UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS FLORIANÓPOLIS, CENTRO TECNOLÓGICO
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Cleisson Fernandes da Silva

Acelerador para a Convolução de Redes Neurais Binárias visando Computação de Borda

Florianópolis
2022

Cleisson Fernandes da Silva

Acelerador para a Convolução de Redes Neurais Binárias visando Computação de Borda

Dissertação submetida ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Santa Catarina para a obtenção do título de mestre em Engenharia Elétrica.
Orientador: Prof. Eduardo Augusto Bezerra, Dr.

Florianópolis
2022

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Silva, Cleisson Fernandes da
Acelerador para a convolução de redes neurais binárias visando computação de borda / Cleisson Fernandes da Silva ; orientador, Eduardo Augusto Bezerra, 2022.
84 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós-Graduação em Engenharia Elétrica, Florianópolis, 2022.

Inclui referências.

1. Engenharia Elétrica. 2. Acelerador de domínio específico. 3. FPGA. 4. Rede neural binária. 5. Computação de borda. I. Bezerra, Eduardo Augusto. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Engenharia Elétrica. III. Título.

Cleisson Fernandes da Silva

Acelerador para a Convolução de Redes Neurais Binárias visando Computação de Borda

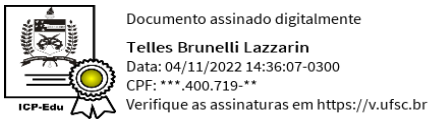
O presente trabalho em nível de mestrado foi avaliado e aprovado por banca examinadora composta pelos seguintes membros:

Prof. Anderson Wedderhoff Spengler, Dr.
UFSC

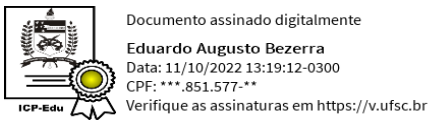
Prof Laio Oriel Seman, Dr.
UNIVALI

Prof. Renan Augusto Starke, Dr.
IFSC

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de mestre em Engenharia Elétrica.



Prof. Telles Brunelli Lazzarin, Dr.
Coordenação do Programa de
Pós-Graduação



Prof. Eduardo Augusto Bezerra, Dr.
Orientador

Florianópolis, 2022.

RESUMO

Com o crescimento exponencial de dados gerados por dispositivos IoT, o meio de comunicação está se tornando um gargalo para a computação em nuvem. Paradigmas como a computação de borda estão sendo cada vez de mais interesse por suas vantagens, tais como redução da latência de comunicação, uso mais eficiente da rede e o menor gasto energético. Ao transferir parte ou toda a computação para os dispositivos de borda é possível reduzir a quantidade de informações enviadas para a nuvem. Entre as aplicações que manipulam uma quantidade considerável de dados, pode-se destacar o processamento de vídeos e imagens. Através de algoritmos como as redes neurais é possível extrair propriedades e características úteis destes dados brutos. Dentro destes algoritmos, as redes neurais binárias têm atraído bastante interesse ultimamente devido às suas vantagens como a redução da complexidade computacional e de requisitos de memória. Além disso, em sistemas embarcados como encontrados nos dispositivos de borda, o uso de aceleradores de domínio específico para realizar uma determinada tarefa tem se mostrado altamente eficiente. Este trabalho apresenta o estudo, projeto e desenvolvimento de um acelerador de domínio específico para realizar o algoritmo de convolução binária presente nestas redes, visando a sua utilização nos sistemas embarcados dos satélites desenvolvidos no SpaceLab (UFSC). Através do uso de técnicas como especialização de dados e paralelismo de operações é possível obter ganho de desempenho significativo na execução do algoritmo em comparação com a sua execução em um núcleo de processamento. Os dados obtidos em simulação mostram que o acelerador executa em até 97,77% menos ciclos do que a execução utilizando apenas um núcleo RISC-V simples e em 96,06% menos ciclos do que um núcleo RISC-V com extensão de manipulação de bits. Além disso, a síntese para FPGA demonstra baixa utilização de recursos lógicos, permitindo sua utilização em dispositivos de baixa densidade disponíveis comercialmente.

Palavras-chave: Acelerador de Domínio Específico. FPGA. Rede Neural Binária. Computação de Borda.

ABSTRACT

With the exponential growth of data generated by IoT devices, communication medium is becoming a bottleneck for cloud computing. Paradigms such as edge computing are getting more and more interest for its advantages, such as reduced communication latency, more efficient use of the network, and lower energy consumption. By offloading some or all of the computation to the edge devices, it is possible to reduce the amount of information sent to the cloud. Among the applications that handle a considerable amount of data, one can highlight video and image processing. Through algorithms such as neural networks it is possible to extract useful properties and characteristics from this raw data Within these algorithms, binary neural networks have attracted a lot of interest lately due to their advantages such as the reduction of computational complexity and memory requirements. Furthermore, in embedded systems as found in edge devices, the use of domain-specific accelerators to perform a certain task has been shown to be highly efficient. This work presents the study, design and development of a domain-specific accelerator to perform the binary convolution algorithm found in these networks, with the goal of using it in the embedded systems of the satellites developed at SpaceLab. Through the use of techniques such as data specialization and operation parallelism, it is possible to obtain significant performance gains in the execution of the algorithm compared to its execution in a processing core. The data obtained in simulation shows that the accelerator runs in up to 97.77% less cycles than the execution using only a simple RISC-V core and in 96.06% less cycles than a RISC-V core with bit manipulation extension. In addition, FPGA synthesis demonstrates low utilization of logic resources, allowing its use in the lowest density devices commercially available.

Keywords: Domain Specific Accelerator. FPGA. Binary Neural Network. Edge Computing.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ASIC	Circuito Integrado de Aplicação Específica (*Application Specific Integrated Circuit*)
BNN	Rede Neural Binária (*Binary Neural Network*)
BRAM	Block RAM
CNN	Rede Neural Convolucional (*Convolutional Neural Network*)
CPU	Unidade Central de Processamento (*Central Processing Unit*)
DNN	Rede Neural Profunda (*Deep Neural Network*)
FPGA	Arranjo de Porta Programável em Campo (*Field Programmable Gate Array*)
GPU	Unidade de Processamento Gráfico (*Graphics Processing Unit*)
IoT	Internet das Coisas (*Internet of Things*)
ISA	Conjunto de Instruções (*Instruction Set Archtecture*)
LoRa	Long Range
LUT	Look-Up Table

SUMÁRIO

1 INTRODUÇÃO

Nos últimos anos, houve um grande crescimento dos dados gerados por dispositivos de Internet das Coisas (IoT, *Internet of Things*). Esses dispositivos, também chamados de dispositivos de borda, são caracterizados por recursos computacionais e energéticos limitados. Essa limitação foi reduzida fazendo com que estes dispositivos transfiram o processamento necessário a ser realizado com os dados para a nuvem. Embora a velocidade de processamento de dados dos dispositivos da nuvem tenha aumentado, a largura de banda das redes que transportam dados para a nuvem não aumentou consideravelmente. Assim, com dispositivos de borda gerando cada vez mais dados, a rede está se tornando o gargalo da computação em nuvem (PREMSANKAR; DI FRANCESCO; TALEB, 2018).

Portanto, devido ao crescimento exponencial de dispositivos IoT que se conectam à Internet para receber informações da nuvem ou entregar dados de volta à nuvem, surgiu o conceito de computação de borda (SHI; DUSTDAR, 2016). A computação de borda é um novo paradigma de computação em que os dados são processados no local onde os mesmos são produzidos. Este modelo de computação traz diversas vantagens como por exemplo:

- **Latência reduzida**: Estes sistemas podem ser colocados mais próximos dos usuários e serviços finais. Desse modo, isso naturalmente evita os atrasos de propagação de rede. Além disso, algumas aplicações possuem requisitos rigorosos de latência e desempenho em tempo real. Aplicações criadas em torno da experiência do usuário e aplicações de segurança automotiva são exemplos onde a latência não pode ser tolerada, portanto, os dados devem ser processados no próprio dispositivo (SORO, 2021);
- **Consumo de Rede**: Certos ambientes têm largura de banda limitada e a computação de borda pode atender melhor esse problema por meio de técnicas de filtragem, armazenamento em *cache*, extração de atributos e compactação de dados para maximizar com eficiência a largura de banda disponível (LEA, 2020); Ou seja, se os dispositivos puderem executar mais processamento nos dados brutos localmente (no próprio *hardware* local), menos dados brutos deverão ser transmitidos (PARKS, 2018);
- **Gasto energético**: Existem casos em que o gasto energético para mandar os dados e receber uma resposta seria maior do que processar os dados no próprio local;
- **Segurança e Privacidade**: Algumas situações devem proteger ou até mesmo remover certos dados antes que saiam do dispositivo de borda.

As áreas de aprendizado profundo e redes neurais, que demonstraram um grande desenvolvimento na última década, podem contribuir drasticamente para o

processamento de dados na borda (LI; OTA; DONG, 2018). Um algoritmo de rede neural pode extrair propriedades e características úteis dos dados brutos. Além disso, a Rede Neural Convolucional (CNN, *Convolutional Neural Network*), que é um tipo específico de rede neural, forneceu uma melhoria dramática de desempenho nos campos de processamento de imagem e visão computacional.

Observa-se em aplicações que é desejável utilizar a visão computacional para extrair as informações significativas de um vídeo diretamente no sensor de imagem e não na nuvem, para reduzir o custo de comunicação, dado que os vídeos envolvem uma grande quantidade de dados. Sendo assim, uma destas aplicações e motivação deste trabalho é a sua utilização nos sistemas embarcados dos satélites pesquisados e desenvolvidos no SpaceLab (Universidade Federal de Santa Catarina). Além das limitações de energia e tamanho, estes sistemas possuem limitações na comunicação ao se utilizar protocolos de comunicação IoT, tais como o LoRa.

Além desta aplicação, foi identificado na literatura outros casos de uso como veículos autônomos, navegação por drones e robótica, onde o processamento local é desejado, pois a latência e o risco de segurança de depender da nuvem são muito altos (SZE *et al.*, 2017a).

A implantação da inteligência artificial nos dispositivos de borda pode ajudar a obter dados mais significativos mais rapidamente, o que não apenas traz as vantagens mencionadas anteriormente como economia dos custos de comunicação e redução do atraso de resposta como também expande muito os cenários de aplicações de inteligência artificial (WANG *et al.*, 2020). Esses dispositivos não possuem inteligência para processar os dados de entrada e tomar decisões, em vez disso, dependem do usuário ou de outras entidades de processamento, como a nuvem, para tomar a decisão e controlá-los remotamente. Sendo assim, a inclusão dos algoritmos de inteligência artificial nestes dispositivos de borda podem permitir a emergência de dispositivos inteligentes que são capazes de processar os dados detectados, avaliar a situação e agir de forma independente (SAMIE; BAUER; HENKEL, 2019).

Portanto, um dos problemas atuais é como adicionar estas redes neurais complexas nestes dispositivos de borda sem reduzir o desempenho. As CNNs de última geração normalmente têm milhões de parâmetros e uma única tarefa de inferência pode invocar bilhões de operações aritméticas e acessos à memória. Os dispositivos onde esses algoritmos serão executados são geralmente constituídos por sistemas embarcados compostos por microcontroladores. Além disso, são normalmente alimentados por bateria e, como consequência, precisam funcionar em uma faixa de baixa potência.

Para contornar esse problema, deu-se início ao desenvolvimento de uma nova geração de redes neurais, mais compactas em tamanho e que focam na eficiência além da acurácia do modelo. Diversos trabalhos foram desenvolvidos do lado do al-

goritmo onde surgiram otimizações como remover pesos da rede neural para diminuir o número de operações (WEN *et al.*, 2016) ou diminuir a representação dos pesos e ativações para um valor de *bits* menor e assim utilizar menos recursos computacionais do *hardware* (HUBARA *et al.*, 2016).

Uma dessas otimizações que recebeu atenção ultimamente é a Rede Neural Binária (BNN, *Binary Neural Network*) (COURBARIAUX *et al.*, 2016), um tipo de rede neural convolucional onde os valores de ambos os pesos e ativações são limitados para apenas os valores +1 e -1. Além de um uso de memória reduzido, essas redes trazem uma grande vantagem na sua execução devido à simplificação das operações ao se utilizar apenas dois valores. As operações de multiplicação e acumulação, onde há o maior gasto computacional das CNNs, podem ser substituídas por operações de XNOR e *popcount* que são muito mais eficientes de serem executadas (RASTEGARI *et al.*, 2016). Inicialmente esse tipo de rede sofria de baixa acurácia devido à quantização massiva imposta na rede neural. No entanto, nos últimos anos, as pesquisas neste tipo de rede mostraram um grande avanço na sua acurácia ao se utilizar técnicas como topologias de rede neural customizadas e estratégias para o treinamento (YUAN; AGAIAN, 2021). Deste modo, as BNNs estão cada vez diminuindo a diferença de acurácia que existe entre elas e os modelos de rede neural que utilizam precisão total.

Além de otimizar o modelo de rede neural, transferir a sua execução para um acelerador de *hardware* dedicado surgiu como uma solução amplamente adotada para melhorar o tempo de execução e a eficiência energética. Isso vai ao encontro com a tendência atual de utilizar aceleradores de domínio específico para executar determinadas tarefas com mais eficiência. Essa tendência se deve ao fato de a lei de Moore (MOORE *et al.*, 1965) ter chegado ao fim (THEIS; WONG, 2017) e há uma busca por novas arquiteturas e tecnologias para continuar escalando desempenho e eficiência.

Diversos aceleradores para BNNs foram propostos como, por exemplo, FINN (UMUROGLU *et al.*, 2017), FP-BNN (LIANG, Shuang *et al.*, 2018) e XNE (CONTI; SCHIAVONE; BENINI, 2018). No entanto, grande parte destes aceleradores foram projetados para modelos de BNNs pequenos onde a imagem de entrada é muito reduzida para se utilizar em aplicações de visão computacional reais ou utilizam topologias de redes já obsoletas que não refletem o estado da arte. Além disso, como apontado por Bannink *et al.* (2021), muitos estudos de BNNs focam em desempenho teórico e muitas vezes carecem de dados empíricos.

Sendo assim, este trabalho foi realizado a fim de possibilitar o uso de redes neurais binárias do estado da arte no contexto de computação de borda através do uso de um acelerador de domínio específico para executar a operação de convolução binária e suprir as limitações impostas neste ambiente.

1.1 OBJETIVOS

Este trabalho apresenta como objetivo geral a pesquisa e desenvolvimento de um acelerador de domínio específico para a execução de Redes Neurais Binárias em dispositivos de computação de borda e análise de suas métricas relacionadas ao desempenho. Para realização do objetivo geral proposto, tem-se como objetivos específicos os seguintes tópicos:

- Realizar uma análise bibliográfica e do estado da arte.
- Elaborar uma microarquitetura para o acelerador.
- Implementar a microarquitetura proposta utilizando uma linguagem de descrição de *hardware*.
- Verificar, com base em síntese e simulações, o desempenho de execução do acelerador implementado.
- Avaliar os resultados obtidos.

2 FUNDAMENTAÇÃO TEÓRICA

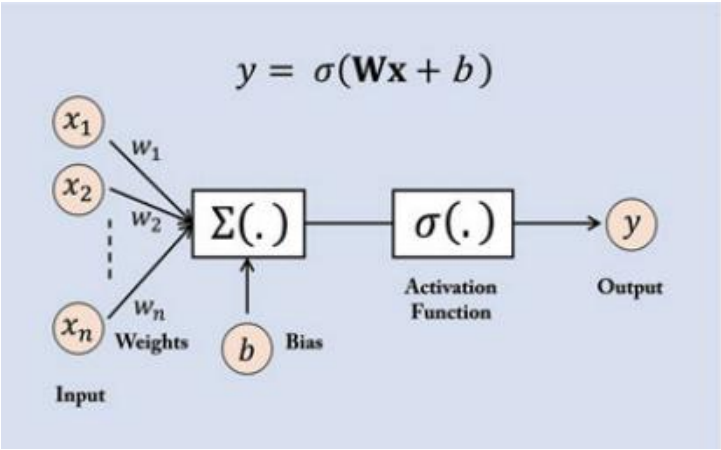
Neste capítulo são apresentados os conceitos base para o entendimento e desenvolvimento do trabalho. O capítulo está dividido em três seções. A Seção 2.1 compreende os conhecimentos pertinentes à área de redes neurais. A Seção 2.2 discorre sobre os aceleradores de domínio específico, a motivação de seu uso, suas arquiteturas e os aspectos relacionados ao seu uso no campo de redes neurais. A Seção 2.3 detalha os componentes utilizados para a concepção do trabalho.

2.1 REDES NEURAIS

2.1.1 Redes Neurais Artificiais

As Redes Neurais Artificiais são um conjunto de algoritmos pertencentes à área de Aprendizado de Máquinas, inspirados no cérebro humano, que são projetados para reconhecer padrões (AGGARWAL *et al.*, 2018). Estas redes são constituídas por unidades computacionais chamadas de neurônio artificial ou simplesmente unidade. O modelo matemático que descreve um neurônio artificial é ilustrado na Figura 1. Segundo Khan *et al.* (2018), este neurônio recebe um conjunto de dados de entrada que são ponderados por um conjunto de pesos, além de um valor de *bias*. Todos esses valores são somados e no valor resultante é aplicado uma função não linear (chamada de função de ativação) para calcular o valor de saída dessa unidade. Após isso, o valor de saída, também chamado de ativação, pode ser utilizado como entrada para outras unidades.

Figura 1 – Modelo computacional de um neurônio artificial

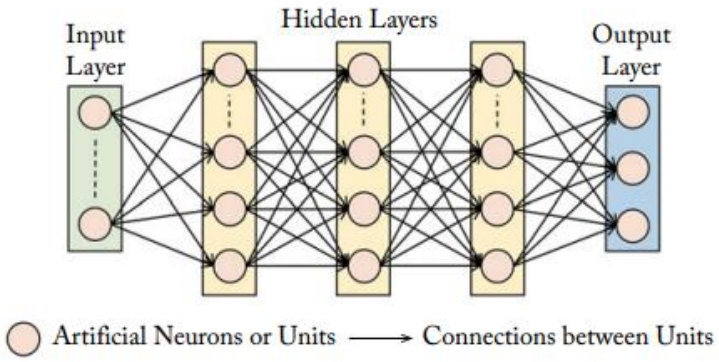


Fonte – Khan *et al.* (2018).

Desse modo, como ilustrado na Figura 2, uma rede neural artificial consiste em unidades computacionais interconectadas agrupadas em camadas, sendo: uma

camada de entrada para receber os dados a serem processados, uma camada de saída que fornece os resultados da tarefa da rede e um conjunto de camadas internas chamadas de camadas ocultas (VÉSTIAS *et al.*, 2020). Segundo Goodfellow, Bengio e Courville (2016), uma rede neural com mais de três camadas ocultas é designada Rede Neural Profunda (DNN, *Deep Neural Network*) para enfatizar o alto número de camadas.

Figura 2 – Estrutura de uma rede neural artificial



Fonte – Khan *et al.* (2018).

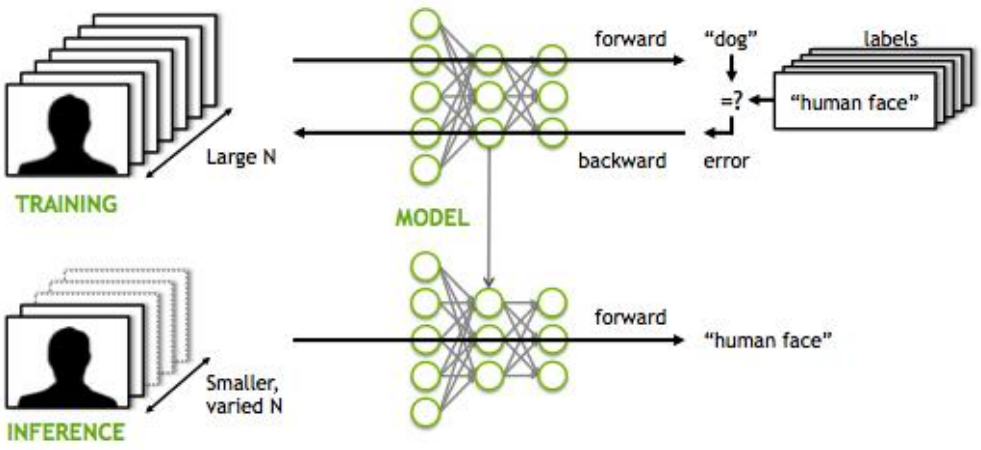
Atualmente as DNNs são utilizadas em diversos setores para uma ampla gama de tarefas que vão desde a previsão de dados à tarefas mais complexas como a classificação de imagens, reconhecimento de voz, robótica, etc. (SZE *et al.*, 2017a).

De acordo com Véstias *et al.* (2020), um modelo de rede neural é primeiro treinado para resolver um problema específico e, em seguida, usado para classificar novas amostras em um processo conhecido como inferência. O treinamento da rede determina sua acurácia. Este treinamento pode ser supervisionado ou não supervisionado. No treinamento supervisionado, o modelo aprende com os dados rotulados manualmente e os pesos são ajustados para que a saída da rede para uma entrada específica corresponda à resposta correta. Já o treinamento não supervisionado não requer dados rotulados. Em vez disso, o modelo extrai padrões da entrada por si só. Para as aplicações de visão computacional o treinamento mais utilizado é o supervisionado.

