (o número necessário para o estudo de caso) a partir dos dados apresentados na Tabela 4.4.

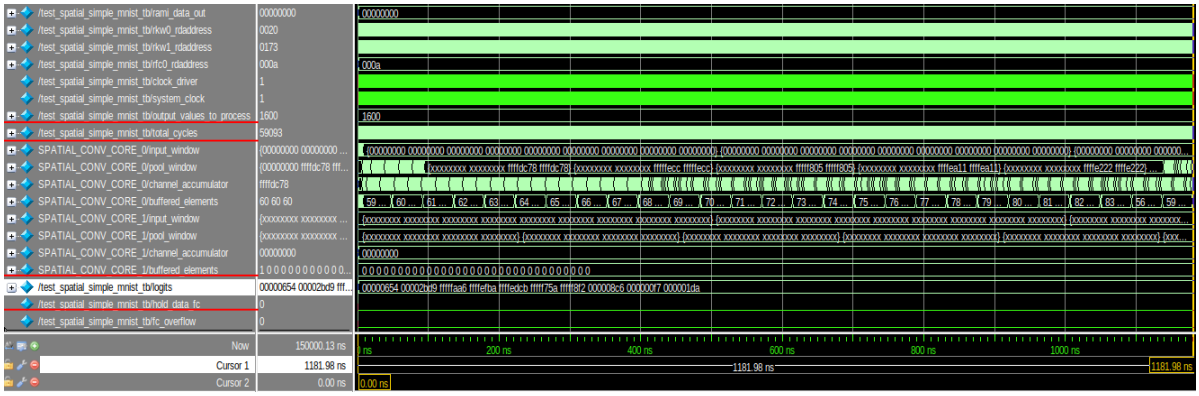
Extrapolando os valores de ALMs obtidos para baixos números de entradas na Tabela 4.4 com uma regressão linear, estima-se que seriam necessários 7.379.857 ALUTs

para a síntese do circuito e 4.128.064 ALMs para mapear a camada FC combinacional da rede de exemplo. Comparando esses valores com os dados da Tabela 4.6 e da Tabela 4.7, fica claro a inviabilidade de implementar o circuito dessa forma.

Devido a esta limitação, foi utilizado o circuito alternativo de camada FC sequencial, o que pode ter gerado um possível atraso no desempenho temporal da inferência. Essa questão é tratada em maior detalhe na Subseção 4.3.3.

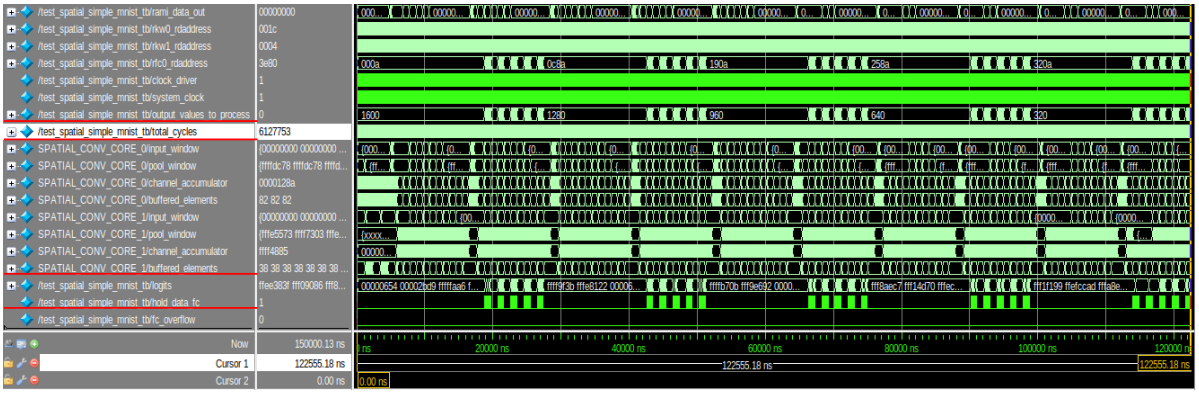
4.3.3 Análise Temporal

Para avaliar o desempenho temporal com maior granularidade, foram feitas simulações de forma de onda para avaliar os atrasos de cada módulo em cada uma das abordagens apresentadas. Todas as simulações foram feitas levando em conta o uso da camada FC sequencial. As simulações para o circuito de algoritmo espacial e para o circuito de algoritmo de Winograd são apresentadas respectivamente pela Figura 4.3 e pela Figura 4.4.