A Figura 3 ilustra as duas fases do processo de aprendizado profundo, treinamento e inferência. A fase de treinamento possui duas etapas: propagação direta e retropropagação. Na propagação direta, o dado de entrada é carregado na camada inicial e é propagado até que uma saída seja produzida. A saída é então comparada com o valor de saída esperado usando uma função de custo e o desempenho é avaliado. O mesmo processo é repetido por retropropagação, onde os dados são realimentados na camada inicial com o objetivo de reduzir a função custo alterando os pesos. Dessa forma, até que os pesos ótimos sejam calculados, o treinamento continua e o modelo

está pronto para inferência ou teste. Na fase de inferência há apenas propagação direta dos dados e a saída produzida é o resultado final da previsão (MUCHANDI, 2020).

Figura 3 – Comparação entre os processos de treinamento e inferência de uma rede neural profunda



Fonte – MITXPC (2022).

Existem diversos tipos de redes neurais e cada uma tem sua configuração e sua utilização. A maneira como os neurônios são organizados em uma camada e como eles estão conectados à camada sucessora definem o que é chamado arquitetura ou topologia da rede neural. Alguns exemplos são a rede neural *feed-forward*, rede neural recorrente (RNNs), rede neural convolucional (CNN), etc. Para este trabalho, a arquitetura de rede de interesse é a CNN devido a sua aplicação na visão computacional.

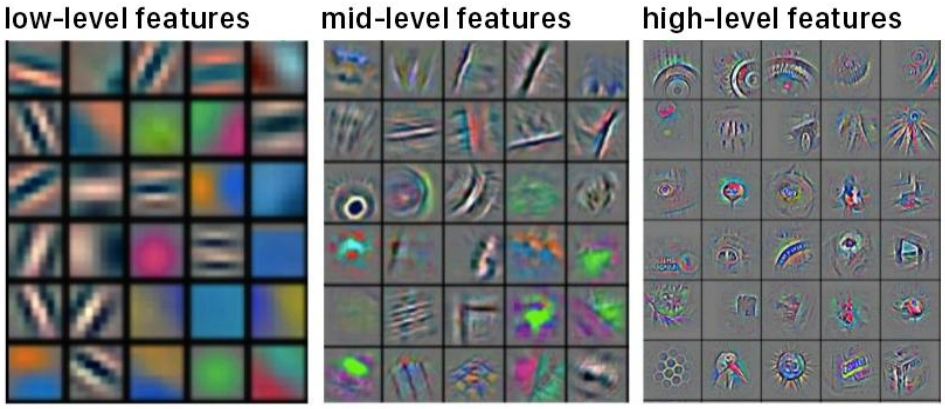
2.1.2 Redes Neurais Convolucionais

Com o rápido desenvolvimento da tecnologia de aprendizado profundo, a Rede Neural Convolucional (CNN, *Convolutional Neural Network*) tornou-se um dos métodos mais populares em reconhecimento de padrões e visão computacional (PATEL *et al.*, 2015). A CNN é um tipo de DNN que adiciona a operação de convolução em sua arquitetura. Essa operação pode ser vista como uma versão estendida de multiplicações de matrizes, que adiciona as propriedades de conectividade local e invariância de translação. Comparado com multiplicações de matrizes, em convoluções, cada elemento de entrada é substituído por um mapa de atributos e cada elemento de peso é substituído por um *kernel* convolucional (ou filtro). Então, o cálculo é baseado em janelas deslizantes (CHEN, Y. *et al.*, 2020).

Para as aplicações de visão computacional, as CNNs são compostas de múltiplas camadas convolucionais e, com cada camada, uma abstração em alto nível dos dados de entrada (chamado mapa de atributos ou *feature map*), é extraído para pre-

servar informações essenciais, porém únicas. Como apontado por Sze *et al.* (2017b), a saída das primeiras camadas convolucionais podem ser interpretadas como representando a presença de diferentes atributos de baixo nível na imagem, como linhas e bordas. Nas camadas subsequentes, esses atributos são então combinados em uma medida da provável presença de atributos de alto nível, por exemplo, as linhas são combinadas em formas. E, finalmente, com todas essas informações, a rede fornece uma probabilidade de que esses atributos de alto nível compõem um objeto ou cenário em particular. A Figura 4 ilustra como estes mapas de atributos podem se apresentar em diferentes camadas de um modelo CNN.

Figura 4 – Exemplos de mapas de atributos em diferentes camadas convolucionais de um modelo de CNN



Fonte – Bebes (2017).

De acordo com Goodfellow, Bengio e Courville (2016), para uma imagem 2D I como entrada e um kernel 2D K, a convolução discreta é dada pela Equação (1)

$$S[i,j] = (K * I)[i,j] = \sum_m \sum_n I[i-m, j-n]K[m,n] \tag{1}$$

Além da camada convolucional, as CNNs são compostas por outras camadas que realizam operações nos mapas de atributos como, por exemplo:

- **Camada de *Pooling***: Introduz invariância translacional no mapa de atributos substituindo os valores de ativações em uma janela por um valor representativo. O método de escolha deste valor pode ser por meio de uma média entre todos os valores dessa janela (*average-pooling*) ou a ativação com o maior valor da mesma (*max-pooling*) (GOODFELLOW; BENGIO; COURVILLE, 2016). O tamanho desta janela é o mesmo do tamanho do filtro de *pooling* e esse filtro desloca por uma certa distância (chamada de *stride*) para percorrer todo o mapa de atributos. Além disso, esta camada é normalmente adicionada ao longo das sucessivas camadas convolucionais e

também é empregada para reduzir progressivamente o tamanho espacial da representação, reduzindo assim a quantidade de parâmetros e computação na rede. (SHAWAHNA; SAIT; EL-MALEH, 2018);

- **Camada de *Batch Normalization*** (IOFFE; SZEGEDY, 2015): Esta operação desloca e dimensiona a distribuição de entrada para ter média zero e variância unitária. Normalmente é utilizada para reduzir o tempo de treinamento e melhorar a capacidade de generalização da rede (LIANG, Songhong *et al.*, 2020). Esta operação é dada pela Equação (2), onde x e y são entrada e saída, respectivamente, μ e σ são estatísticas coletadas sobre o conjunto de treinamento, γ e β são parâmetros treinados, e ε para evitar problemas de arredondamento. Durante a inferência, todos os parâmetros são fixos, portanto, é preciso apenas aplicar Equação (2) a cada ativação do respectivo mapa de atributos. Cada mapa de atributos requer seu próprio conjunto de parâmetros de *batch normalization*;

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \varepsilon}} \cdot \gamma + \beta \tag{2}$$

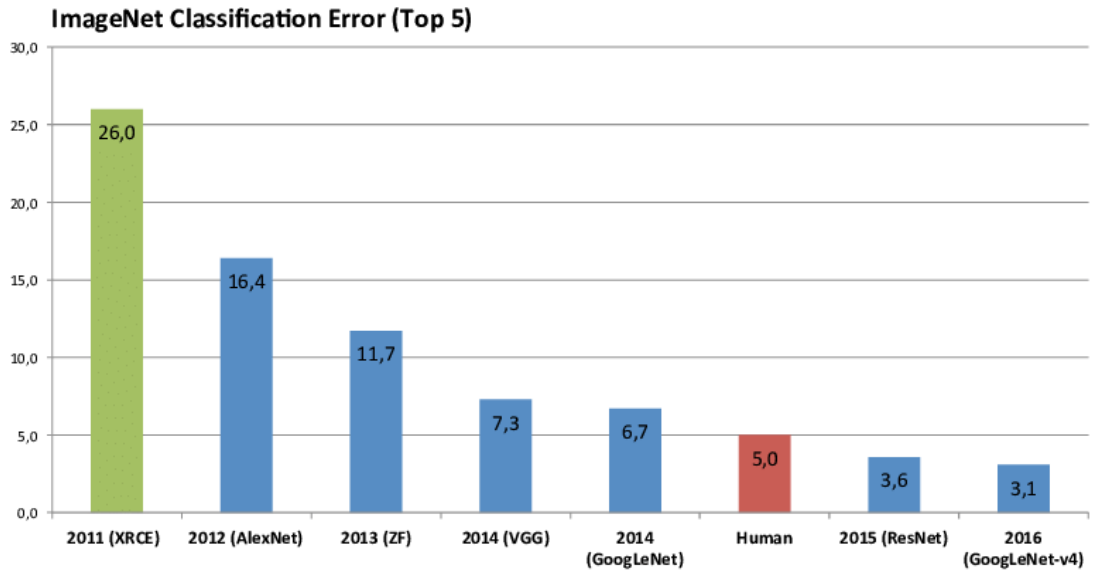
- **Camada Densa**: Geralmente vem no final da rede onde o mapa de atributos da camada anterior é achatado em um vetor. A partir disso, todas as ativações desse vetor são conectadas à próxima camada ou à camada de saída que é o vetor de predição, onde seu tamanho é o número de classes e os valores suas probabilidades. As camadas densas desempenham o papel da tarefa de classificação, enquanto que as camadas anteriores atuam na extração de atributos.

Embora as CNNs sejam conhecidas pelos pesquisadores há décadas (LECUN *et al.*, 1989), elas foram popularizadas após demonstrarem alta acurácia no desafio de reconhecimento ImageNet de 2012 (RUSSAKOVSKY *et al.*, 2015). ImageNet é um conjunto de dados amplamente utilizado para avaliar o desempenho de algoritmos de detecção de objetos e classificação de imagens que consiste em milhões de imagens diferentes distribuídas em dezenas de milhares de classes de objetos (RUSSAKOVSKY *et al.*, 2015). Além disso, para comparar o desempenho dos algoritmos de classificação, costuma-se utilizar as métricas *top*-1 e *top*-5, onde *top*-1 indica que a resposta do modelo (a com maior probabilidade) é exatamente a resposta esperada e o *top*-5 indica que a resposta esperada está entre as 5 respostas de maior probabilidade geradas pelo modelo.

A Figura 5 apresenta a evolução do desempenho dos modelos CNNs ao longo do tempo. Os vencedores do ano 2011 e anteriores utilizavam uma abordagem de visão computacional com métodos tradicionais, onde a funcionalidade de detecção era realizada através de atributos inseridos manualmente. Em 2012 houve o grande avanço através do modelo AlexNet (KRIZHEVSKY; SUTSKEVER; HINTON, G. E., 2012) em

que os atributos eram aprendidos através do aprendizado profundo. Nos anos seguintes, todos os vencedores estavam usando o aprendizado profundo para continuar melhorando significativamente o estado da arte. Entre estes estão os modelos ZF (ZEILER; FERGUS, 2014), VGG (SIMONYAN; ZISSERMAN, 2014) e GoogleLeNet (também chamado de InceptionV1) (SZEGEDY *et al.*, 2015) Em 2015, o modelo ResNet (HE *et al.*, 2016) conseguiu superar o desempenho considerado de nível humano no conjunto de dados ImageNet (RUSSAKOVSKY *et al.*, 2015). Subsequentemente, as CNNs tornaram-se o estado da arte para tarefas de classificação, detecção e localização de imagens. A pesquisa em CNNs e outras áreas de aprendizado profundo continua em ritmo acelerado, com centenas de novos artigos publicados a cada ano apresentando novos modelos e técnicas.

Figura 5 – Erro *top*-5 no conjunto de dados ImageNet ao longo dos anos.



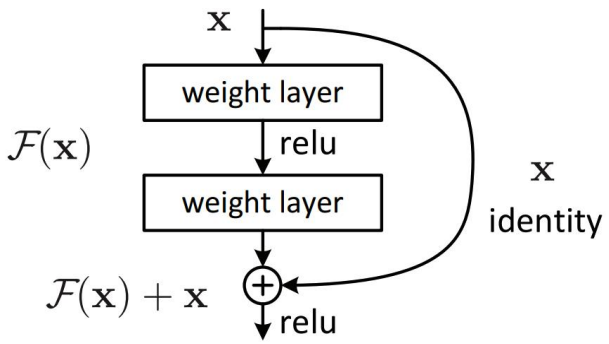
Fonte – Poel (2020).

O modelo ResNet adiciona em sua topologia os blocos residuais, ilustrado na Figura 6. Este bloco consiste em adicionar uma conexão de atalho (também chamada de conexão residual) em que após passar por uma ou mais camada convolucional, o resultado de saída é somado elemento a elemento com os valores anteriores ao bloco. Esta estratégia permitiu treinar redes extremamente profundas, com mais de 150 camadas.

2.1.3 Otimizações

Os modelos CNNs são computacionalmente caros e intensivos no uso de memória, dificultando assim seu uso no contexto de dispositivos com recursos escassos, como aqueles empregados na computação de borda. Portanto, muito esforço tem sido

Figura 6 – Bloco residual.



Fonte – He *et al.* (2016).

dedicado à definição de técnicas para compressão e aceleração de modelos CNNs sem afetar significativamente a acurácia dos mesmos (RUSSO *et al.*, 2021). Em geral, essas técnicas podem ser classificadas nas seguintes categorias principais:

- ***Pruning***: também chamada de poda, essas técnicas são baseadas na suposição de que muitos parâmetros em DNNs são desnecessários ou sem importância, sendo assim, os métodos de *pruning* são usados para remover estes parâmetros. Após uma fase de treinamento inicial, conexões sem importância são removidas, levando a uma rede esparsa. O impacto no desempenho quando estas técnicas são usadas está relacionado ao menor número de cálculos devido ao fato de vários parâmetros terem sido suprimidos. No entanto, como a rede podada é esparsa, o acesso irregular induzido à memória afeta negativamente a aceleração prática em plataformas de *hardware*. Na verdade, em alguns casos, o ganho de desempenho sobre a rede original não podada é muito limitada ou negativa, mesmo que a esparsidade da rede seja alta, maior que 95% (WEN *et al.*, 2016). Outro problema com as técnicas de poda é que elas exigem um retreinamento que, em alguns casos, pode exigir mais tempo do que o próprio treinamento. Por exemplo, em Han *et al.* (2015) é relatado que o AlexNet original levou 75 horas para treinar na GPU NVIDIA Titan X. Após a poda, toda a rede é treinada novamente com 1/100 da taxa de aprendizado inicial da rede original, exigindo 173 horas para retreinar a rede podada (RUSSO *et al.*, 2021);
- **Quantização**: A quantização reduz a largura de *bits* dos dados que fluem através do modelo de rede neural. Deste modo, portanto, é possível diminuir o tamanho do modelo para economizar memória e reduzir os requisitos de largura de banda para as memórias externa e interna. Além disso, este método resulta em circuitos aritméticos menos complexos, pois precisam

trabalhar em operandos com menor número de *bits*, resultando em uma redução do números de componentes lógicos necessários para cada unidade de operação de multiplicação e acumulação (MAC) além da quantidade de roteamento (DENG, L. *et al.*, 2020);

- **Decomposição matricial**: Operações com matrizes é a computação básica de uma rede neural (DENG, L. *et al.*, 2020). Portanto, métodos de fatoração são usados para aproximar a matriz de pesos de uma camada DNN com a multiplicação de duas ou múltiplas matrizes com um menor posto (SAINATH *et al.*, 2013). Este método pode ser aplicado tanto em camadas convolucionais quanto em camadas densas, mas seu uso prático não é tão fácil, pois envolve extensas operações de decomposição que são computacionalmente caras. Outra questão da fatoração é que ela requer um extenso retreinamento do modelo para alcançar a convergência quando comparado ao modelo original (RUSSO *et al.*, 2021);
- **Destilação de conhecimento**: Neste método é realizado o processo de tranferir o conhecimento de uma rede maior e mais complexa para um modelo menor, de modo que o modelo menor tenta imitar a função aprendida pelo modelo complexo (HINTON, G.; VINYALS; DEAN, Jeff *et al.*, 2015). Uma das desvantagens desta técnica é que ela só pode ser aplicada a tarefas que possuam função perda *softmax* (DENG, L. *et al.*, 2020).

2.1.4 Redes Neurais Binárias

A quantização é conhecida por ser uma das formas mais eficazes de reduzir a complexidade de inferência de rede neural, que consiste em reduzir a precisão do formato de dados utilizado para representar os pesos do modelo e as entradas/saídas das camadas (HUBARA *et al.*, 2017). A forma mais comum de quantização é utilizar inteiros de 8 *bits* para pesos e ativações, proporcionando redução de complexidade considerável com pouca queda de precisão em relação aos modelos de ponto flutuante (JACOB *et al.*, 2018). No entanto, o consumo de memória e o número total de operações para a inferência de modelos quantizados em 8 *bits* ainda são excessivos para dispositivos de computação de borda (DAGHERO *et al.*, 2021). Para alcançar uma redução ainda maior da complexidade, Courbariaux *et al.* (2016) foram os primeiros a apresentar o conceito de Redes Neurais Binárias.

As Redes Neurais Binárias (COURBARIAUX *et al.*, 2016) podem ser consideradas como uma versão de rede convolucional com quantização extrema, onde a precisão tanto dos pesos quanto das ativações são reduzidas para 1 bit e assim restringindo os valores possíveis para +1 e -1. O processo de compactar valores codificados em 32 *bits* para 1 *bit* é chamado de binarização e é obtido simplesmente com a função

sign(·), apresentada na Equação (3). A binarização de pesos e ativações afeta significativamente o uso de memória do modelo, reduzindo-a em 32x em relação a uma implementação de ponto flutuante de precisão simples (RASTEGARI *et al.*, 2016).

$$y = sign(x) = \begin{cases} +1 & \text{se } x \geq 0, \\ -1 & \text{caso contrário} \end{cases} \tag{3}$$

A função $sign(\cdot)$ possui derivada igual a 0 em quase todos os lugares, o que torna impossível calcular o gradiente utilizando a regra da cadeia na fase de treinamento. A fim de contornar este problema, uma prática amplamente aceita é usar uma função diferente durante a retropropagação que é diferenciável, mas muito parecida com a ativação binária (RAJ; NAYAK; KALYANI, 2020). A função mais utilizada é o *Straight-Through Estimator* (BENGIO; LÉONARD; COURVILLE, 2013), em que permite que apenas valores retropropagados em um determinado intervalo passem.

Segundo Yuan e Agaian (2021), o objetivo da binarização é de não apenas poder economizar no armazenamento de um modelo DNN, mas também reduzir os custos da computação de matrizes na sua inferência. Sendo assim, a operação de convolução pode ser aproximada pela binarização das matrizes dos operandos, como mostrado na Equação (4).

$$\mathbf{I} * \mathbf{W} \approx sign(\mathbf{I}) \circledast sign(\mathbf{W}) \tag{4}$$

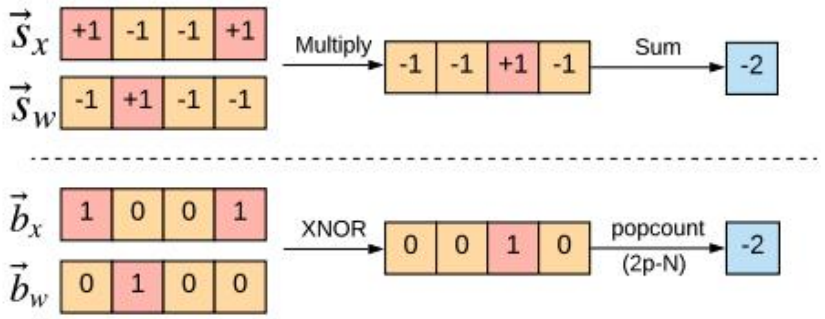
Como os valores após a binarização são +1 ou -1, a convolução denotada pelo símbolo $\circledast$ pode ser reduzida a operações *bit* a *bit*. Isto é, se os valores binários obtidos dessas operações forem codificados com +1 como um *bit* de valor um e -1 como um *bit* de valor zero, uma operação de multiplicação é equivalente a uma operação lógica XNOR nos valores binários. A soma das operações XNOR também pode ser calculada através de um simples *popcount*, que é a contagem do número de *bits* definido como valor 1. Esta equivalência entre as operações é ilustrada na Figura 7 e uma comparação visual entre as operações realizadas em uma rede neural com valores reais e uma rede neural binária é apresentada na Figura 8.

Devido a transformação dos valores {-1,+1} para {0,1}, o resultado final do produto escalar entre dois vetores binários de pesos (w) e entradas (x) é obtido com a Equação (5), onde $\odot$ é a operação de XNOR e N é o tamanho do vetor.

$$y = 2 \cdot popcount(w \odot x) - N \tag{5}$$

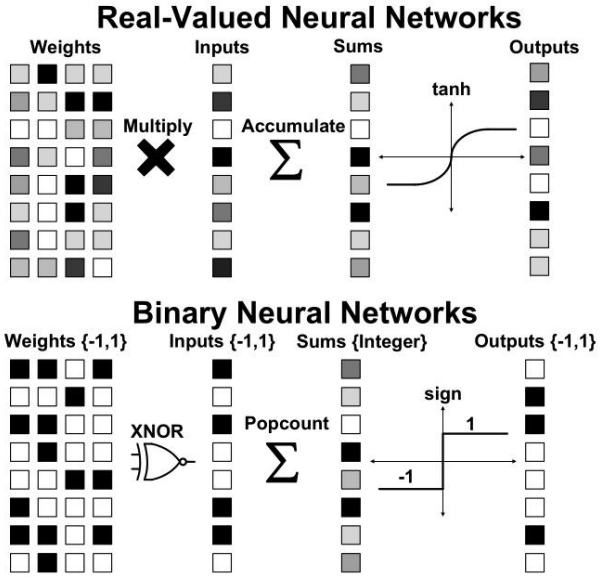
Desta forma, a binarização reduz os requisitos de armazenamento e largura de banda de memória e elimina completamente as operações de multiplicação e acumulação (MAC) presentes nas camadas de computação da rede neural por operações binárias que podem ser executadas com muita eficiência em um hardware dedicado, como um FPGA.