(a) Simulação até a obtenção da primeira saída da primeira camada convolucional.

(b) Simulação até a obtenção da primeira saída da segunda camada convolucional.

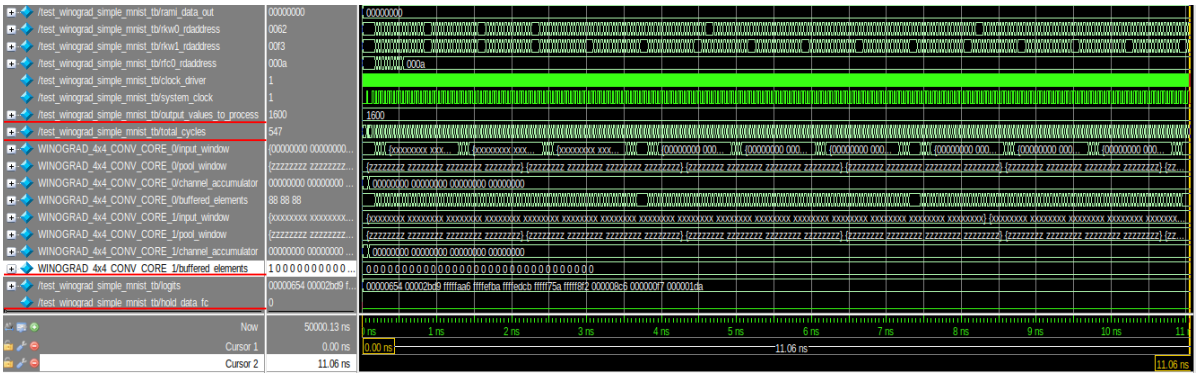


(c) Simulação completa (até a obtenção do resultado final da camada FC).

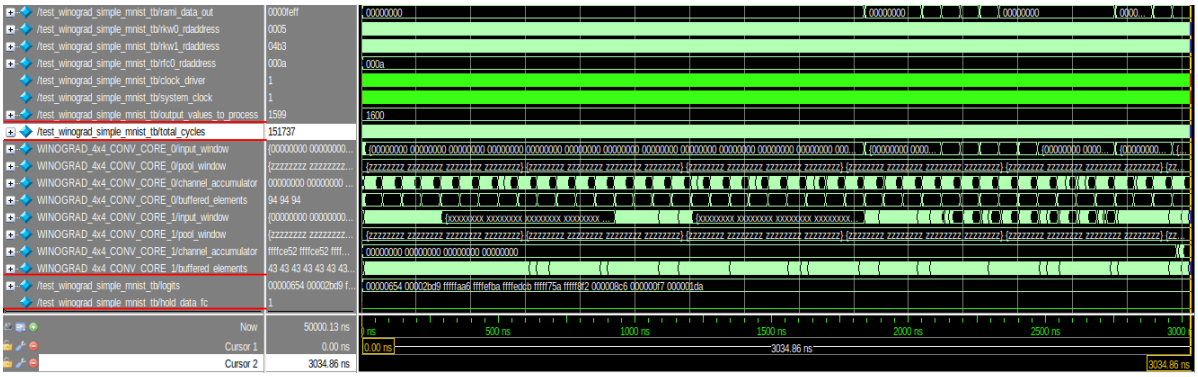
Figura 4.3: Simulação do estudo de caso usando algoritmo espacial.

A Figura 4.3a e a Figura 4.4a abrangem o período de simulação até a obtenção da primeira saída da primeira cama convolucional. Isso é monitorado pelo sinal de buffered_elements da segunda camada, destacado pelo número 3. Esse sinal é um arranjo de registradores que conta quantos valores foram inseridos, desde a última janela válida, em cada canal do *buffer* de entrada da camada.

Os valores dos sinais apresentados nas figuras, que podem ser vistos na segunda coluna, correspondem ao estado do sinal ao fim do período de simulação. Ou seja, os períodos



(a) Simulação até a obtenção da primeira saída da primeira camada convolucional.



(b) Simulação até a obtenção da primeira saída da segunda camada convolucional.

(c) Simulação completa (até a obtenção do resultado final da camada FC).

Figura 4.4: Simulação do estudo de caso usando algoritmo de Winograd.

das simulações da Figura 4.3a e da Figura 4.4a terminam quando o primeiro elemento do buffered_elements destacado é 1. Isso representa que a primeira camada calcula sua primeira saída, que é recebida pelo primeiro canal do *buffer* da camada seguinte.

A Figura 4.3b e a Figura 4.4b abrangem o período até a obtenção da primeira saída da segunda camada convolucional, o que é monitorado pela primeira ativação do sinal hold_data_fc, destacado nas imagens com o número 4. Por fim, a Figura 4.3c e a Figura 4.4c apresentam a duração total da simulação até obtenção dos resultados finais, indicado pelo valor 0 do sinal output_values_to_process (destacado pelo número 1), que representam o número de elementos do mapa de ativação da segunda camada que faltam para a camada FC processar.

Para facilitar a comparação, os valores do sinal total_cycles de cada simulação, destacados pelo número 2, são apresentados na Tabela 4.8.

Tabela 4.8: Número de ciclos necessários para cada circuito calcular a primeira saída da primeira camada, da segunda camada e para chegar no resultado final.

Período	**Conv. Espacial**	**Conv. Winograd**	**Razão $\frac{\text{Winograd}}{\text{Espacial}}$**
Primeira camada	59.093	547	0,93%
Segunda camada	939.041	151.737	16,16%
Resultado final	6.127.753	2.334.853	38,10%

Observando os valores lado a lado, fica evidente que a abordagem da convolução de Winograd de calcular quatro saídas simultaneamente dá uma grande vantagem temporal a essa implementação. A última coluna da Tabela 4.8 apresenta a razão entre o número de ciclos usado pelo circuito de Winograd e o número de ciclos usado pelo circuito espacial. Para a obtenção do resultado final, o circuito de Winograd usou 38,10% do número de ciclos necessário para o circuito espacial.

Além das simulações, para avaliar o atraso introduzido pela camada FC sequencial, foram medidos também os números de ciclos necessários para computar a saída da última camada convolucional sem uma camada FC conectada a ela. A Tabela 4.9 apresenta os valores obtidos para cada circuito implementado com e sem a camada FC sequencial.

Tabela 4.9: Frequências máximas, números de ciclos e tempos totais para processamento de uma entrada com e sem uma camada FC sequencial.

Resultados	**Conv. Espacial**		**Conv. Winograd**	
	Sem FC	**Com FC**	**Sem FC**	**Com FC**
f_{max} (MHz)	29,54	22,15	21,25	19,97
Ciclos	6.125.385	6.127.753	2.334.853	2.334.853
Tempo total (ms)	207,36	276,65	109,88	116,92

É possível observar pelos valores que o circuito de Winograd não foi impactado pela natureza sequencial da camada, ou seja, o tempo que a camada FC leva para ler os próximos pesos da memória é menor que o tempo que a camada anterior leva para calcular a próxima saída de um mesmo canal. Já no caso da convolução espacial, em alguns casos a camada convolucional teve que esperar a descida do sinal de hold_data da camada FC, resultando em uma diferença de 2.368 ciclos.

A partir da análise de tempos de atraso feita na Seção 4.2, é possível extrapolar o valor do atraso de propagação, obtendo-se $t_{pd} = 2.721$ ns para o circuito de FC combinacional de 1.600 entradas necessário para o estudo de caso. Levando em conta o valor de 29,54 MHz de frequência máxima obtidos para o circuito sem FC na Tabela 4.9, tem-se que o atraso introduzido pela FC combinacional seria de 81 ciclos. Portanto, para o circuito de convolução espacial, seria vantajoso usar a FC combinacional.

Já para o circuito de Winograd, considerando a frequência de 21,25 MHz do circuito sem a FC sequencial, seriam introduzidos 58 ciclos de atraso, mantendo o mesmo tempo total. Logo, também seria vantajoso para o circuito de Winograd usar a FC combinacional se não houvessem limitações espaciais para o circuito.

Considerando todos os resultados obtidos, o próximo capítulo apresenta uma breve discussão e considerações finais do projeto, além de elencar pontos a serem desenvolvidos em trabalhos futuros.