Figura 7 – Equivalência do produto escalar com as operações de multiplicação e acumulação (em cima) com as operações XNOR e popcount (em baixo)



Fonte – Ghasemzadeh, Samragh e Koushanfar (2018).

Figura 8 – Comparação de uma rede neural com valores reais e uma Rede Neural Binária



Fonte – Knag *et al.* (2020).

Apesar dos benefícios atraentes, é consideravelmente difícil treinar e otimizar redes fortemente quantizadas como as redes binárias para se obter um alto desempenho (ZHANG, Y.; ZHANG, Z.; LEW, 2021). As BNNs pioneiras costumavam sofrer de uma diferença de acurácia no *top*-1 do dataset *ImageNet* de mais de 20% em comparação com suas contrapartes de ponto flutuante (COURBARIAUX *et al.*, 2016). Apenas recentemente as BNNs se tornaram comparáveis em qualidade aos modelos populares como por exemplo o *ResNet* (HE *et al.*, 2016). Segundo Liu *et al.* (2021), um dos motivos para essa dificuldade é que as BNNs tendem a ter uma função perda

muito caótica e descontínua que torna sua otimização desafiadora. Além disso, há uma perda severa de informações devido à binarização dos parâmetros. Para que a binarização funcione, é preciso mudar muitas coisas em comparação com as práticas padrões de DNNs. Segundo Yuan e Agaian (2021), para reduzir essa diferença de acurácia em comparação com as CNNs de precisão total, uma variedade de novas soluções de otimização foram propostas nos últimos anos que incluem:

- Minimizar o erro de quantização.
- Melhorar a função perda.
- Melhor aproximação do gradiente.
- Novas estruturas de topologias de redes.
- Estratégias e truques para o treinamento.

Como mostrado na Figura 9, os modelos BNNs estão atingindo uma acurácia melhor a cada ano. No entanto, como notado por Bannink *et al.* (2021), embora as BNNs tenham o potencial de tornar as aplicações de aprendizado profundo radicalmente mais eficientes, na prática as redes que utilizam ponto flutuante ou com quantização de 8 *bits* ainda dominam os modelos de DNN em produção. Segundo Yuan e Agaian (2021), um dos motivos é que os principais *frameworks* para a implementação de modelos DNNs como o Tensorflow (ABADI *et al.*, 2016) e o Pytorch (PASZKE *et al.*, 2019) ainda não possuem suporte nativo para este tipo de rede. No entanto, na literatura, foram publicados diversos *frameworks* para a utilização de BNNs como BMXNet (YANG *et al.*, 2017), FINN (BLOTT *et al.*, 2018), Riptide (FROMM *et al.*, 2020) e Larq (BANNINK *et al.*, 2021). Larq[1] é um framework *open source* considerado o estado da arte para a implementação, treinamento, conversão e inferência de BNNs.

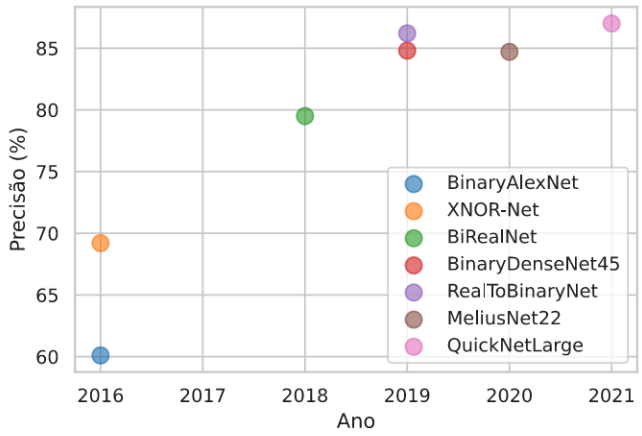
Além das vantagens de reduções do tamanho de armazenamento e da computação, as BNNs são mais eficientes em termos de energia pois a redução da complexidade do *hardware* resulta em menos componentes lógicos além de uma área de *chip* menor. Outro ponto é que as BNNs podem ser consideradas mais confiáveis, pois demonstraram ter maior robustez contra ataques adversariais do que as DNNs normais (GALLOWAY; TAYLOR; MOUSSA, 2017). Em vista disso, portanto, as BNNs são promissoras para uma implementação eficiente e precisa para a computação de borda.

2.1.5 Recursos Arquiteturais

A Unidade de Processamento Gráfico (GPU, *Graphics Processing Unit*) é a plataforma mais amplamente utilizada para implementar CNNs, pois oferece o melhor desempenho em termos de rendimento computacional puro. No entanto, GPUs possuem um gasto energético elevado e existem outras plataformas que oferecem um

[1] https://github.com/larq/larq

Figura 9 – Acurácia *top*-5 de modelos BNNs nos últimos anos para o conjunto de dados ImageNet.



Fonte – Autor.

gasto energético menor, como o Circuito Integrado de Aplicação Específica (ASIC, *Application Specific Integrated Circuit*) ou Arranjo de Porta Programável em Campo (FPGA, *Field Programmable Gate Array*). Com um projeto de *hardware* orientado para rede neural, estas plataformas podem implementar um alto paralelismo e fazer uso das propriedades de computação da rede neural para remover lógica adicional resultando em um processamento com um menor gasto energético. Como multiplicações e convoluções de matrizes dominam mais de 90% das operações de uma CNN, estes são os principais alvos dos projetos de aceleradores para CNN (CHEN, Y. *et al.*, 2020).

O ASIC fornece o melhor desempenho por valores de energia, pois pode ser estritamente adaptado para uma determinada aplicação. No entanto, ele não tem a flexibilidade de plataformas de computação de uso geral, como a Unidade Central de Processamento (CPU, *Central Processing Unit*) e GPU, e, portanto, não conseguem se adaptar a algoritmos de aprendizado profundo que mudam rapidamente.

Enquanto a implementação de um acelerador ASIC supera os aceleradores FPGAs em desempenho, o acelerador em FPGA oferece maior flexibilidade, limites de desenvolvimento mais baixos e menos ciclos de desenvolvimento. Além disso, é possível desenvolver um projeto para FPGA e posteriormente convertê-lo para ASIC (ON SEMICONDUCTOR, 2021). Portanto, o FPGA é a plataforma mais utilizada para desenvolvimento e testes de aceleradores de CNN.

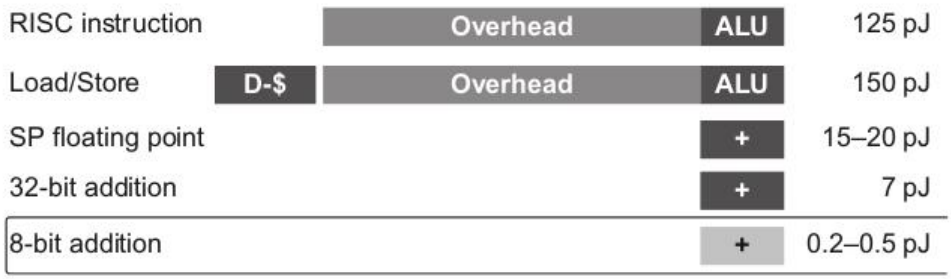
2.2 ACELERADORES DE DOMÍNIO ESPECÍFICO

2.2.1 Contextualização e Definição

A maior parte da computação hoje em dia é realizada programando-se para processadores de propósito geral, ou CPUs. Segundo William J. Dally, Turakhia e Han (2020), o grande atrativo das CPUs se deve ao fato de que são fáceis de programar além de que existe uma ampla base de códigos para elas. No entanto, segundo Qadeer *et al.* (2015) os processadores podem ser considerados como uma máquina ineficiente.

A baixa eficiência dos processadores de uso geral é explicada na Figura 10, que compara o gasto energético de várias operações aritméticas com o gasto energético total de uma instrução para um processador RISC extremamente simples. A dissipação de energia das operações aritméticas que realizam o trabalho útil em uma computação permanece muito menor do que a energia desperdiçada nos *overheads* de instrução, como busca de instruções, decodificação, gerenciamento de *pipeline*, sequenciamento de programas, etc. (QADEER *et al.*, 2015). Dado esse *overhead*, pequenas mudanças nos núcleos existentes podem nos trazer melhorias de até 10%, mas se quisermos melhorias de maior ordem de grandeza ao mesmo tempo em que oferecemos programabilidade, precisamos aumentar o número de operações aritméticas por instrução de um para centenas (HENNESSY; PATTERSON, 2017).

Figura 10 – Comparação do custo energético de instruções para um processador RISC com o custo de operações aritméticas.



Fonte – Qadeer *et al.* (2015).

Outro ponto importante é que a lei de Moore (MOORE *et al.*, 1965), que dita que o número de transistores de um *chip* de silício dobra a cada 2 anos, chegou ao fim (THEIS; WONG, 2017). Como consequência, será necessário mais especialização do processador para obter melhorias de desempenho. Além disso, aplicações emergentes, como inteligência artificial, estão exigindo desempenho computacional pesado que não pode ser atendido por arquiteturas convencionais. Segundo Urquhart (2021), para uma tarefa fixa ou uma gama limitada de tarefas, o dimensionamento de energia funciona

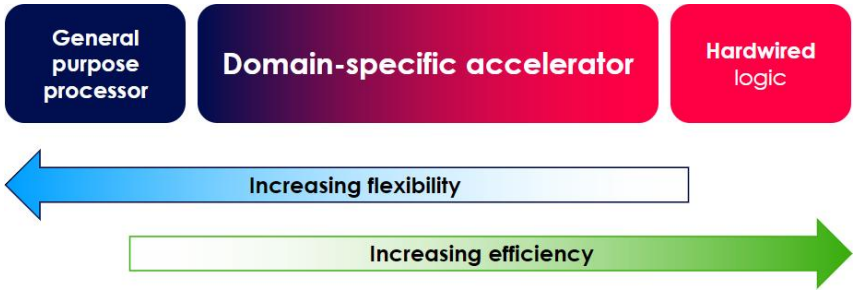
melhor do que para uma ampla gama de tarefas. Em vista destes problemas, para continuar escalando desempenho e eficiência devemos procurar arquiteturas alternativas, como os aceleradores de domínio específico (DALLY, William J.; TURAKHIA; HAN, 2020).

Um acelerador de domínio específico é um processador ou conjunto de processadores que são otimizados para realizar tarefas específicas. Eles são adaptados para atender às necessidades dos algoritmos exigidos para seu domínio. Por exemplo, no processamento de áudio, um processador pode ter um conjunto de instruções para implementar de forma otimizada algoritmos para cancelamento de eco. Em outro exemplo, um acelerador de DNN pode ter a operação de multiplicação e acumulação em um conjunto de elementos simultaneamente a fim de realizar operações de matriz com eficiência (URQUHART, 2021). Segundo Shepard (2021), inicialmente, os aceleradores de domínio específico foram desenvolvidos para reduzir o tempo de execução de tarefas particularmente intensivas em computação e, mais recentemente, os aceleradores estão sendo adotados para reduzir o consumo de energia necessário para executar determinadas tarefas. Além disso, o autor destaca que, mesmo que o consumo de energia de pico do acelerador seja maior que o consumo de energia de pico consumido por um processador de uso geral, o tempo de execução da tarefa é reduzido em uma quantidade tão grande que a energia total necessária para concluí-la é menor. Aceleradores foram projetados para aplicações em diferentes domínios, como processamento de imagens e aplicações de visão embarcada (SUN *et al.*, 2018), criptografia (WU; WEAVER; AUSTIN, 2001), aprendizagem profunda (CHEN, Y.-H. *et al.*, 2016), processamento de grafos (HAM *et al.*, 2016), banco de dados (WU *et al.*, 2014), etc. Como ilustrado na Figura 11, os aceleradores podem variar consideravelmente em sua especialização. Enquanto alguns aceleradores podem ser similares ou derivados de um núcleo de processador embarcado, outros podem ter uma programação limitada e parecer mais perto de um *hardware* com lógica fixa. Aceleradores mais especializados serão mais eficientes em termos de área de silício e consumo de energia. De acordo com Yu-Ting Chen *et al.* (2013), arquiteturas com aceleradores podem trazer uma eficiência energética de 10 a 100 vezes ao transferir a computação de um CPU de uso geral para aceleradores de domínio específico.

Segundo William J. Dally, Turakhia e Han (2020), os aceleradores de domínio específico exploram quatro técnicas diferentes para ganhos de desempenho e eficiência energética, são estes:

- **Especialização de dados**: Operações especializadas em tipos de dados específicos de um determinado domínio podem fazer em um ciclo o que poderia levar dezenas de ciclos em uma unidade de computação convencional. O laço interno de execução do algoritmo de muitas aplicações exigentes realizam de dezenas a centenas de operações aritméticas e lógicas utilizando

Figura 11 – Comparação da especialização de um acelerador de domínio específico



Fonte – CODASIP (2022)

apenas referências de memória local. Em muitos casos, a lógica especializada pode executar todo o laço interno em um único ciclo com uma pequena quantidade de área e potência.

- **Paralelismo**: Altos graus de paralelismo, frequentemente explorados em diversos níveis, proporcionando ganhos em desempenho. Para serem eficazes, as unidades paralelas devem explorar a localidade e fazer poucas referências de memória global ou seu desempenho será limitado à memória.
- **Memória local e otimizada**: Ao armazenar as principais estruturas de dados em várias pequenas memórias locais, uma largura de banda de memória muito alta pode ser alcançada com baixo custo e baixo consumo energético. Os padrões de acesso à memória global podem ser otimizados para alcançar a maior largura de banda de memória possível. Além disso, as principais estruturas de dados utilizadas na aplicação podem ser compactadas para multiplicar a largura de banda. Os acessos à memória podem ser balanceados em todos os canais de memória e cuidadosamente programados para maximizar a sua utilização.
- **Redução de *overhead***: A especialização de *hardware* elimina ou reduz o *overhead* de interpretação do programa. De acordo com William J Dally *et al.* (2008), em um processador com execução "em ordem"gasta mais de 90% de sua energia em *overheads* como busca de instrução, decodificação de instrução, controle, etc.

William J. Dally, Turakhia e Han (2020) pontuam que para obter uma aceleração elevada e ganhos de eficiência de um *hardware* especializado geralmente é necessário a modificação do algoritmo subjacente. Os autores explicam que, como os algoritmos existentes são altamente ajustados para processadores convencionais de uso geral, eles raramente são a abordagem ideal para uma solução especializada. Em vez disso, o algoritmo e o *hardware* devem ser projetados em conjunto para otimizar simultane-

amente o desempenho e a eficiência, preservando ou aprimorando a acurácia. Para elucidar este ponto, William J. Dally, Turakhia e Han (2020) se referem ao trabalho de Han, Mao e William J Dally (2015) que mostrou como as redes neurais podem utilizar da otimização de *pruning* para serem compactadas até 30 vezes. Eles concluem que o *overhead* de se utilizar métodos esparsos como o *pruning* em *hardware* convencional tornava estes algoritmos desinteressantes, exceto pela compactação de memória. No entanto, o projeto em conjunto de um *hardware* para a aceleração das operações esparsas permite que esses algoritmos também sejam usados para reduzir a computação.

Segundo Patterson e Hennessy (2020), assim como o campo da computação mudou o uso de processadores com um único núcleo para processadores com diversos núcleos na última década devido à necessidade de escalar o desempenho, os projetistas de sistemas computacionais de agora estão trabalhando em aceleradores de domínio específico. Segundo os autores, o novo normal será que um computador consistirá em processadores padrões de uso geral para executar os programas convencionais e maiores, como sistemas operacionais, juntamente com processadores e aceleradores de domínio específico que executam apenas uma pequena gama de tarefas, mas que as executam extremamente bem. Desta forma, os autores preveem que os computadores serão muito mais heterogêneos do que os *chips* com processador multinúcleos do passado.

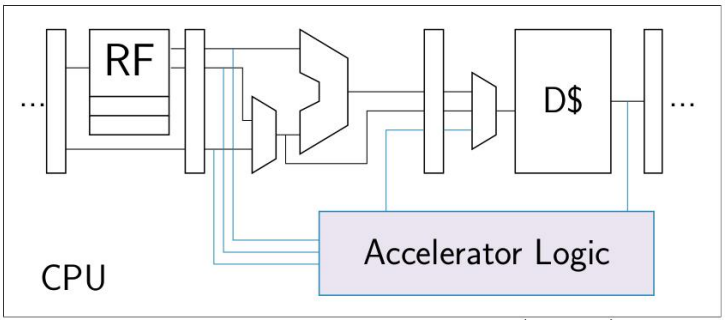
2.2.2 Modelos de Aceleradores

A escolha do tipo de integração com o processador tem um efeito fundamental nas considerações do projeto do acelerador e vice-versa. Os aceleradores são amplamente diferenciados em fortemente acoplados (do inglês *tightly-coupled*) e fracamente acoplados (do inglês *loosely-coupled*) com base em sua integração com o processador (MUCHANDI, 2020).

A Figura 12 apresenta um modelo de acelerador fortemente acoplado. Este modelo de acelerador consiste em uma ou mais unidades funcionais de *hardware* especializadas projetadas para acelerar partes críticas de uma aplicação; por exemplo, o *loop* interno para um determinado algoritmo ou uma sequência de funções trigonométricas. Este tipo de acelerador está localizado dentro ou muito próximo do núcleo de processamento e portanto são incluídos no datapath da CPU como coprocessador. Desta maneira, para acessá-los é necessário expandir o Conjunto de Instruções (ISA, *Instruction Set Archtecture*) para incluir instruções especiais para gerenciar sua operação. Além disso, este modelo compartilha com o núcleo de processamento os recursos chaves como banco de registradores, *cache* de dados, etc. e, portanto, paralisa a execução do pipeline até que o acelerador complete a sua execução. A vantagem deste modelo é que possui um *overhead* de chamada nula. No entanto, eles podem

complicar ainda mais o projeto da CPU e impor desafios de sincronização para atender as restrições de frequência de clock que são definidas para a CPU. Além disso, eles possuem portabilidade limitada entre diferentes projetos de sistemas, pois é muitas vezes necessário adaptar a interface do acelerador com a CPU (COTA *et al.*, 2015).

Figura 12 – Acelerador fortemente acoplado

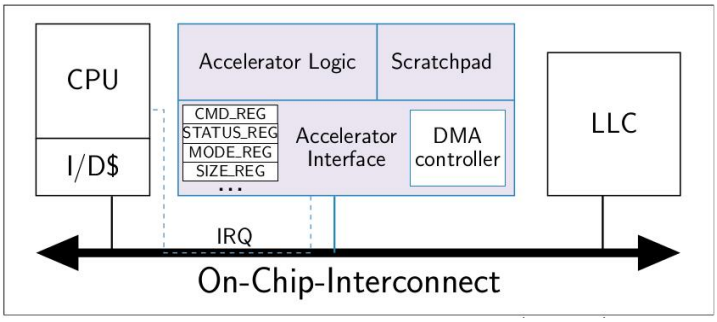


Fonte – Cota *et al.* (2015).

A Figura 13 apresenta um modelo de acelerador fracamente acoplado. Neste modelo, o acelerador é localizado fora do núcleo da CPU e interage com a mesma através do barramento de interconexão do *chip*. Estar fora do núcleo oferece ao acelerador com esse modelo um orçamento de área maior do que os aceleradores fortemente acoplados, pois neste caso ele não degrada o desempenho do *pipeline* do processador ou o tempo de acesso à *cache*. Isso permite blocos lógicos maiores e com caminhos de dados mais complexos que podem implementar e acelerar uma determinada aplicação por completo, como, por exemplo, uma transformada rápida de Fourier ou um algoritmo de codificação de imagem completo. Além disso, estar fora do núcleo da CPU permite implementar memórias locais privadas, também conhecidas como *scratchpads* (BANAKAR *et al.*, 2002), que armazenam os dados de entrada a serem processados, resultados temporários e os dados de saída a serem escritos de volta na memória (COTA *et al.*, 2015).

Enquanto que os aceleradores fortemente acoplados são restritos no pipeline de uma CPU e, portanto, experimentam benefícios limitados de uma personalização completa, os aceleradores fracamente acoplados ignoram completamente os *overheads* de CPU (instruções, *caches*, etc.) e podem ser otimizados com uma maior liberdade (CHEN, Y.-T. *et al.*, 2013). Cota *et al.* (2015) analisaram os efeitos dos *overhead* de comunicação interna do chip e chamadas do acelerador em diferentes projetos de aceleradores com base no acoplamento com o processador e na comunicação com a memória. Os autores apresentaram uma comparação quantitativa entre projetos de aceleradores de alto desempenho para tarefas como Transformada Rápida de Fourier, encriptação AES, etc. aplicados em diversos modelos de acoplamento como forte-

Figura 13 – Acelerador fracamente acoplado



Fonte – Cota *et al.* (2015).

mente acoplado com uma CPU, fracamente acoplado com DMA para a cache de último nível e fracamente acoplado com DMA para a memória DRAM. Por fim, os autores concluíram que para cargas de trabalho com tamanhos de dados não triviais a melhor configuração é de aceleradores fracamente acoplados com blocos de memória privados personalizados para a determinada tarefa.

2.2.3 Aceleradores de DNN

Além das melhorias de algoritmos das redes neurais e otimizações como especificados na Seção 2.1.3, o desempenho de execução dessas redes depende muito da capacidade de computação do *hardware*. Como mencionado na Seção 2.1.5, os processadores de uso geral, como GPUs, atuam como o pilar da era do aprendizado profundo, especialmente no lado da nuvem. No entanto, para a computação de borda, a disponibilidade de recursos e energia geralmente é muito limitada de modo que minimizar a latência, energia e área tornou-se uma preocupação de *design* inevitável. Isso motiva o estudo de aceleradores especializados desenvolvidos para o domínio das redes neurais. Ao sacrificar a flexibilidade até certo ponto, esses aceleradores se concentram em padrões específicos das redes neurais para obter um desempenho satisfatório por meio da otimização da arquitetura de processamento, hierarquia de memória e mapeamento de fluxo de dados (DENG, L. *et al.*, 2020).

Como ilustrado na Figura 14, os aceleradores de DNNs podem ser divididos fundamentalmente em duas categorias baseado na forma em como executam a rede neural: aceleradores de streaming e aceleradores de camadas. Uma comparação entre estas duas arquiteturas é explicado a seguir:

- **Aceleradores de *streaming***: Aceleradores com essa arquitetura são customizados para uma topologia de rede neural específica onde é instanciado uma unidade de computação para cada camada (KROES *et al.*, 2020). Como é implementado um *hardware* otimizado para cada camada, esse tipo de ar-

quitetura geralmente oferece uma latência melhor, no entanto, requerem mais recursos do que os aceleradores de camadas. Além disso, apenas suportam a topologia de rede neural previamente definida antes da implementação. Portanto, se uma rede é otimizada e implementada para uma aplicação específica, ela precisa ser alterada estruturalmente e reprojetada para então ser utilizada em outras aplicações (CHO *et al.*, 2021). Como notado por Kroes *et al.* (2020), é um grande desafio alcançar alta acurácia com este tipo de arquitetura pois os modelos de redes convolucionais do estado da arte aumentam significativamente a quantidade de unidades computacionais necessárias para implementar a topologia da rede inteira, como também necessitam de um tamanho de memória interna maior para armazenar os pesos.

- **Aceleradores de camadas**: Aceleradores de camadas são projetados para manipular uma determinada camada por vez da topologia de rede neural. Além disso, como um acelerador com este tipo de arquitetura precisa ser capaz de lidar com vários tipos de camadas diferentes (diferentes tamanhos de mapas de atributos, diferentes números filtros, etc), a maioria destes aceleradores são projetados com arquiteturas reconfiguráveis que podem lidar com várias topologias diferentes de redes. Portanto, como apontado por Cho *et al.* (2021), este tipo de arquitetura pode ser mais adequado para aplicações com recursos limitados como na computação.

2.3 PLATAFORMAS E COMPONENTES

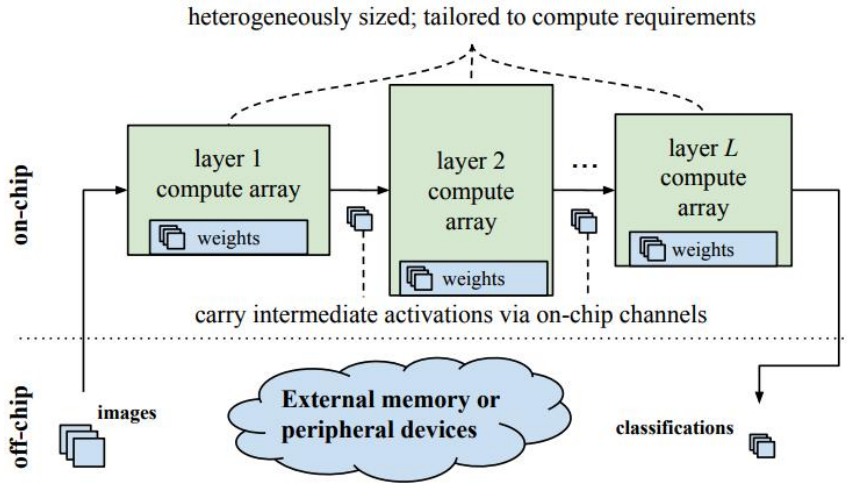
Esta seção apresenta conceitos adicionais que compõem o trabalho.

2.3.1 FPGA

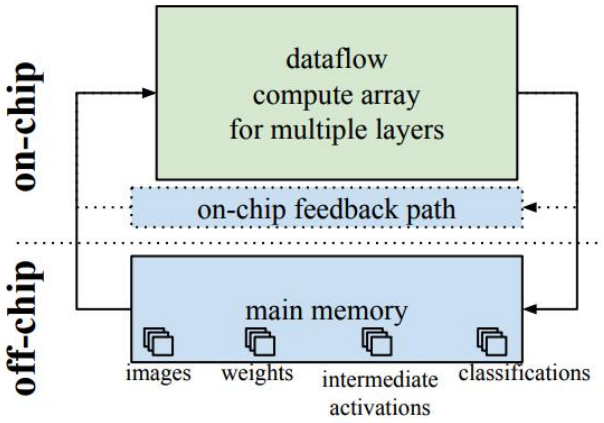
Os FPGAs são dispositivos programáveis que fornecem uma plataforma flexível para implementar funcionalidades de *hardware* personalizadas a um baixo custo de desenvolvimento. Eles consistem principalmente de um conjunto de células lógicas programáveis, chamadas de blocos lógicos configuráveis (CLBs), uma rede de interconexão programável e um conjunto de células programáveis de entrada e saída ao redor do dispositivo (VILLASENOR; MANGIONE-SMITH, 1997). Além disso, eles têm um rico conjunto de componentes incorporados, como blocos de processamento de sinal digital (DSP) que são usados para executar operações aritméticas intensivas, como multiplicar e acumular, *Block RAMs (BRAMs)*, *Look-Up Tables (LUTs)* , *Flip-Flops* (FFs), unidade de gerenciamento de clock e outros.

Em um FPGA, a porta lógica XNOR pode ser implementada como uma LUT. A LUT é um dos principais recursos em um FPGA moderno e, através dessas LUTs,

Figura 14 – Comparação entre os tipos de aceleradores para DNN



(a) Acelerador de *streaming*



(b) Acelerador de camadas

Fonte – Blott *et al.* (2018).

mesmo dispositivos FPGA pequenos podem realizar até trilhões de operações XNOR em um segundo. Além disso, em comparação com CNNs de ponto flutuante ou quantizadas, os pesos binários em BNNs gastam menos espaço de memória para armazená-los. Isso significa que estes pesos podem ser armazenados na memória interna destes dispositivos, a BRAM, onde é alcançado uma velocidade de acesso muito mais alta, bem como menor consumo de energia.

Sendo assim, FPGAs são adequados para BNNs, pois seus cálculos dominantes são operações lógicas *bit* a *bit* e seus requisitos de memória são bastante reduzidos.

2.3.2 RISC-V

O RISC-V (WATERMAN *et al.*, 2014) é uma arquitetura que começou a ser desenvolvida pela Universidade de Berkeley em 2010. Atualmente, o RISC-V é um ISA

modular, ampliável através de extensões, aberta e *royalty-free*. Sua especificação está inteiramente disponível e seu objetivo principal é servir de base para que empresas, instituições, ou qualquer outra entidade possa desenvolver não somente periféricos compatíveis mas também seus próprios núcleos, SoCs e aceleradores.

Por ser projetado de forma modular, o RISC-V permite que grupos de instruções sejam utilizados conforme necessário. Isto resulta em uma implementação que utiliza exatamente o que a aplicação precisa, sem a necessidade de desperdiçar área e energia que não seriam utilizadas. Além disso, possui um grupo especial de instruções em que desenvolvedores podem adicionar qualquer instrução necessária para a aplicação que desejam acelerar, sem quebrar a compatibilidade de software.

3 TRABALHOS RELACIONADOS

A partir de uma pesquisa exploratória, foram levantados diversos trabalhos que desenvolveram uma arquitetura para realizar a aceleração de BNNs. Com a taxonomia apresentada na Seção 2.2.3 foi possível filtrar ainda mais os trabalhos de interesse.

Os primeiros aceleradores de BNNs geralmente possuíam a arquitetura de *streaming* e eram validados com topologias de redes consideradas pequenas, onde a resolução da imagem de entrada é, como por exemplo, 32x32 *pixels*. Isso se deve ao fato de que este é o método que possui menos *overhead* e menor latência. No entanto, esta arquitetura limita o uso de aplicações, pois o tamanho da topologia de rede suportada também é limitada.

O acelerador de *streaming* FINN (UMUROGLU *et al.*, 2017) apresentou um resultado para topologia AlexNet, mas, como previsto, é necessário utilizar FPGA de alto desempenho com grande quantidade de componentes lógicos e memória, inviabilizando seu uso na computação de borda.

O acelerador LP-BNN (GENG *et al.*, 2019) aprimora a arquitetura de FINN para dar suporte às camadas residuais presentes na rede ResNet e propõe uma técnica de balanceamento de carga de trabalho entre o paralelismo das camadas para alcançar a máxima utilização de *hardware*. Porém, mesmo com as otimizações, os autores reportam o uso de 509K LUTs de um Xilinx Kintex KCU1500 portanto, o uso de recursos do FPGA ainda é exorbitante e inviável na computação de borda. Sendo assim, limitou-se a análise para arquiteturas de aceleradores de camadas.

FBNA (GUO *et al.*, 2018) foi a primeira arquitetura de acelerador de rede neural totalmente binarizada. Este acelerador utiliza memórias locais para o armazenamento dos mapas de atributos de entrada, pesos e mapas de atributos de saída. Assim, é realizada a busca dos dados de entrada e pesos, realizada as operações em seu *datapath* e o resultado da acumulação escrito na memória local de saída. Os autores avaliam o acelerador para as topologias beseadas em CIFAR10 e SVHN, que possuem dimensões de entrada 32x32.

Shuang Liang *et al.* (2018) apresentaram o acelerador FP-BNN, onde foi reportado resultado para um modelo de rede mais complexa, a AlexNet. Os dados de entrada são carregados da memória externa para uma memória interna e então é realizado as operações utilizandos vários *datapaths* para o processamento de dados em paralelo. Apesar do ganho de desempenho pela paralelização, a quantidade de recursos utilizados ainda é muito alta para a implementação na computação de borda visto que, para o modelo AlexNet, o acelerador requer mais de 230K ALMs de um FPGA Altera Stratix-V.

Conti, Schiavone e Benini (2018) divulgaram o acelerador XNE. Este é um acelerador configurável em que os autores validaram em um sistema baseado em micro-

controlador. Sua arquitetura é composta por um controlador que pode ser configurado através do mapeamento de memória utilizando protocolo AMBA APB. Além disso, Em vez de depender de um processador externo para calcular os deslocamentos para acesso à memória e iterações da execução do laço interno, os autores incluíram um processador de microcódigo que utiliza uma pequena ISA personalizada e suas instruções são carregadas através do mapeamento de memória. Este trabalho demonstrou o suporte a cenários de trabalho complicados e para aplicações práticas, no entanto, apenas avaliado para dispositivo ASIC.

Cho *et al.* (2021) estendem o trabalho de Conti, Schiavone e Benini (2018) e modificam o agendamento de dados das primeiras camadas para extrair um maior paralelismo. Desse modo, os autores alcançaram uma taxa de transferência e eficiência energética maior. Além disso, os autores reportaram a implementação em FPGA Xilinx onde é utilizado apenas 4,8K LUTs.

A Tabela 1 sintetiza as informações levantadas destes aceleradores selecionados. Em relação às topologias de redes utilizadas, percebe-se que os aceleradores são predominantemente avaliados utilizando modelos de BNNs que não refletem o estado da arte. Como mencionado anteriormente, as BNNs recentes utilizam técnicas que permitem alcançar uma acurária maior. Sendo assim, este trabalho visa utilizar conceitos destas arquiteturas e modificá-los para sua utilização com topologias de BNN modernas em FPGA.

Tabela 1 – Comparação entre trabalhos relacionados.

Trabalho	**Camadas**	**Topologias**	**Plataforma**
FBNA (GUO *et al.*, 2018)	BC, BN, P, FC	CIFAR10, SVHN	FPGA
FP-BNN (LIANG, Shuang *et al.*, 2018)	BC, BN, P, FC	MNIST, CIFAR10, AlexNet	FPGA
XNE (CONTI; SCHIAVONE; BENINI, 2018)	BC, BN, FC	VGG, ResNet	ASIC
Cho *et al.* (2021)	BC, BN, P, FC	CIFAR10, VGG	FPGA

BC=Convolução Binária, BN=*Batch Normalization*, P=*Pooling*, FC=Camada Densa (*Fully Connected*)
Fonte – Autor.

4 PROJETO E IMPLEMENTAÇÃO

A metodologia empregada consistiu primeiramente em realizar uma análise nas topologias de BNNs do estado da arte e levantar especificações que delimitam a arquitetura e funcionamento do acelerador projetado. Após essa investigação, são definidos conceitos importantes como a organização dos dados e o algoritmo para convolução binária. Por fim, é apresentado a microarquitetura proposta e seus componentes.

4.1 ANÁLISE BNNS

4.1.1 Topologia

A topologia da BNN a ser executada influencia diretamente no projeto do acelerador. Certos parâmetros da rede podem inviabilizar algumas soluções enquanto pode permitir outras soluções mais otimizadas. Em vista disso, como uma das motivações é executar BNNs do estado da arte, inicialmente, foram analisadas as topologias das seguintes BNNs: *QuickNet* (BANNINK *et al.*, 2021) e *BinaryResNetE18* (BETHGE *et al.*, 2019). A Figura 15 apresenta topologia da rede *QuickNet*. Devido à semelhança estre estas duas redes, apenas uma delas (a rede *Quicknet*) será detalhada e utilizada como base para a implementação deste trabalho. A área delimitada pela linha tracejada indica que a sequência de componentes no seu interior se repete pela quantidade de vezes indicada no lado direito. Para a operação de convolução binária (denominada "BinConv"), é implicitamente realizada a operação de binarização dos dados seguindo a Equação (3).

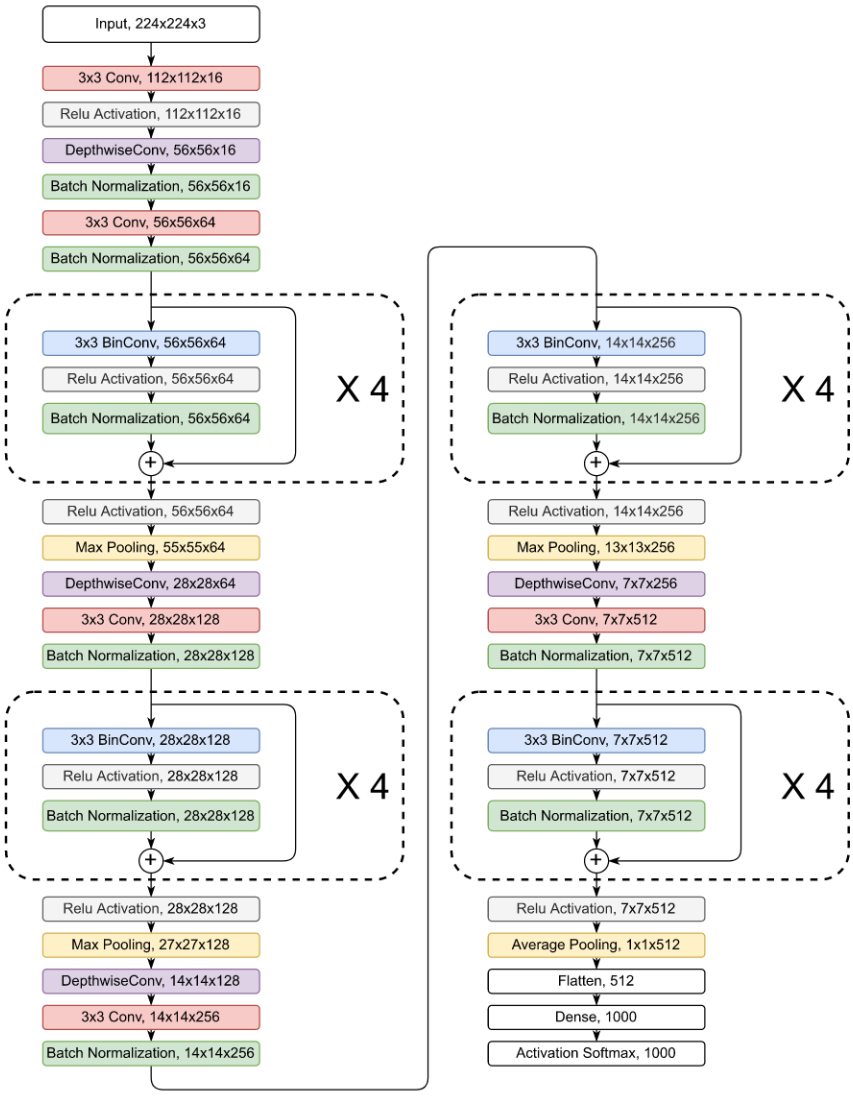
A partir da análise realizada foram levantados os seguintes pontos:

- É utilizado exclusivamente um filtro de convolução com a dimensão 3x3;
- O bloco residual é predominantemente utilizado para as camadas que utilizam convolução binária;
- É empregado o *padding* em todas as convoluções binárias.

Segundo Liu *et al.* (2018), o bloco residual aumenta a capacidade de representação da rede binária e a ajuda a atingir uma acurácia maior. Como exemplificado pela Figura 16, as conexões residuais permitem a propagação de atributos para as camadas mais profundas da rede e assim, reduzindo o erro decorrente pela binarização.

De acordo com Liu *et al.* (2018) e também reforçado por Bannink *et al.* (2021), o custo computacional de realizar a adição em atributos com precisão total decorrentes da conexão residual é baixo enquanto a acurácia melhora drasticamente. Portanto, este componente se tornou algo indispensável nas BNNs do estado da arte. Apesar dessa vantagem, uma consequência de se utilizar o bloco residual com a convolução binária é de que a saída da operação não pode ser diretamente binarizada, para permitir a

Figura 15 – Topologia da rede *QuickNet*.



Fonte – Autor.

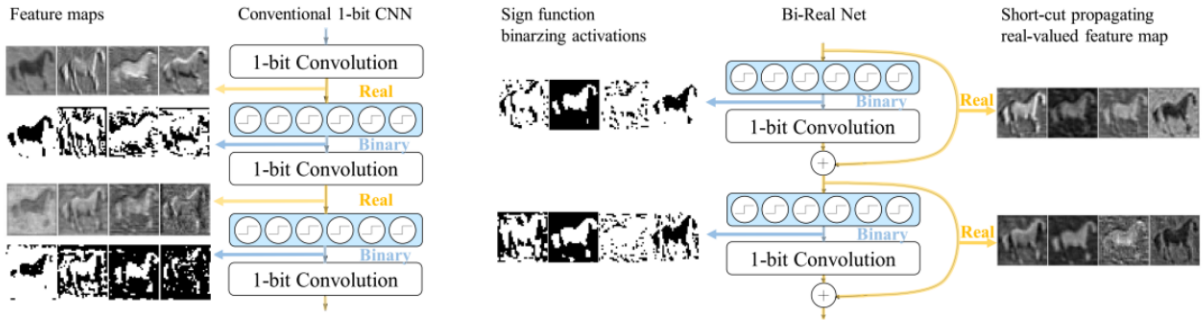
soma com os valores advindos da conexão residual. Como consequência, a saída do acelerador deve ser um valor inteiro para permitir essa soma posterior.

A Figura 17 apresenta a área de atuação do acelerador implementado em comparação ao bloco residual da rede. O acelerador obtém os mapas de atributos já binarizados como valor de entrada e realiza a convolução dos mesmos com os pesos treinados, que são previamente binarizados. O resultado da convolução gera um valor inteiro ao se utilizar a operação de *popcount* (Equação (5)).