Capítulo 5

Conclusões

A área de Inteligência Artificial têm avançado muito com o advento do Aprendizado Profundo (do inglês *Deep Learning*). Uma das classes de Redes Neurais Profundas (do inglês, *Deep Neural Networks*) mais utilizadas nos dias de hoje são as Redes Neurais Convolucionais (do inglês, *Convolutional Neural Networks*), usadas frequentemente para análise de imagens e vídeos. Os métodos altamente eficazes de DL, no entanto, vêm acompanhados de um elevado custo computacional. Para mitigar esse custo, diversas técnicas e tecnologias são implementadas para acelerar e otimizar o processamento de CNNs.

Este trabalho se propôs a explorar implementações de algumas dessas técnicas em Arranjo de Portas Programável em Campo (do inglês *Field-Programmable Gate Array*), uma plataforma flexível para o projeto de *hardware* dedicado. Duas implementações foram propostas: uma baseada no algoritmo convencional de convolução, também referido como convolução espacial, e outra baseada no algoritmo de Winograd. Para analisar o desempenho das duas abordagens e avaliar o erro introduzido pela representação numérica de ponto fixo e pelas transformações lineares do algoritmo de Winograd, um estudo de caso foi implementado fisicamente no *kit* de desenvolvimento DE1-SoC. Devido às limitações de recursos do FPGA utilizado, a CNN usada como estudo de caso foi uma rede de duas camadas convolucionais que resolve o simples problema de classificação de dígitos escritos à mão do conjunto de dados MNIST.

Os módulos implementados foram caracterizados em relação aos seus requisitos espaciais e foi percebido que o módulo de Winograd apresenta um custo mais elevado nesse quesito. Porém, no estudo de caso realizado, foi possível observar que o desempenho temporal do circuito de Winograd foi maior, concluindo o processamento de uma entrada da CNN de exemplo em quase um terço dos ciclos necessários para o circuito de convolução espacial. Esse resultado era esperado, uma vez que o algoritmo de Winograd realiza o cômputo de quatro elementos de saída da convolução simultaneamente para cada janela

de entrada processada. Portanto, seria necessário paralelizar quatro módulos de filtro da convolução espacial para atingir esse desempenho, o que certamente aumentaria os requisitos espaciais do circuito.

A avaliação do erro introduzido pelas implementações feitas foi realizada a partir da subamostragem aleatória de 30 amostras do conjunto de teste, sendo 3 de cada classe de dígito. A partir dos resultados obtidos, foi possível observar que o erro introduzido para a precisão em ponto fixo de 32 *bits*, sendo 16 deles fracionários, foi bem menos substancial que o erro introduzido pelas transformações lineares de Winograd. Porém, considerando que os maiores erros obtidos foram da ordem de 10^{-2} e que o valor mais alto de um neurônio de saída difere do segundo maior por unidades, pode-se concluir que os circuitos projetados impactam minimamente na acurácia da arquitetura implementada em *hardware*.

5.1 Trabalhos Futuros

Para trabalhos futuros, alguns pontos podem ser elencados. O primeiro deles é em relação ao estudo de caso. A escolha de uma arquitetura simples para a CNN implementada foi devido às limitações de recursos do *kit* de desenvolvimento utilizado. Em trabalhos futuros, pode ser interessante explorar o uso de recursos adicionais como a memória externa da DE1-SoC para armazenar uma quantidade maior de pesos ou usar um FPGA mais robusto.

O segundo ponto é a avaliação de outros aspectos de desempenho além do tempo de execução. Um ponto forte do uso de FPGAs para aceleração de CNNs é a possibilidade de alcançar maior eficiência no uso de energia, como evidenciado por alguns trabalhos [29] [31].

Por fim, o último aspecto a se destacar para trabalhos futuros são as técnicas de otimização utilizadas. Nesse trabalho, foram usadas técnicas de simplificação de computação através do algoritmo de Winograd e da representação de ponto fixo, técnicas de paralelismo como o desenrolamento de laços e técnicas de reutilização de dados com o uso de *buffers* de linha. Porém, existe uma gama de outras técnicas a serem exploradas, além de pontos no projeto atual que podem ser melhorados com técnicas já utilizadas. Um desses pontos é o processamento dos filtros de uma camada. O circuito proposto usa um núcleo convolucional por camada, o que acaba criando um gargalo nas camadas que possuem maior número de filtros. Uma possível melhoria seria delegar o processamento de uma camada a dois núcleos convolucionais em paralelo, sendo cada um responsável por metade dos filtros da camada.