4.1.2 Memória

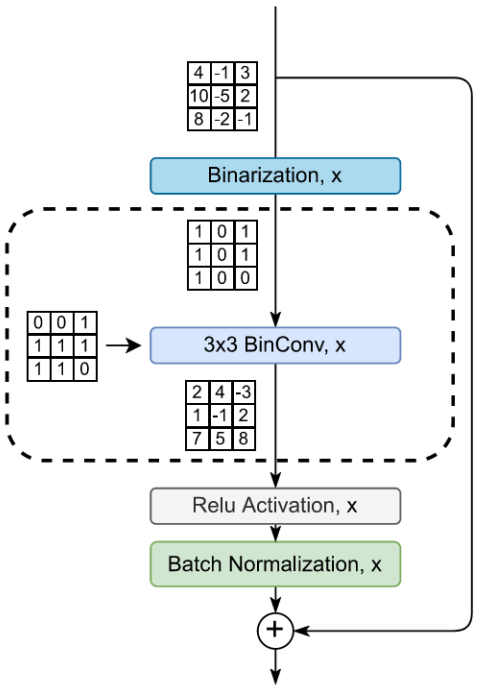
Foi realizada uma análise no uso de memória da rede *QuickNet*. A Tabela 2 apresenta o uso de memória do mapa de atributos de entrada e os respectivos pesos

Figura 16 – Influência da inserção de conexões residuais em uma rede binária.



Fonte – Liu *et al.* (2018).

Figura 17 – Área de atuação do acelerador implementado.



Fonte – Autor.

para diferentes camadas selecionadas dessa mesma rede. Para isso, é considerado os valores binarizados e, portanto, oito atributos é equivalente a 8 *bits* (1 *byte*). Percebe-se que o maior uso de memória da entrada se encontra nas camadas iniciais, enquanto que para os pesos o maior uso encontra-se nas camadas finais. Isso se deve ao fato de que o número de pesos aumenta com o número de canais do mapa de atributos de saída. Esses dados são comparados com os valores da Tabela 3, que apresenta a quantidade de recursos de memória para dispositivos FPGAs da família *Artix7* da

empresa *Xilinx*. Esta família de FPGA é otimizada para aplicações de menor custo e menor consumo de energia (XILINX, 2020). Destaca-se que estes valores apresentados representam a memória utilizável para uma organização de memória com dados de 32 *bits*.

Frente às Tabelas 2 e 3, verificou-se que os valores dos mapas de atributos de entrada binarizados podem ser facilmente carregados por inteiro na memória interna do FPGA e, como consequência, resultam em menos acessos na memória externa. O mesmo já não pode ser dito para os pesos, pois, para as camadas finais, a sua demanda de recursos é superior à quantidade de memória disponível. Sendo assim, o acelerador deve carregar os pesos para a memória interna em estágios, utilizando a mesma memória para pesos diferentes.

Tabela 2 – Memória utilizada em camadas selecionadas da rede *QuickNet*.

Camada	**Memória BRAM (kB)**	
	Entrada	Pesos
56x56x64	24,50	4,50
28x28x128	12,25	18,00
14x14x256	6,12	72,00
7x7x512	3,06	288,00

Fonte – Autor.

Tabela 3 – Memória disponível em dispositivos FPGAs da família Artix7 da empresa Xilinx.

Dispositivo	**Memória BRAM (kB)**
XC7A15T	100
XC7A25T	180
XC7A35T	200

Fonte – Xilinx (2020).

Em relação aos atributos de saída da convolução binária, os valores são representados por números inteiros. Segundo Conti, Schiavone e Benini (2018), 16 *bits* são suficientes para representar este valor sem sofrer problemas de estouros de magnitude (*overflow*/*underflow*) devido às acumulações mesmo com redes mais profundas que 512 canais.

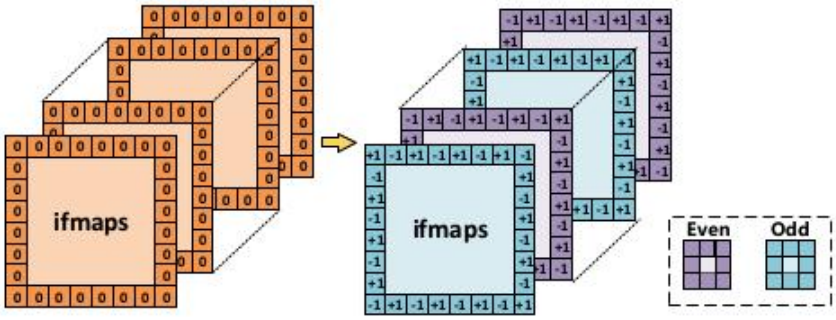
4.1.3 Padding

O *padding* consiste em adicionar novos elementos ao redor das bordas do mapa de atributos de entrada, permitindo que a saída mantenha as mesmas dimensões da

entrada. Em redes com precisão total, as convoluções são geralmente preenchidas com o valor zero. Essa prática padrão não pode ser aplicada a BNNs que exigem valores dentro do conjunto de codificação para habilitar operações *bit* a *bit*. Isto é, em uma BNN que utiliza a codificação -1,+1, adicionar um terceiro valor 0 torna as convoluções incompatíveis com as operações utilizando XNOR e *popcount* (FERRARINI *et al.*, 2022).

Alguns autores propuseram alternativas para não utilizar o *padding* com o mesmo valor. Guo *et al.* (2018) apresentaram a estratégia *Odd-Even Padding* em que os valores +1 e -1 são intercalados, como mostrado na Figura 18. No entanto, a diferença no erro resultante da rede ao se aplicar estes métodos é muito baixa e, portanto, optou-se por utilizar o *padding* com um valor fixo de -1 (0).

Figura 18 – Esquema de *padding* proposto por Guo *et al.* (2018).



Fonte – Guo *et al.* (2018).

Outro ponto analisado em relação ao *padding* foi o custo de sua movimentação da memória. A Tabela 4 apresenta uma comparação quantitativa entre o número de atributos considerados válidos e atributos resultantes de um padding (chamados aqui de *nulos*) em diversas camadas da BNN *QuickNet*. Como exemplo, realizar o *padding* na camada que possui dimensões 56x56x64 resultará em mapas de atributos com dimensões 58x58x64, nesse caso 7,3% dos atributos totais após o *padding* são sem relevância para a rede. Esse problema fica ainda mais evidente na camada com dimensões 7x7x512 em que mais da metade de todos os atributos são resultantes do *padding* e não carregam informação útil.

Em vista desse problema, o acelerador pode realizar esse *padding* em *hardware*, dispensando o movimento destes dados e levando à um desempenho superior tanto em questão energética quanto temporal.

Tabela 4 – Comparação da quantidade de elementos adicionados ao se realizar o *padding*.

Camada	**Quantidade de Atributos**		**Taxa de atributos nulos (%)**
	Válidos	Nulos	
56x56x64	200704	14592	7,3
28x28x128	100352	14848	14,8
14x14x256	50176	15360	30,6
7x7x512	25088	16384	65,3

Fonte – Autor.

4.2 ORGANIZAÇÃO DOS DADOS

Para a tarefa definida, os mapas de atributos e os pesos são dados tridimensionais. Como as memórias são inerentemente lineares, o mapeamento de dados multidimensionais pode ser realizado de várias maneiras. A definição de como os dados são armazenados e acessados na memória influencia diretamente na execução do algoritmo.

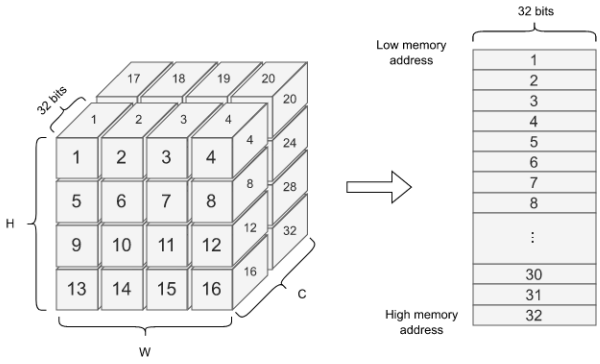
De acordo com Fortune Business Insights (2022), os microcontroladores mais utilizados hoje em dia possuem uma arquitetura e barramento de dados de 32 *bits* (4 *bytes*). Sendo assim, o acelerador foi projetado considerando a sua execução em um sistema onde as operações aritméticas e os acessos à memória são alinhados em 32 *bits* e com ordenação *little-endian*.

A Figura 19 apresenta a organização de um mapa de atributos 3D binarizado e sua correspondência aos endereços de memória. Dentro de um mesmo endereço, os atributos binários são agrupados em canais, isto é, os 32 *bits* desse endereço correspondem à sequência de 32 canais de uma mesma posição na linha e coluna da matriz 2D do mapa de atributos.

Como mostrado na Figura 20, a organização dos pesos da BNN seguem um padrão semelhante ao dos mapas de atributos de entrada. Uma vez que é necessário um cubo 3D de pesos para cada mapa de atributo 2D da saída, o conjunto de pesos é constituído por *C_out* cubos 3D para uma saída de *C_out* canais. Estes cubos 3D de pesos são dispostos sequencialmente na memória.

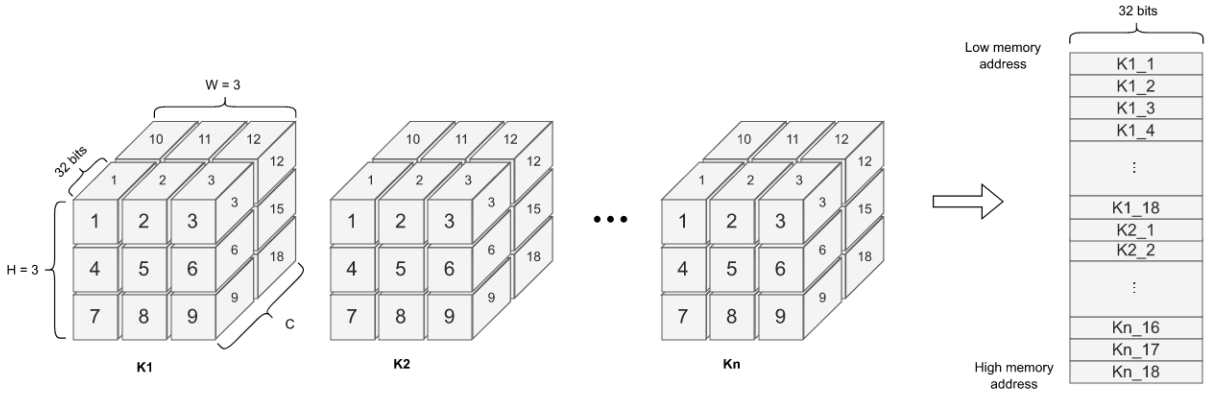
A organização dos atributos de saída segue o mesmo padrão dos atributos de entrada. No entanto, devido ao resultado da operação de *popcount*, cada atributo é representado por um número inteiro de 16 *bits*. Isso resulta em um número muito menor de canais armazenados em um mesmo endereço de memória em comparação com a entrada, como ilustrado na Figura 21.

Figura 19 – Organização dos mapas de atributos binarizados na memória.



Fonte – Autor.

Figura 20 – Organização dos pesos binarizados na memória.



Fonte – Autor.

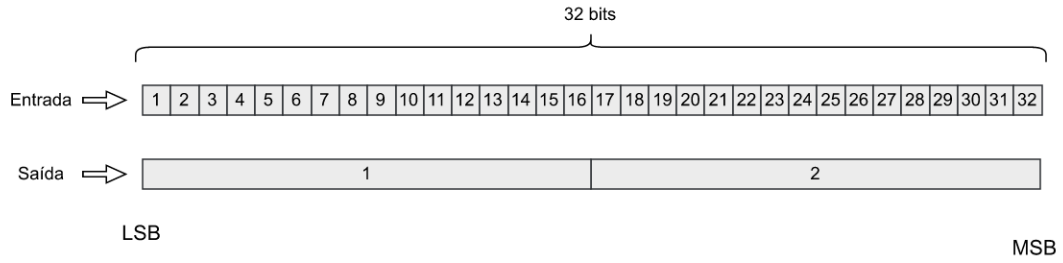
4.3 ALGORITMO DE CONVOLUÇÃO BINÁRIA

A Figura 22 ilustra os parâmetros que compreendem uma convolução em 3D e a Quadro 1 apresenta um pseudocódigo de um algoritmo simples para realizar a convolução binária.

É possível reordenar a ordem dos laços de execução para um acesso mais eficiente à memória. Por exemplo, como os resultados intermediários da convolução binária são números inteiros provenientes da operação de *popcount* (16 *bits* nesse caso), é preferível executar o algoritmo de modo que esse valor permaneça na memória interna e seja movido para a memória externa apenas no seu valor final.

O algoritmo de execução utilizado no acelerador está apresentado na Quadro 2.

Figura 21 – Comparação entra a organização dos atributos de entrada e saída em um mesmo endereço de memória.

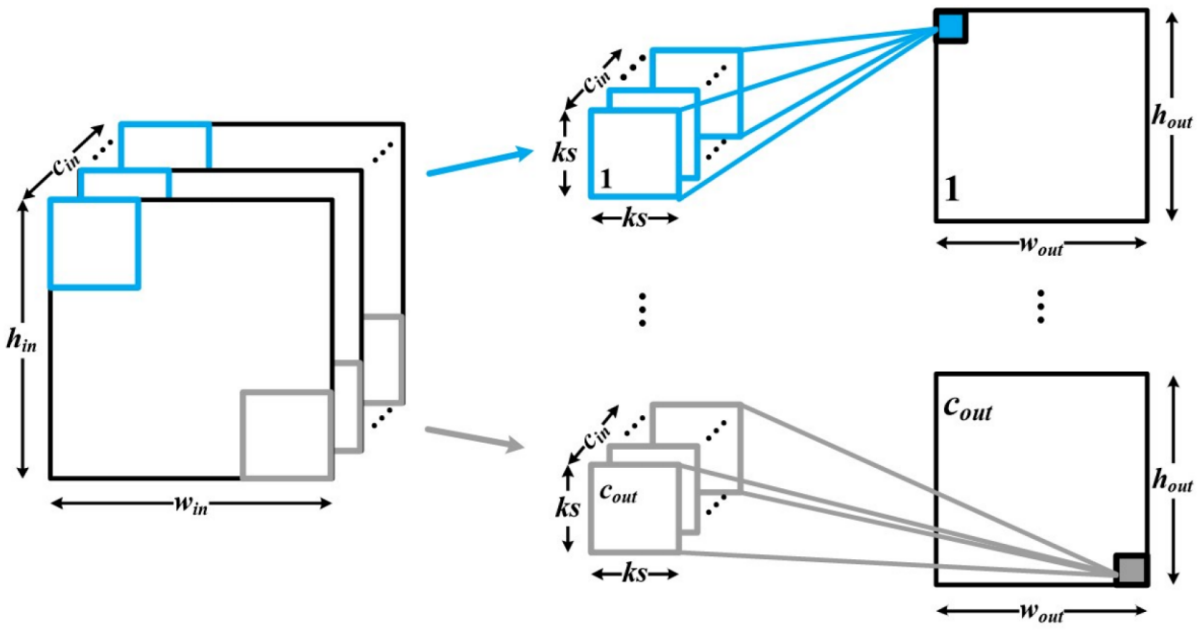


Fonte – Autor.

Neste algoritmo, além de dispensar movimentos desnecessários dos atributos de saída, os pesos são utilizados para gerar todos os seus mapas de atributos de saída possíveis e, portanto, não é necessário carregá-los novamente da memória externa.

O parâmetro TPO é definido em projeto e estabelece o número de canais dos atributos de saída simultâneos que são calculados além do número de pesos que são carregados da memória externa. A cada finalização do laço de linha da saída, estes atributos são salvos na memória externa e um novo grupo de TPO números de atributos são calculados.

Figura 22 – Parâmetros para a convolução 3D.



Fonte – Cho *et al.* (2021).

Quadro 1 – Pseudocódigo de um algoritmo de convolução simples.

```
for j in range(0, h_out):                          # coluna saida
  for i in range(0, w_out):                        # linha  saida
    for oc in range(0, c_out):                     # canal  saida
      for ic in range(0, c_in):                    # canal  entrada
        for wj in range(0, ks):                    # coluna pesos
          for wi in range(0, ks):                  # linha  pesos
            y[oc,j,i] += W[oc,ic,wj,wi] * x[ic,j+w_j,i+w_i]
```

Fonte – Autor.

Quadro 2 – Pseudocódigo do algoritmo utilizado para a convolução binária.

```
# carrega entrada
for oc_o in range(0, c_out/TPO):                   # canal  saida (externo)
  # carrega pesos
  for j in range(0, h_out):                        # coluna saida
    for i in range(0, w_out):                      # linha  saida

      for wj in range(0, ks):                      # coluna pesos
        for wi in range(0, ks):                    # linha  pesos
          for ic in range(0, c_in/32):             # canal  entrada
            for oc_i in range(0, TPO):             # canal  saida (interno)
              oc = oc_o * TPO + oc_i
              y[oc,j,i] += popcount(XNOR(W[oc,ic,wj,wi],x[ic,j+w_j,i+w_i
                 ]))
      # salva TPO canais da saida
```

Fonte – Autor.

4.4 MICROARQUITETURA

4.4.1 Visão Geral

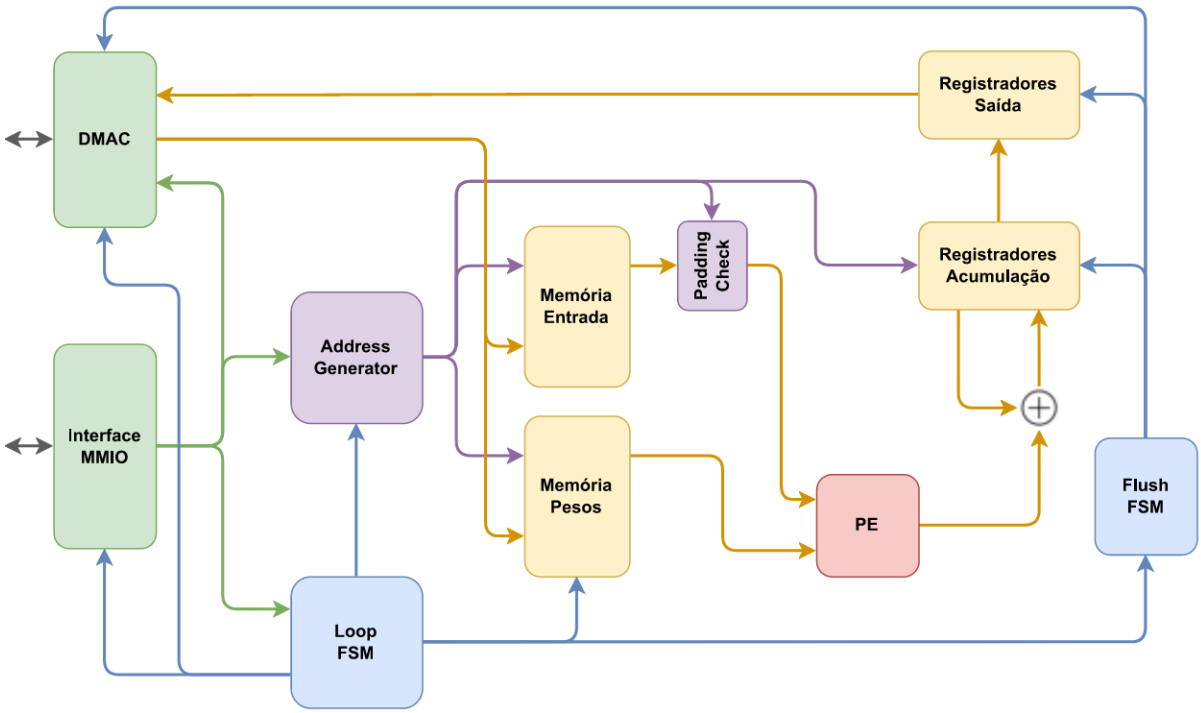
Como investigado na Seção 2.2.2, o modelo de acelerador mais adequado para a categoria de tarefa desejada é o acelerador fracamente acoplado com memória privada. Sendo assim, o acelerador implementado neste trabalho foi desenvolvido seguindo estes conceitos. Além disso, no paradigma de aceleradores de DNN abordado na Seção 2.2.3, ele se encaixa como acelerador de camadas, pois realiza a aceleração de uma camada convolucional por vez.

O acelerador implementado, apelidado com o codinome “Xnorator”, foi desenvolvido utilizando a linguagem de descrição de *hardware SystemVerilog*. Seu código fonte é disponibilizado no repositório https://github.com/cleissom/Xnorator. A Figura 23 apresenta um diagrama de blocos com sua arquitetura geral. Uma descrição de cada bloco presente na microarquitetura implementada é apresentada nas seções seguintes.

Na Figura 23 encontra-se o diagrama de sequência para a utilização do acelerador. Como ilustrado, após configurá-lo para iniciar a convolução, o núcleo de processamento é liberado para realizar quaisquer outras computações enquanto a convolução

é executada em paralelo pelo acelerador.

Figura 23 – Diagrama de blocos da arquitetura interna do acelerador.



Fonte – Autor.

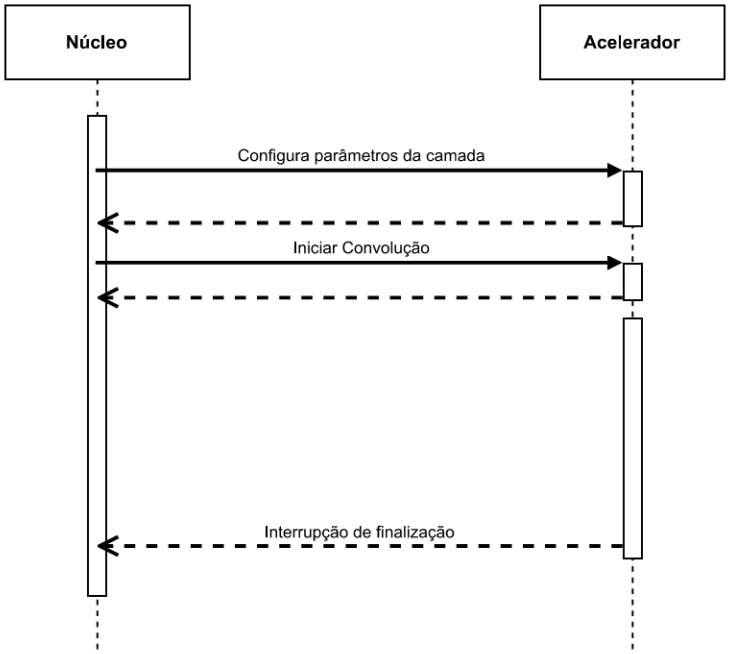
4.4.2 Interface MMIO

O núcleo de processamento comunica-se com a memória e os periféricos por meio de um barramento, que permite que os dados sejam roteados por um conjunto fixo de conexões. A partir do núcleo ou, em geral, de qualquer dispositivo mestre do barramento, não há diferença entre periféricos e memórias, pois ambos são referenciados por endereços. Nesse esquema, chamado de *memory mapped I/O* (MMIO), partes do espaço de endereçamento são atribuídas a dispositivos periféricos, e leituras e gravações nesses endereços são interpretadas como comandos para os mesmos (PATTERSON; HENNESSY, 2020).

Na Tabela 5 é apresentado os endereços de memórias disponibilizados na interface MMIO. A coluna “Endereço” corresponde ao deslocamento do endereço base, que é atribuído pelo mapeamento no barramento de dados. Por exemplo, em um sistema que foi atribuído para este acelerador o espaço de memória no endereço "0x0004_0000", para escrever no registrador START é utilizado o endereço "0x0004_0020".

Através destes endereços é possível configurar parâmetros como as dimensões dos mapas de atributos de entrada e saída e endereços para o DMA do acelerador.

Figura 24 – Diagrama de sequência para a utilização do acelerador.



Fonte – Autor.

Após configurado os parâmetros, uma escrita com valor ’1’ no endereço de “start” inicia o algoritmo de convolução. A finalização da convolução define o registrador do endereço DONE com o valor ’1’ e habilita o sinal de interrupção disponível para conexão.

Atualmente, o acelerador permite apenas valores de canais múltiplos de 32.

Tabela 5 – Endereços dos registradores da interface MMIO.

Nome	**Endereço**	**Leitura(R) Escrita(W)**	**Descrição**
INPUT_ADDR	0x00	W	Endereço base para a leitura dos dados de entrada
WEIGHT_ADDR	0x04	W	Endereço base para a leitura dos pesos
OUTPUT_ADDR	0x08	W	Endereço base para a escrita dos dados de saída
INPUT_WIDTH	0x0C	W	Largura do mapa de atributos de entrada
INPUT_HEIGHT	0x10	W	Altura do mapa de atributos de entrada
INPUT_CHANNEL	0x14	W	Número de canais no mapa de atributos de entrada
OUPUT_CHANNEL	0x18	W	Número de canais no mapa de atributos de saída
PADDING	0x1C	W	Define o uso de padding na convolução
START	0x20	W	Inicia a convolução com os dados fornecidos
DONE	0x24	R	Retorna o status da finalização da convolução

Fonte – Autor.

4.4.3 DMAC

De acordo com Patterson e Hennessy (2020), o DMA (*Direct Memory Access*) é um mecanismo que fornece a um dispositivo a capacidade de transferir dados diretamente para ou da memória sem envolver o processador.

O Controlador DMA (DMAC) utiliza o barramento de dados como mestre e acessa a memória através do endereçamento semelhante ao procedimento que o processador realizaria. Como o DMAC é feito completamente por hardware, ele pode substituir efetivamente as instruções de *load/store* que o processador utiliza para manipular os dados da memória. Deste modo, é possível transmitir uma quantidade maior de dados com menos *overhead*.

O DMAC aguarda os sinais de requisição de transferência de dados, são estes: requisição dos dados de entrada, requisição dos pesos e requisição de envio dos dados de saída. Para as requisições dos dados de entrada e pesos, o DMAC calcula o endereço a realizar a leitura da memória externa com base nos parâmetros fornecidos pela interface MMIO, e então realiza a transferência através da leitura na memória externa e escrita na memória interna sequencialmente. O número de elementos transferidos também é calculado com base nos parâmetros fornecidos.

A requisição de envio dos dados de saída faz um processo semelhante, no entanto, o fluxo dos dados é inverso. Isto é, realiza-se a leitura dos registradores contendo os valores de saída e escreve-se na memória externa no endereço correspondente, calculado seguindo a formato dos dados de saída.

4.4.4 Memória Local

Como constatado no Seção 2.2.2, o uso de uma memória local privada personalizada para a tarefa resulta em um maior desempenho para o acelerador pois, acessos à memória externa são ordens de magnitude mais custosas tanto em questão temporal quanto energética.

Na arquitetura do acelerador são utilizados dois bancos de memórias separados para os dados de entrada e para os dados dos pesos. A separação desses dados em memórias diferentes possibilita a leitura em paralelo e a computação mais eficiente do respectivo endereço de leitura no componente “AddressGenerator”.

Como já exposto na Seção 4.1.2, os atributos dos dados de entrada são carregados por completo na sua respectiva memória. O tamanho máximo dessa memória é vinculado à topologia da rede utilizada. Já o tamanho da memória dos pesos depende do número máximo de canais da rede como também o número de atributos de saída definidos pelo parâmetro TPO.

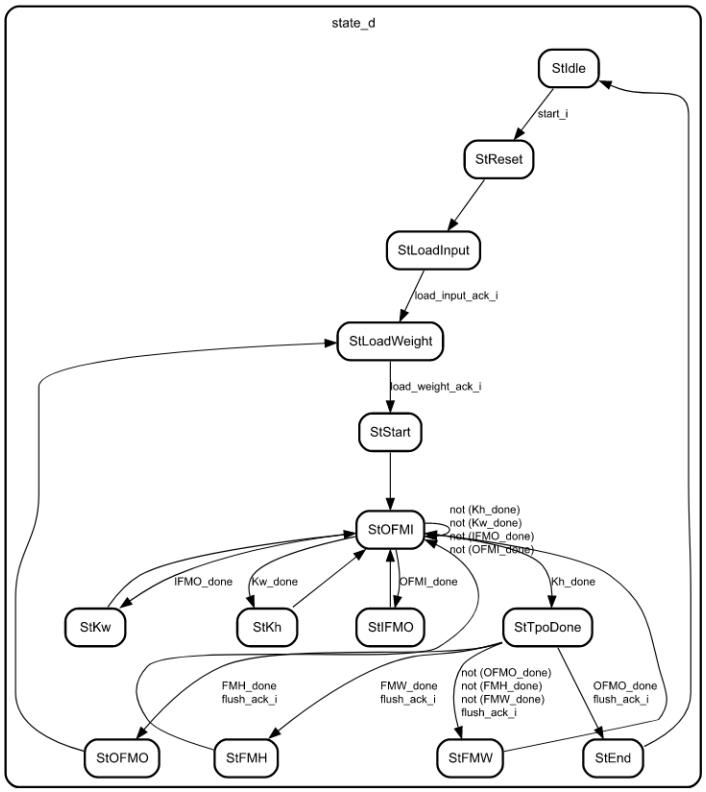
4.4.5 PE

O componente PE (*Processing Engine*) é o responsável por realizar as operações de XNOR e *popcount*. A operação de XNOR é realizada em 2 valores de 32 *bits* e a saída é encaminhada para uma uma árvore somadora binária que realiza a contagem do número de *bits* igual a 1, resultando no valor da operação de *popcount*. Por fim, é realizado o ajuste de escala em que o valor inteiro pode ser positivo ou negativo.

4.4.6 Loop FSM e Address Generator

O componente “Loop FSM” é composto por uma máquina de estados finitos, ilustrado na Figura 25, que realiza os laços de execução do algoritmo de convolução binária. Os limites de contagem são variáveis dado que é utilizado como base os parâmetros da camada a ser calculada fornecidos na interface MMIO. As variáveis de contagem dos laços de execução são enviadas para o componente “Address Generator” onde é realizada a decodificação destes valores para um índice de memória com base no mapeamento de dados estabelecido na Seção 4.1.2.

Figura 25 – Diagrama de estados do componente Loop FSM.



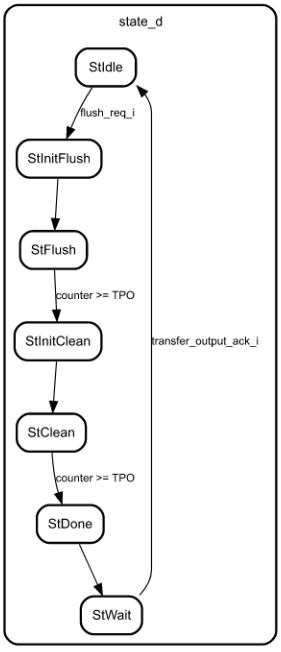
Fonte – Autor.

4.4.7 Banco de Registradores e Flush FSM

Para os atributos de saída é utilizado um banco de registradores com o número de registradores igual ao parâmetro TPO. O registrador ativo para leitura e escrita é controlado pelo componente “Address Generator”. Em seu funcionamento, o valor do endereço atual é lido e sobrescrito com a soma do seu valor e o resultado da operação *popcount*, resultando na acumulação proveniente do algoritmo de convolução.

Após a finalização do cálculo dos atributos de saída, estes valores são transferidos para outro banco de registradores de mesmo tamanho por itermédio do componente “Flush FSM”. Deste modo, o cálculo dos próximos atributos de saída pode ser iniciado enquanto que o DMAC realiza a transferência dos valores já calculados, sobrepondo a comunicação com a computação usando o conhecido *buffer* duplo (ZHANG, C. *et al.*, 2015). O diagrama de estados do Flush FSM está ilustrado na Figura 26. Como mostrado na Figura 27, essa estratégia implementada permite extrair um maior desempenho temporal através do paralelismo.

Figura 26 – Diagrama de estados do componente Flush FSM.

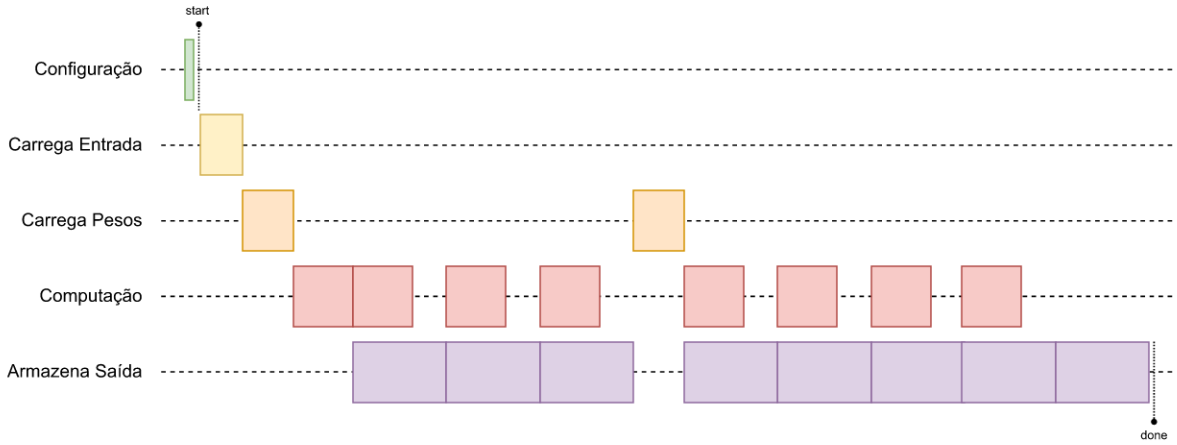


Fonte – Autor.

4.4.8 Padding Check

Como mencionado no Seção 4.1.3, o acelerador realiza o padding em *hardware*, dispensando o carregamento de dados irrelevantes. Este componente utiliza a infor-

Figura 27 – Paralelismo de computação e comunicação.



Fonte – Autor.

mação de endereços proveniente do componente “AddressGenerator” e realiza uma multiplexação entre o dado lido da memória e um dado com valor fixo para o *padding*.

5 TESTES E RESULTADOS

Esse capítulo inclui simulações e dados com o intuito de validar os resultados deste trabalho.

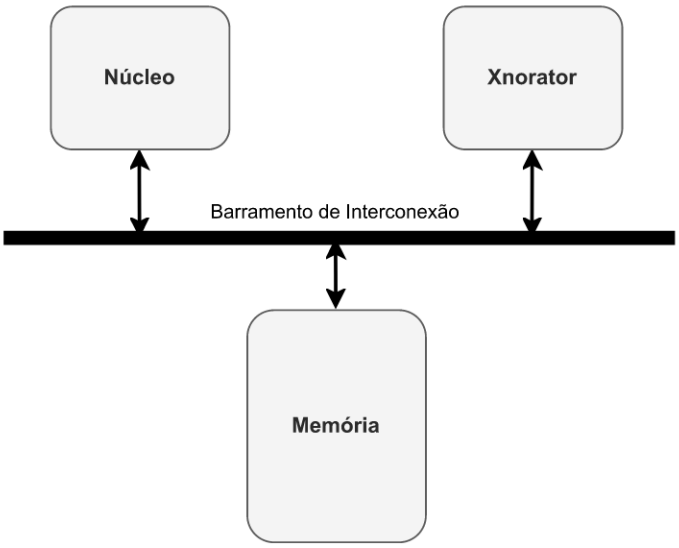
5.1 SIMULAÇÃO

Como mencionado anteriormente, o acelerador foi desenvolvido com a finalidade de aplicação em computação de borda, onde são utilizados dispositivos geralmente constituídos por sistemas embarcados compostos por microcontroladores. Portanto, a fim de avaliar o desempenho do acelerador em seu âmbito alvo, foi montando um ambiente de simulação constituído pelos elementos base:

- Núcleo de processamento;
- Memória de dados;
- Acelerador implementado;
- Barramento de dados.

A Figura 28 ilustra a conexão do acelerador com o núcleo de processamento e a memória.

Figura 28 – Diagrama de conexão com o acelerador.



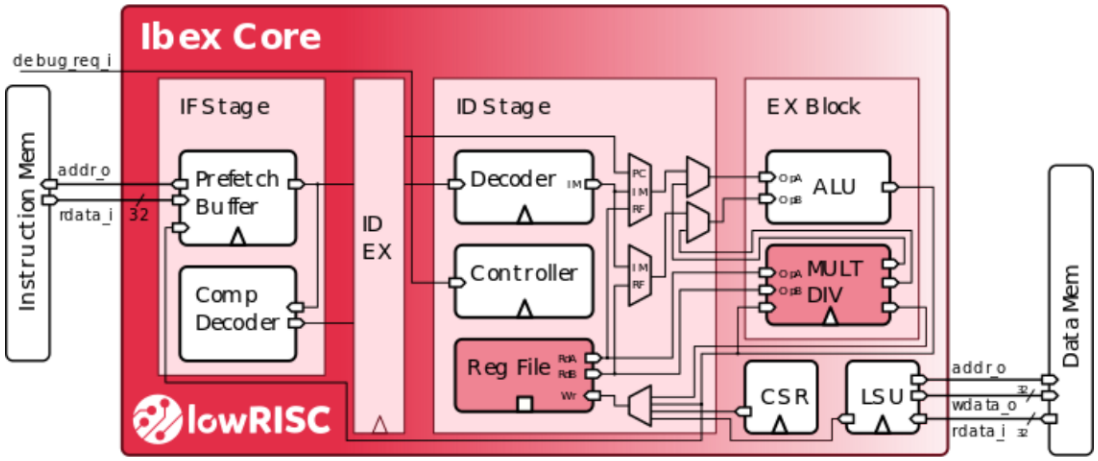
Fonte – Autor.

Para o núcleo de processamento, foi utilizado o núcleo RISC-V de 32 *bits* Ibex[1]. Este é um núcleo de código aberto, parametrizável, com *pipeline* de 2 estágios e, de

[1] https://github.com/lowRISC/ibex

acordo com Schiavone *et al.* (2017), adequado para a utilização em sistemas embarcados de computação de borda. Seu diagrama de blocos é apresentado na Figura 29. Além disso, este núcleo possui suporte à extensão padrão para instruções de manipulação de *bits*, o que é de interesse para utilização neste trabalho.

Figura 29 – Diagrama de blocos do núcleo Ibex.



Fonte – lowRISC (2022).

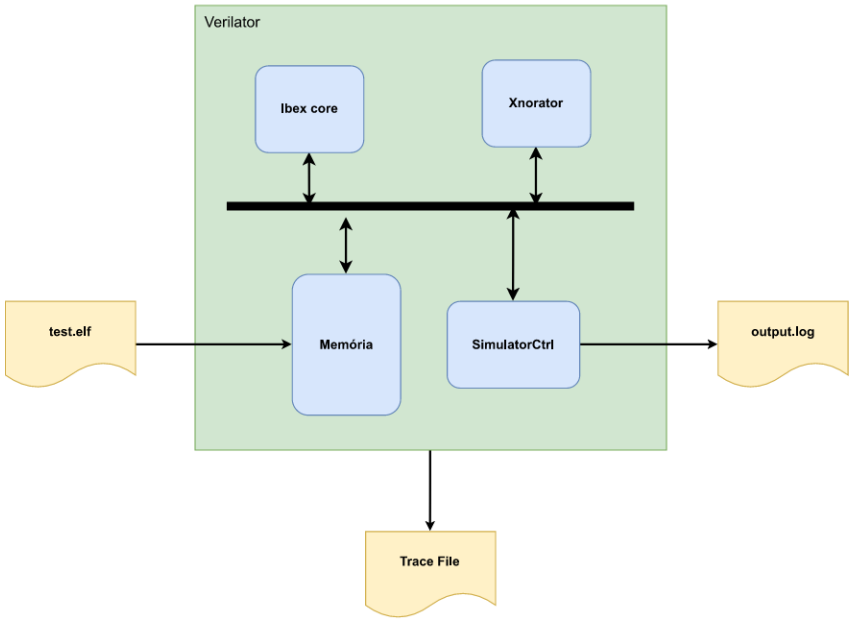
A simulação funcional do sistema é realizada utilizando-se Verilator, um simulador HDL de código aberto que compila *SystemVerilog* e *Verilog* em um código C++. Deste modo, é possível executar a simulação como uma aplicação nativa do sistema sendo utilizado.

A Figura 30 ilustra a organização dos componentes utilizados na simulação. Um arquivo contendo a aplicação compilada para RISC-V é escrito na memória durante a inicialização da simulação. Após o início da simulação, o núcleo Ibex realiza a busca e leitura das instruções nesta memória e prossegue sua execução.

Além dos módulos que compõem o sistema citados anteriormente, para a simulação no Verilator é utilizado um componente denominado de “SimulatorCtrl” onde é possível escrever valores e caracteres em um arquivo no sistema onde está sendo executado a simulação. Deste modo, é possível executar a aplicação e receber dados de retorno de sua execução. Além disso, a aplicação de simulação gera um arquivo que permite a visualização das formas de ondas da execução efetuada. Destaca-se que, para a simulação e consequentemente a obtenção dos dados, é utilizado uma memória simples com latência de acesso de 1 ciclo de *clock*.

A fim de demonstrar o comportamento do acelerador através da simulação, foi utilizado um acelerador com parâmetro TPO igual a 8 e mapas de atributos de entrada com dimensões 7x7x64. Para visualização das formas de ondas foi utilizado o *software* GtkWave.

Figura 30 – Esquema de simulação utilizando o Verilator.



Fonte – Autor.

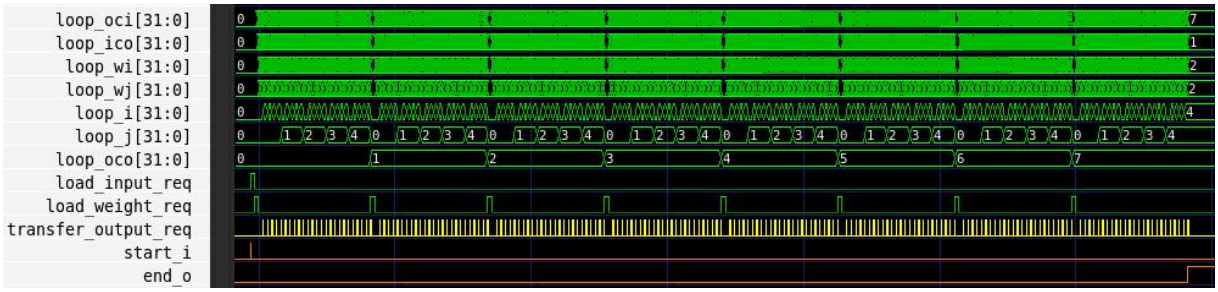
Na Figura 31 é apresentada as formas de ondas para a execução completa do algoritmo de convolução binária. O sinal *start_i* é disparado quando há a escrita no endereço START da interface MMIO do acelerador e sinaliza para a o componente “Loop FSM” o início da execução. Os sinais que possuem o prefixo “loop_” no nome fazem parte das variáveis de contagem dos laços presentes no algoritmo de convolução.