Referências

[1] Frank Rosenblatt. The Perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958. 1

[2] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980. 4

[3] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, e Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989. 4

[4] Steven B Damelin e Willard Miller Jr. *The mathematics of signal processing*. Number 48. Cambridge University Press, 2012. 5

[5] Martin Anthony. *Discrete mathematics of neural networks: selected topics*. SIAM, 2001. 7

[6] Alex Krizhevsky, Ilya Sutskever, e Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012. 7, 10

[7] John S Bridle. Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In *Neurocomputing: Algorithms, architectures and applications*, pages 227–236. Springer, 1990. 8

[8] Shmuel Winograd. *Arithmetic complexity of computations*. CBMS-NSF regional conference series in applied mathematics 33. Society for Industrial and Applied Mathematics, 1987. 8

[9] Andrew Lavin e Scott Gray. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4013–4021, 2016. 8

[10] Y Chen e W du Plessis. Neural network implementation on a FPGA. In *IEEE AFRICON. 6th Africon Conference in Africa,*, volume 1, pages 337–342. IEEE, 2002. 10

[11] Babak Noory e Voicu Groza. A reconfigurable approach to hardware implementation of neural networks. In *CCECE 2003-Canadian Conference on Electrical and Computer Engineering. Toward a Caring and Humane Technology (Cat. No. 03CH37436)*, volume 3, pages 1861–1864. IEEE, 2003. 10

[12] Seul Jung e Sung su Kim. Hardware implementation of a real-time neural network controller with a DSP and an FPGA for nonlinear systems. *IEEE Transactions on Industrial Electronics*, 54(1):265–271, 2007. 10

[13] A Muthuramalingam, S Himavathi, e E Srinivasan. Neural network implementation using FPGA: issues and application. *International Journal of Electrical and Computer Engineering*, 2(12):2802–2808, 2008. 10

[14] Rajat Raina, Anand Madhavan, e Andrew Y Ng. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th annual international conference on machine learning*, pages 873–880, 2009. 10

[15] Dan Claudiu Cireşan, Ueli Meier, Luca Maria Gambardella, e Jürgen Schmidhuber. Deep, big, simple neural nets for handwritten digit recognition. *Neural computation*, 22(12):3207–3220, 2010. 10

[16] Je Yang, Seongmin Hong, e Joo-Young Kim. Fixar: A fixed-point deep reinforcement learning platform with quantization-aware training and adaptive parallelism. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 259–264, 2021. 10

[17] Slaviša Jovanović e Hiroomi Hikawa. A survey of hardware self-organizing maps. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–20, 2022. 10

[18] Hao Wang, Danfeng Qiu, Fen Ge, e Ying Yang. Implementation of bidirectional LSTM accelerator based on FPGA. In *2022 IEEE 22nd International Conference on Communication Technology (ICCT)*, pages 1512–1516, 2022. 10

[19] Ran Wu, Xinmin Guo, Jian Du, e Junbao Li. Accelerating neural network inference on FPGA-based platforms—a survey. *Electronics*, 10(9):1025, 2021. 10

[20] Peng-Fei Song, Jeng-Shyang Pan, Chun-Sheng Yang, e Chiou-Yng Lee. An efficient FPGA-based accelerator design for convolution. In *2017 IEEE 8th International Conference on Awareness Science and Technology (iCAST)*, pages 494–500, 2017. 10

[21] Stylianos I. Venieris e Christos-Savvas Bouganis. f-CNNx: A toolflow for mapping multiple convolutional neural networks on FPGAs. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 381–3817, 2018. 10

[22] Lin Bai, Yecheng Lyu, e Xinming Huang. A unified hardware architecture for convolutions and deconvolutions in CNN. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2020. 10

[23] Ahmad Shawahna, Sadiq M Sait, e Aiman El-Maleh. FPGA-based accelerators of deep learning networks for learning and classification: A review. *ieee Access*, 7:7823–7859, 2018. 10, 14