Além disso, nesta figura estão presentes os três sinais que comunicam ao DMAC a transferência de dados. Os sinais *load_input_req* e *load_weight_req* fazem a requisição para a transferência dos dados dos mapas de atributos de entrada e pesos, respectivamente e é realizado a transferência de dados da memória externa para a memória interna. O sinal *transfer_output_req* faz a requisição para realizar a transferência dos dados de saída calculados do banco de registradores para a memória externa. Por fim, ao final da execução do algoritmo é habilitado o sinal *end_o*. Destaca-se que que os sinais apresentados são selecionados e referentes aos componentes que realizam o controle do *datapath*.

Na Figura 32 é alterada a escala de tempo para exibir e apenas uma execução do laço externo do canal de saída. A Figura 33 reduz ainda mais a escala de tempo para exibir apenas uma execução do laço da coluna de saída. Nesta figura é possível ver a requisição de transferência dos valores de saída sendo realizada ao final de cada laço da linha de saída.

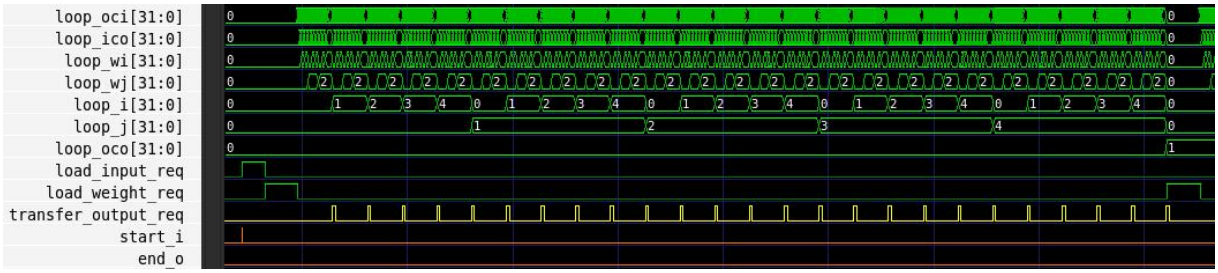
As Figuras 34, 35 e 36 apresentam as formas de ondas para o mesmo exemplo discutido anteriormente, no entanto, é adicionado a configuração do parâmetro de

Figura 31 – Formas de onda do laço de convolução completo.



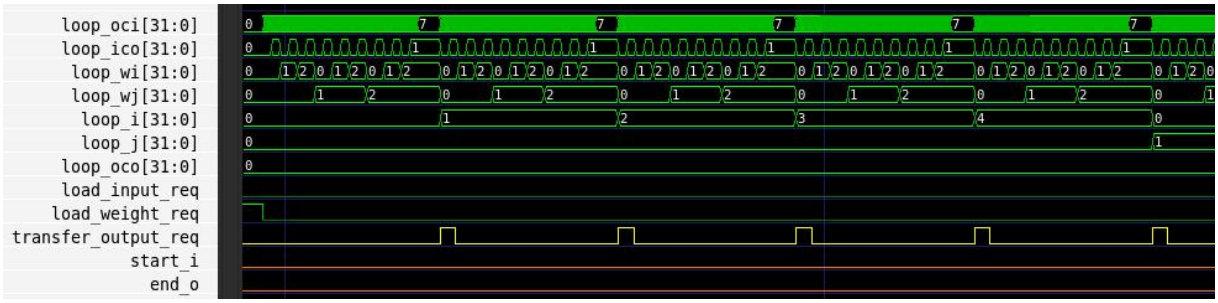
Fonte – Autor.

Figura 32 – Formas de onda do laço externo de canal da saída.



Fonte – Autor.

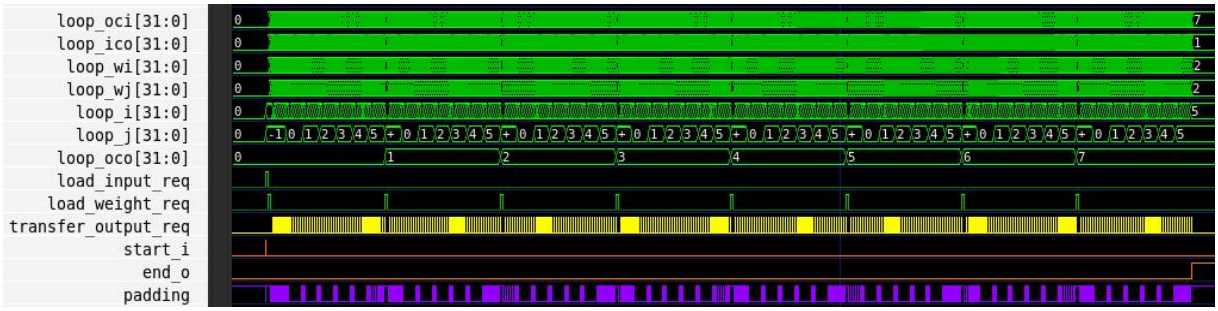
Figura 33 – Formas de onda do laço de coluna da saída.



Fonte – Autor.

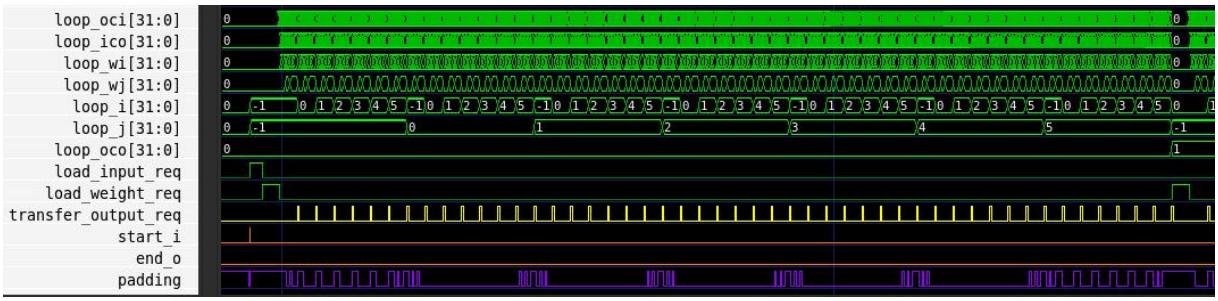
padding na interface MMIO. Estas figuras apresentam o sinal *padding* indicando a atuação do componente “PaddingCheck” toda vez que é detectado o acesso a um elemento decorrente do mesmo. Nota-se que, para esta configuração, os valores dos contadores de linha e coluna da saída possuem inicialização e limites de contagem diferentes dos anteriores.

Figura 34 – Formas de onda do laço de convolução completo com *padding*.



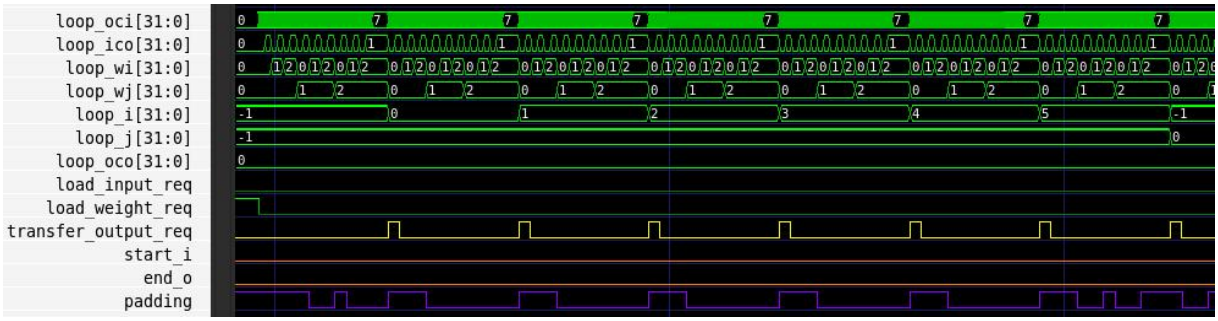
Fonte – Autor.

Figura 35 – Formas de onda do laço externo de canal da saída com *padding*.



Fonte – Autor.

Figura 36 – Formas de onda do laço de coluna da saída com *padding*.



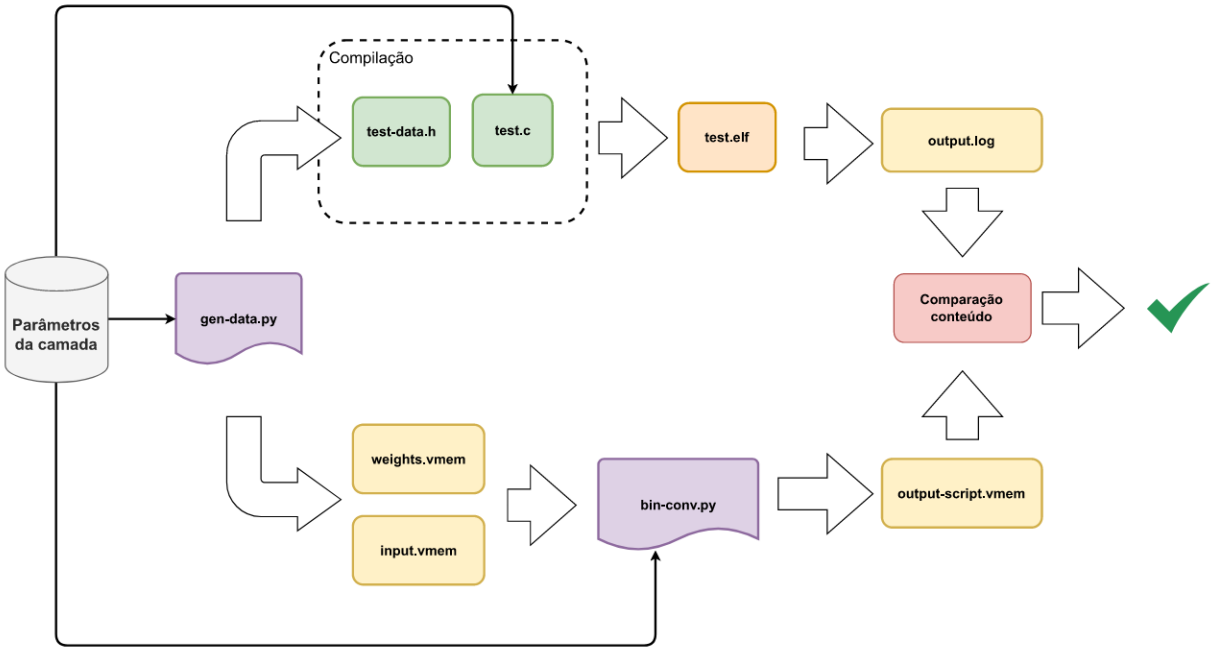
Fonte – Autor.

5.2 VALIDAÇÃO

Para realizar a verificação dos dados calculados pelo acelerador, foi elaborado o ambiente de testes ilustrado na Figura 37.

O *script* Python desenvolvido *gen-data.py* (Anexo B) realiza a geração dos dados utilizados posteriormente para teste da convolução. A partir do fornecimento

Figura 37 – Esquema utilizado para testes.



Fonte – Autor.

dos parâmetros de dimensões dos mapas de atributos de entrada e saída, é gerado dois arquivos contendo números aleatórios, um destes com valores para o mapa de atributos de entrada e outro com valores para serem utilizados como pesos para a convolução. O número de valores é determinado e calculado através dos parâmetros fornecidos e os valores são armazenados em um arquivo *vmem* em que cada linha possui um dado de 32 *bits* em formato hexadecimal com codificação ASCII.

Além dos arquivos *vmem*, os mesmos valores são utilizados para gerar um único arquivo de cabeçalho (*header*) de linguagem C contendo *arrays* inicializadas com os respectivos valores. Além da entrada e dos pesos, é adicionado uma *array* de saída alocada com o número de parâmetros de saída calculados pelo mesmo *script* como também é fornecido uma variável contendo este valor para acesso pelo programa. Deste modo, para testes, o programa compilado utiliza os valores gerados inicializados de forma estática na memória.

Após gerado os dados de teste, os arquivos *vmem* são utilizados por outro *script* Python desenvolvido (Anexo A) onde é realizada a convolução binária utilizando a biblioteca *NumPy*, que disponibiliza funções matemáticas e de manipulação de matriz. A sua saída é um outro arquivo *vmem* contendo os valores hexadecimal que a *array* de saída do acelerador deve apresentar, em concordância com o formato dos dados de saída como apresentado na Seção 4.2.

De forma paralela, o ambiente de simulação do acelerador executa o programa

compilado contendo os mesmos valores gerados. Ao final da execução, é chamada uma rotina que realiza a leitura de todos os dados de saída presentes na memória e envia os mesmos para o arquivo de saída *output.log* através do procedimento de simulação explicado no capítulo anterior.

Por fim, é possível realizar um utilitário de comparação de arquivos para analisar os dois itens gerados paralelamente e verificar a presença de diferenças de valores. Para o cálculo correto do acelerador, o conteúdo do arquivo gerado pela simulação deve ser exatamente o mesmo do arquivo gerado pelo *script*, dado os mesmos parâmetros de entrada.

Em suma, a operação do acelerador foi validada utilizando dados gerados aleatóriamente e parâmetros de dimensões dos mapas de atributos conforme encontrados na rede *QuickNet*.

5.3 MÉTRICAS

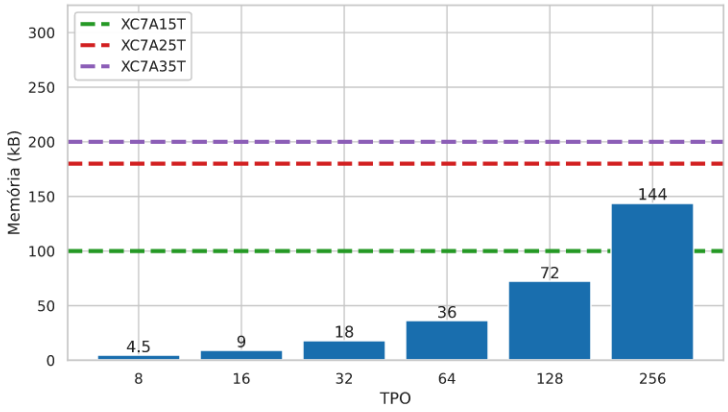
A partir do ambiente de simulação montado, foram levantados dados pertinentes à execução do acelerador. Primeiramente, foi realizado uma análise na influência da variação no parâmetro de projeto TPO. Além de definir o tamanho do banco de registradores, este parâmetro influencia diretamente o tamanho de memória necessário para carregar os pesos. O cálculo do tamanho de memória necessário para os pesos pode ser obtido através da Equação (6). O valor *MAX_DEPTH* é uma constante que representa o número máximo de canais possíveis de serem utilizados no acelerador. Para a rede *QuickNet* este valor é 512 e, portanto, este é o valor utilizado para os testes realizados. O parâmetro k_S é o mesmo utilizado no Seção 4.3 e define a dimensão de largura e altura dos pesos.

$$kB = \frac{TPO * {k_S}^2 * MAX\_DEPTH}{8 * 1024} \tag{6}$$

A Figura 38 apresenta um gráfico contendo a variação do tamanho de memória de pesos necessário para cada valor TPO estipulado. Adicionalmente, o gráfico apresenta linhas horizontais que delimitam o tamanho máximo de memória disponível nos determinados dispositivos FPGA, com base nos dados da Tabela 3. Podemos perceber que o valor de TPO maior que 128 excede a capacidade de memória do FPGA XC7A15T.

A quantidade de memória necessária para armazenar os valores do mapa de atributos de entrada não possui variação com o valor de TPO. Estes valores são carregados ao iniciar o algoritmo de convolução binária e permanecem inalterados até a sua finalização. Além disso, para este caso, a quantidade de memória necessária é definido pelo tamanho máximo dos dados de entradas e, conforme os dados já apresentados na Tabela 2, para a rede *QuickNet* é necessário uma memória de pelo

Figura 38 – Memória necessária para armazenar os valores de pesos para diferentes parâmetros TPO.



Fonte – Autor.

menos 24,5 kB para armazenar a primeira camada, onde se encontra a maior demanda de armazenamento.

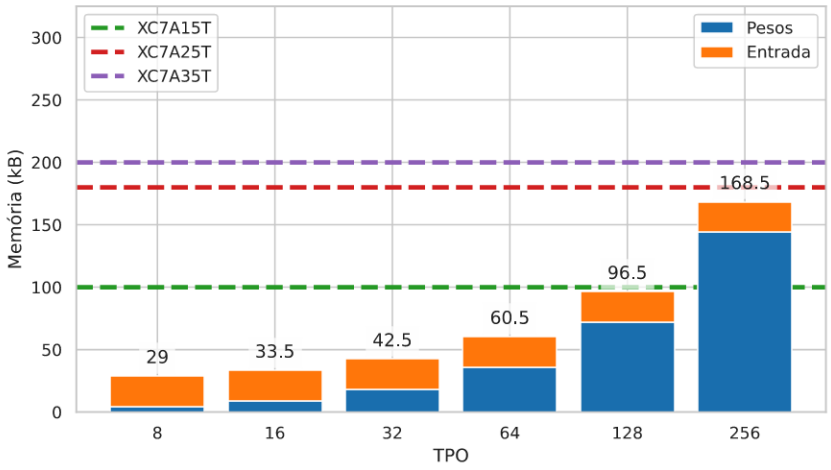
A partir disso, a Figura 39 apresenta a soma das memórias de entrada e pesos com a variação do parâmetro TPO. Podemos concluir que os valores de TPO ideais para esta arquitetura de acelerador e para o uso no dispositivo XC7A15T são 32 e 64, pois, não subutilizam os recursos disponíveis como também não chegam perto do limite de excedê-los. Sendo assim, nos testes seguintes é utilizado o valor fixo do parâmetro TPO definido como 64, caso o mesmo não seja especificado com outro valor.

Para analisar a influência da utilização do acelerador desenvolvido, foi realizada uma comparação entre a execução do algoritmo de convolução binária utilizando apenas o núcleo de processamento RISC-V e posteriormente o núcleo em conjunto com o acelerador. Além disso, como mencionado na Seção 2.3.2, o RISC-V permite a inclusão (ou não) de extensões em sua implementação. A extensão de manipulação de *bits* possui uma instrução especializada para a execução da operação de *popcount* como também para a operação lógica XNOR. Sendo assim, essa extensão pode ser de interesse para a utilização de BNNs dado que estas operações são predominantes na camada de convolução binária.

Esse teste de comparação utilizando o algoritmo de convolução binária foi executado nas seguintes configurações de núcleo RISC-V:

- **RV32IMC**: Configuração básica utilizando as instruções base de 32 *bits* como também as extensões para multiplicação e divisão de inteiros e a de compressão de instruções;

Figura 39 – Memória necessária para armazenar os valores dos mapas de atributos de entrada e pesos para diferentes parâmetros TPO.



Fonte – Autor.

- **RV32IMCB**: Configuração que adiciona a extensão de manipulação de *bits*.

Dado que a configuração RV32IMC não possui instrução dedicada para realizar a operação de *popcount*, foi utilizado o algoritmo de Warren (2012), apresentado na Quadro 3, para a obtenção de seu resultado. Além disso, a operação lógica XNOR é obtida nesta configuração através da operação de XOR seguida da operação NOT.

Quadro 3 – Algoritmo para operação de *popcount*

```
uint32_t __popcount(uint32_t a) {
    uint32_t x = a;
    x = x - ((x >> 1) & 0x55555555);
    x = ((x >> 2) & 0x33333333) + (x & 0x33333333);
    x = (x + (x >> 4)) & 0x0F0F0F0F;
    x = (x + (x >> 16));
    return (x + (x >> 8)) & 0x0000003F;
}
```

Fonte – Autor.

Um ponto importante a se destacar é a influência do *padding* na execução do acelerador. O uso do *padding* neste acelerador tem influência no número de ciclos necessários para a leitura dos dados de entrada, no entanto, não possui influência nos ciclos de execução da computação do algoritmo de convolução binária. Portanto, para realizar a análise e comparação do desempenho da execução do algoritmo e permitir utilizar os mesmos dados de entrada, nestes testes de comparação não foi utilizado o *padding* do acelerador.

O binário executado foi gerado pela *toolchain* lowrisc-toolchain-gcc-rv32imcb[2] através do compilador GCC com otimização Os.

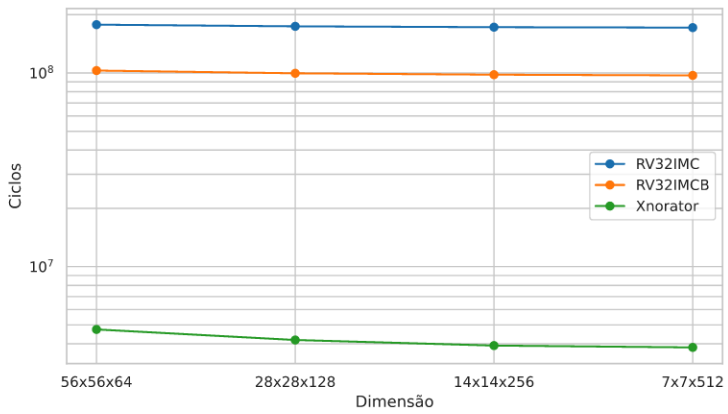
A Tabela 6 apresenta o número de ciclos de execução do simulador para as diferentes configurações definidas. Observa-se que apesar da utilização de um núcleo com a extensão de *bits* reduzir a número de ciclos necessários para completar o algoritmo em quase pela metade, a redução ao se utilizar o acelerador é de quase duas ordens de grandeza de diferença.