[24] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C Ling, e Gordon R Chiu. An OpenCL™ deep learning accelerator on Arria 10. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 55–64, 2017. 10

[25] Yun Liang, Liqiang Lu, Qingcheng Xiao, e Shengen Yan. Evaluating fast algorithms for convolutional neural networks on FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(4):857–870, 2019. 10

[26] Dong Wang, Ke Xu, e Diankun Jiang. Pipecnn: An OpenCL-based open-source FPGA accelerator for convolution neural networks. In *2017 International Conference on Field Programmable Technology (ICFPT)*, pages 279–282, 2017. 10

[27] Peng Lv, Wei Liu, e Jinghui Li. A FPGA-based accelerator implementation for YOLOv2 object detection using Winograd algorithm. In *2020 5th International Conference on Mechanical, Control and Computer Engineering (ICMCCE)*, pages 1894–1898, 2020. 10

[28] Zhijian Lin, Meng Zhang, Dongpeng Weng, e Fei Liu. An efficient accelerator with Winograd for novel convolutional neural networks. In *2022 5th International Conference on Circuits, Systems and Simulation (ICCSS)*, pages 126–130, 2022. 10

[29] Shuai Li, Yukui Luo, Kuangyuan Sun, Nandakishor Yadav, e Kyuwon Ken Choi. A novel FPGA accelerator design for real-time and ultra-low power deep convolutional neural networks compared with Titan X GPU. *IEEE Access*, 8:105455–105471, 2020. 11, 39

[30] Gianmarco Dinelli, Gabriele Meoni, Emilio Rapuano, e Luca Fanucci. Advantages and limitations of fully on-chip CNN FPGA-based hardware accelerator. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2020. 11

[31] Ahmed J. Abd El-Maksoud, Mohamed Ebbed, Ahmed H. Khalil, e Hassan Mostafa. Power efficient design of high-performance convolutional neural networks hardware accelerator on FPGA: A case study with GoogLeNet. *IEEE Access*, 9:151897–151911, 2021. 11, 39

[32] Shashi Kant Sharma, Anu Gupta, e Kota Solomon Raju. Energy efficient hardware implementation of 2-D convolution for convolutional neural network. In *2022 IEEE 9th Uttar Pradesh Section International Conference on Electrical, Electronics and Computer Engineering (UPCON)*, pages 1–6, 2022. 11

[33] Bahadir Özkilbaç, Ibrahim Yücel Ozbek, e Tevhit Karacali. Real-time fixed-point hardware accelerator of convolutional neural network on FPGA based. In *2022 5th International Conference on Computing and Informatics (ICCI)*, pages 1–5, 2022. 11

[34] Zhao Yiwei, Zou Hao, Tang Ming, e Lin Qiutong. An adaptive hardware accelerator for convolution layers with diverse sizes. In *2022 19th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP)*, pages 1–10, 2022. 11

[35] Hao-Ran Bai. A flexible and low-resource CNN accelerator on FPGA for edge computing. In *2023 3rd International Conference on Neural Networks, Information and Communication Engineering (NNICE)*, pages 646–650, 2023. 11

[36] Thiago S. Cruz, João P. M. Gomes, Lucas F. Martins, Danyllo W. Albuquerque, Gutemberg G. dos Santos, Danilo F.S. Santos, e Jemerson Damásio. Extensible hardware inference accelerator for FPGA using models from TensorFlow Lite. In *2023 IEEE International Conference on Consumer Electronics (ICCE)*, pages 1–5, 2023. 11

[37] S. Srilakshmi e G.L. Madhumati. A comparative analysis of HDL and HLS for developing CNN accelerators. In *2023 Third International Conference on Artificial Intelligence and Smart Energy (ICAIS)*, pages 1060–1065, 2023. 11

[38] Keras: Deep Learning for humans. `https://keras.io/`. Acessado em: 14/07/2023. 13

[39] HDF5: Reference manual. `https://docs.hdfgroup.org/hdf5/v1_12/_r_m.html`. Acessado em: 14/07/2023. 13

[40] Yann LeCun. The MNIST Database of Handwritten Digits. `http://yann.lecun.com/exdb/mnist/`. 29

[41] Simple MNIST convnet. `https://keras.io/examples/vision/mnist_convnet/`. Acessado em: 26/07/2023. 30