Tabela 6 – Ciclos de execução do algoritmo de convolução binária para as diferentes configurações de teste.

Dimensão	**Configuração**		
	RV32IMC	RV32IMCB	Xnorator
56x56x64	177567659	102902679	4740270
28x28x128	174072139	99610187	4180868
14x14x256	172332611	97971585	3913344
7x7x512	171464905	97154195	3828128

Fonte – Autor.

Figura 40 – Comparação entre os ciclos de execução do algoritmo de convolução binária para as diferentes configurações.



Fonte – Autor.

As Tabelas 7 e 8 mostram os dados de ciclos obtidos nos testes convertidos para variações percentuais de redução do número de ciclos executados. A Tabela 7 apresenta o percentual de redução de ciclos de execução ao se utilizar o acelerador

[2] https://github.com/lowRISC/lowrisc-toolchains/blob/20220210-1/lowrisc-toolchain-gcc-rv32imcb.config

ao invés das configurações de núcleo RISC-V estipuladas. Já na Tabela 8 é mostrado esse percentual de redução ao se partir da configuração de núcleo RISC-V simples para a configuração de RISC-V com a extensão de manipulação de *bits*. A Figura 41 mostra estes valores em forma de gráfico.

Em vista destes resultados, podemos concluir que a execução do algoritmo de convolução binária em um hardware especializado como o de um acelerador específico pode reduzir drasticamente o número de ciclos necessários. Isso é ocasionado por diversos fatores e, dentre eles podemos citar como exemplo que os cálculos de acesso aos endereços são realizados paralelamente em apenas um ciclo, em contraste com o cálculo no núcleo de processamento que deve ser sequencial e necessita de vários ciclos.

Tabela 7 – Redução percentual do número de ciclos necessários para executar o algoritmo de convolução binária no acelerador em comparação com as configurações de RISCV.

Dimensão	**Configuração**	
	RV32IMC	RV32IMCB
56x56x64	97,33%	95,39%
28x28x128	97,60%	95,80%
14x14x256	97,73%	96,01%
7x7x512	97,77%	96,06%

Fonte – Autor.

Tabela 8 – Redução percentual do número de ciclos necessários para executar o algoritmo de convolução binária no núcleo com extensão de manipulação bits em comparação com o núcleo simples.

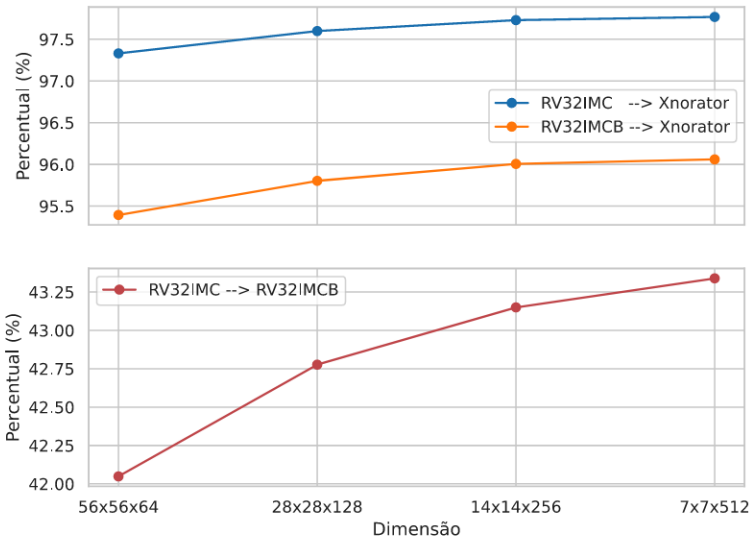
Dimensão	**Percentual (%)**
56x56x64	42,05
28x28x128	42,78
14x14x256	43,15
7x7x512	43,34

Fonte – Autor.

5.4 SÍNTESE

Por fim, o acelerador proposto foi sintetizado para FPGA utilizando o *software* Vivado da empresa Xilinx, com otimização padrão da ferramenta. A Tabela 9 apresenta os recursos utilizados da síntese para um dispositivo FPGA XC7A15T utilizando o parâmetro TPO como 64.

Figura 41 – Comparação entre as reduções percentuais do número de ciclos entre as configurações de teste.



Fonte – Autor.

Além dos dados de utilização, foram obtidos dados relacionados à execução do acelerador. Através do projeto sintetizado, o *software* Vivado permite o cálculo dos atrasos de propagação entre os componentes do FPGA baseado em suas conectividades. A partir disso, obteve-se o valor de atraso máximo de 13,372 ns e, deste modo, é possível estimar a frequência máxima de operação do acelerador como 74,7 MHz. Além disso, esta ferramenta reportou a estimativa para a dissipação de potência do acelerador como 208 mW.

Tabela 9 – Utilização de recursos do acelerador sintetizado para FPGA.

Recurso	**Usado**	**Disponível**	**Utilização (%)**
LUT	1760	10400	18,7%
LUTRAM	179	9600	0,5%
FF	1173	20800	5,6%
DSP	27	45	60,0%
BRAM	24	25	96,0%

Fonte – Autor.

A fim de estudar a influência do parâmetro de projeto TPO na utilização dos recursos de FPGA, foram levantados dados referentes à utilização para valores de TPOs selecionados. A Tabela 10 apresenta estes resultados. Pode-se perceber que

as maiores variações no uso dos recursos se encontram no número de LUTRAM e no número de BRAM. A variação de LUTRAM deve-se ao fato de que os bancos de registradores são implementados utilizando os componentes primitivos LUTRAM presentes nos dispositivos FPGA Xilinx. Deste modo, o valor de uso deste componente é diretamente influenciado pelo valor do parâmetro TPO. De mesmo modo, o número de BRAM utilizado está relacionado com o tamanho de memória alocado para as memórias dos mapas de atributos de pesos e entradas.

Outro ponto a se destacar é de que o número de recursos DSPs utilizados permanece constante, pois, os mesmos são utilizados nos cálculos de parâmetros através dos valores fornecidos na interface MMIO e para os endereços de acesso à memória. Sendo assim, não possui relação direta com o valor do parâmetro TPO.

Tabela 10 – Utilização de recursos para diferentes TPOs.

TPO	LUT	LUTRAM	FF	DSP	BRAM
16	1721	51	1178	27	12
32	1750	51	1169	27	16
64	1760	91	1173	27	24

Fonte – Autor.

A fim de comparar o acelerador implementado com os trabalhos relacionados, foram levantados os dados da Tabela 11, onde é apresentado uma comparação de parâmetros relacionados à síntese em FPGA.

Tabela 11 – Comparação da implementação em FPGA do acelerador desenvolvido com trabalhos relacionados.

Acelerador	LUT	FF	DSP	Potência Dissipada (W)	Frequência de Operação (MHz)	FPGA
FP-BNN (LIANG, Shuang *et al.*, 2018)	230918	-	1963	26,2	150	5SGSD8
FBNA (GUO *et al.*, 2018)	29600	-	0	3,3	143	XC7Z020
Cho *et al.* (2021)	4800	1050	2	0,71	371	XCZU7EV
Xnorator (Este trabalho)	1760	1173	27	0,21	74,7	XC7A15T

Fonte – Autor.

5.5 CONSIDERAÇÕES FINAIS

Como mencionado anteriormente, uma das motivações deste acelerador é a sua utilização nos satélites pesquisados e desenvolvidos no SpaceLab (UFSC) como o FloripaSat-2 (SPACELAB, 2022).

O dispositivo FPGA utilizado para este trabalho vai ao encontro com os utilizados nos CubeSats comerciais. Por exemplo, a GomSpace, que é um dos principais fornecedores para nanosatélites e CubeSats, utiliza o dispositivo Zynq 7030 (GOMSPACE, 2022). Desse modo, a portabilidade é facilitada ao se utilizar um dispositivo da empresa Xilinx de geração semelhante.

Em relação ao consumo energético, conforme indicado na Tabela 3.5 da documentação do FloripaSat-2 (SPACELAB, 2022), a potência média dissipada em uma órbita para todos os subsistemas e cargas úteis do FloripaSat-2 é de aproximadamente 2.668 mW. No mesmo documento, está indicado que a potência de entrada média em uma órbita é de aproximadamente 2.745 mW. Logo, existe uma disponibilidade de aproximadamente 77 mW para utilização do acelerador proposto no FloripaSat-2. Considerando que os valores listados na documentação do FloripaSat-2 são preliminares, e que ao se adotar um sensor de imagem na missão, algumas das cargas úteis atuais serão removidas, logo os 208 mW previstos para o acelerador proposto poderão ser acomodados na previsão de energia da missão FloripaSat-2. Além disso, esse valor de 208 mW foi obtido ao se considerar um determinado FPGA, e poderá variar de acordo com a eficiência energética do dispositivo selecionado para a implementação final.

Sem a abordagem da computação de borda e utilização de uma rede neural no próprio satélite, seriam enviadas várias imagens que poderiam não ter significado algum. Deste modo, haveria um consumo grande de energia para a transmissão, além da banda para a comunicação. Utilizando o modelo de processamento proposto, imagens adequadas poderão ser selecionadas antes do envio para à Terra.

6 CONCLUSÃO

Esta dissertação propôs a pesquisa e desenvolvimento de um acelerador de domínio específico para redes neurais binárias. Dado que o propósito deste acelerador é ser utilizado em computação de borda, ele foi planejado buscando suprir as restrições impostas neste ambiente tais como a necessidade de um baixo consumo energético e a eficiência computacional.

A metodologia empregada foi de realizar um estudo nas redes neurais binárias do estado da arte e, a partir disso, projetar e desenvolver um acelerador para a execução da camada de convolução binária das mesmas. Um dos principais aspectos destas redes é o predominante uso de blocos residuais, de modo que a saída da operação realizada pelo acelerador não pode ser binarizada, resultando em um número maior de dados a ser transmitido para a memória.

Os aceleradores de domínio específico utilizam diferentes técnicas para ganhos de desempenho e eficiência energética como, por exemplo, especialização de dados, paralelismo de operações, utilização de uma memória local otimizada e redução do *overhead* da execução de operações aritméticas. Portanto, o acelerador implementado usufrui destas técnicas para realizar a operação de convolução binária com um melhor desempenho.

Foi montado um ambiente de simulação e realizado a validação funcional do acelerador implementado. Neste ambiente, o resultado da convolução binária computada pelo acelerador é comparada com a execução de um script em paralelo que utiliza os mesmos dados de entrada. Após sua validação, foi analisada a influência do uso deste acelerador no ambiente de simulação para a execução do algoritmo de convolução binária. Para isto, é utilizado diferentes configurações de sistemas que determinam como o algoritmo é computado.

Os resultados obtidos mostram que o acelerador executa um algoritmo de convolução binária em até 97,77% menos ciclos do que a execução utilizando apenas um núcleo RISC-V simples e em 96,06% menos ciclos do que um núcleo RISC-V com a extensão de manipulação de *bits*. Além disso, o acelerador foi sintetizado para um dispositivo FPGA Xilinx XC7A15T e o mesmo apresentou baixa utilização de componentes lógicos.

Os resultados deste trabalho demonstram que a utilização de um acelerador de domínio específico pode viabilizar o uso de algoritmos relativamente complexos como as redes neurais binárias em sistemas com limitações de consumo energético e baixo poder de processamento, tais como os encontrados em computação de borda.

Tendo em vista que a área de redes neurais binárias é relativamente recente e está em constante evolução nos últimos anos, é possível expandir e incrementar este trabalho ainda mais. Além disso, o uso de aceleradores de domínio específico

apresenta ser uma solução a ser amplamente explorada nos próximos anos para resolver as problemáticas decorrentes do fim da lei de Moore. Sendo assim, como trabalhos futuros provenientes dessa dissertação, pode-se citar:

- Execução de melhorias no *datapath* do acelerador para reduzir o uso de recursos do FPGA.
- Adição de aceleração para outras camadas além da camada de convolução binária.
- Realizar simulação com um modelo de memória mais complexa, como a DRAM.
- Obtenção de dados reais de execução e consumo em uma placa FPGA.
- Implementação nos sistemas embarcados dos satélites do SpaceLab (UFSC).

REFERÊNCIAS

ABADI, Martín *et al.* TensorFlow: A System for Large-Scale Machine Learning. *In*: 12TH USENIX symposium on operating systems design and implementation (OSDI 16). [*S.l.*: *s.n.*], 2016. p. 265–283.

AGGARWAL, Charu C *et al.* Neural networks and deep learning. **Springer**, Springer, v. 10, p. 978–3, 2018.

BANAKAR, Rajeshwari; STEINKE, Stefan; LEE, Bo-Sik; BALAKRISHNAN, Mahesh; MARWEDEL, Peter. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. *In*: IEEE. PROCEEDINGS of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No. 02TH8627). [*S.l.*: *s.n.*], 2002. p. 73–78.

BANNINK, Tom; HILLIER, Adam; GEIGER, Lukas; BRUIN, Tim de; OVERWEEL, Leon; NEEVEN, Jelmer; HELWEGEN, Koen. Larq compute engine: Design, benchmark and deploy state-of-the-art binarized neural networks. **Proceedings of Machine Learning and Systems**, v. 3, p. 680–695, 2021.

BEBES, Ahmed. **Understanding Deep Convolutional Neural Networks with a practical use-case in Tensorflow and Keras**. 2017. Disponível em: `www.ahmedbesbes.com/blog/introduction-to-cnns`. Acesso em: 7 mar. 2022.

BENGIO, Yoshua; LÉONARD, Nicholas; COURVILLE, Aaron. Estimating or propagating gradients through stochastic neurons for conditional computation. **arXiv preprint arXiv:1308.3432**, 2013.

BETHGE, Joseph; YANG, Haojin; BORNSTEIN, Marvin; MEINEL, Christoph. Back to simplicity: How to train accurate bnns from scratch? **arXiv preprint arXiv:1906.08637**, 2019.

BLOTT, Michaela; PREUSSER, Thomas B; FRASER, Nicholas J; GAMBARDELLA, Giulio; O'BRIEN, Kenneth; UMUROGLU, Yaman; LEESER, Miriam; VISSERS, Kees. FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks. **ACM Transactions on Reconfigurable Technology and Systems (TRETS)**, ACM New York, NY, USA, v. 11, n. 3, p. 1–23, 2018.

CHEN, Yu-Hsin; KRISHNA, Tushar; EMER, Joel S; SZE, Vivienne. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. **IEEE journal of solid-state circuits**, IEEE, v. 52, n. 1, p. 127–138, 2016.

CHEN, Yu-Ting; CONG, Jason; GHODRAT, Mohammad Ali; HUANG, Muhuan; LIU, Chunyue; XIAO, Bingjun; ZOU, Yi. Accelerator-rich CMPs: From concept to real hardware. *In*: IEEE. 2013 IEEE 31st International Conference on Computer Design (ICCD). [*S.l.*: *s.n.*], 2013. p. 169–176.

CHEN, Yiran; XIE, Yuan; SONG, Linghao; CHEN, Fan; TANG, Tianqi. A Survey of Accelerator Architectures for Deep Neural Networks. **Engineering**, v. 6, n. 3, p. 264–274, 2020. ISSN 2095-8099.

CHO, Jaechan; JUNG, Yongchul; LEE, Seongjoo; JUNG, Yunho. Reconfigurable binary neural network accelerator with adaptive parallelism scheme. **Electronics**, Multidisciplinary Digital Publishing Institute, v. 10, n. 3, p. 230, 2021.

CODASIP. **DOMAIN-SPECIFIC ACCELERATORS**. 2022. Disponível em: `https://codasip.com/solutions/dsa/`. Acesso em: 29 mar. 2022.

CONTI, Francesco; SCHIAVONE, Pasquale Davide; BENINI, Luca. XNOR neural engine: A hardware accelerator IP for 21.6-fJ/op binary neural network inference. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 37, n. 11, p. 2940–2951, 2018.

COTA, Emilio G; MANTOVANI, Paolo; DI GUGLIELMO, Giuseppe; CARLONI, Luca P. An analysis of accelerator coupling in heterogeneous architectures. *In*: IEEE. 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC). [*S.l.*: *s.n.*], 2015. p. 1–6.

COURBARIAUX, Matthieu; HUBARA, Itay; SOUDRY, Daniel; EL-YANIV, Ran; BENGIO, Yoshua. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. **arXiv preprint arXiv:1602.02830**, 2016.

DAGHERO, Francesco; XIE, Chen; PAGLIARI, Daniele Jahier; BURRELLO, Alessio; CASTELLANO, Marco; GANDOLFI, Luca; CALIMERA, Andrea; MACII, Enrico; PONCINO, Massimo. Ultra-compact binary neural networks for human activity recognition on RISC-V processors. *In*: PROCEEDINGS of the 18th ACM International Conference on Computing Frontiers. [*S.l.*: *s.n.*], 2021. p. 3–11.

DALLY, William J; BALFOUR, James; BLACK-SHAFFER, David; CHEN, James; HARTING, R Curtis; PARIKH, Vishal; PARK, Jongsoo; SHEFFIELD, David. Efficient embedded computing. **Computer**, IEEE, v. 41, n. 7, p. 27–32, 2008.

DALLY, William J.; TURAKHIA, Yatish; HAN, Song. Domain-Specific Hardware Accelerators. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 63, n. 7, p. 48–57, jun. 2020. ISSN 0001-0782.

DENG, Lei; LI, Guoqi; HAN, Song; SHI, Luping; XIE, Yuan. Model compression and hardware acceleration for neural networks: A comprehensive survey. **Proceedings of the IEEE**, IEEE, v. 108, n. 4, p. 485–532, 2020.

FERRARINI, Bruno; MILFORD, Michael J; MCDONALD-MAIER, Klaus D; EHSAN, Shoaib. Binary Neural Networks for Memory-Efficient and Effective Visual

Place Recognition in Changing Environments. **IEEE Transactions on Robotics**, IEEE, 2022.

FORTUNE BUSINESS INSIGHTS. **Microcontroller Market Size, Share & COVID-19 Impact Analysis, By Product Type (8-bit, 16-bit, and 32-bit Microcontroller), By Application (Networking & Communication, Automotive, Consumer Electronics, Industrial, Medical Devices, and Military & Defense) and Regional Forecast, 2021-2028**. 2022. Disponível em: https://www.fortunebusinessinsights.com/microcontroller-market-106430. Acesso em: 23 jun. 2022.

FROMM, Joshua; COWAN, Meghan; PHILIPOSE, Matthai; CEZE, Luis; PATEL, Shwetak. Riptide: Fast end-to-end binarized neural networks. **Proceedings of Machine Learning and Systems**, v. 2, p. 379–389, 2020.

GALLOWAY, Angus; TAYLOR, Graham W; MOUSSA, Medhat. Attacking binarized neural networks. **arXiv preprint arXiv:1711.00449**, 2017.

GENG, Tong; WANG, Tianqi; WU, Chunshu; YANG, Chen; SONG, Shuaiwen Leon; LI, Ang; HERBORDT, Martin. LP-BNN: Ultra-low-latency BNN inference with layer parallelism. *In*: IEEE. 2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP). [*S.l.*: *s.n.*], 2019. v. 2160, p. 9–16.

GHASEMZADEH, Mohammad; SAMRAGH, Mohammad; KOUSHANFAR, Farinaz. ReBNet: Residual binarized neural network. *In*: IEEE. 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). [*S.l.*: *s.n.*], 2018. p. 57–64.

GOMSPACE. **NanoMind Z7000**. 2022. Disponível em: https://gomspace.com/shop/support/nanomind-z7000.aspx. Acesso em: 8 ago. 2022.

GOODFELLOW, Ian; BENGIO, Yoshua; COURVILLE, Aaron. **Deep Learning**. [*S.l.*]: MIT Press, 2016. http://www.deeplearningbook.org.

GUO, Peng; MA, Hong; CHEN, Ruizhi; LI, Pin; XIE, Shaolin; WANG, Donglin. FBNA: A fully binarized neural network accelerator. *In*: IEEE. 2018 28th International Conference on Field Programmable Logic and Applications (FPL). [*S.l.*: *s.n.*], 2018. p. 51–513.

HAM, Tae Jun; WU, Lisa; SUNDARAM, Narayanan; SATISH, Nadathur; MARTONOSI, Margaret. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. *In*: IEEE. 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). [*S.l.*: *s.n.*], 2016. p. 1–13.

HAN, Song; MAO, Huizi; DALLY, William J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. **arXiv preprint arXiv:1510.00149**, 2015.

HAN, Song; POOL, Jeff; TRAN, John; DALLY, William. Learning both weights and connections for efficient neural network. **Advances in neural information processing systems**, v. 28, 2015.

HE, Kaiming; ZHANG, Xiangyu; REN, Shaoqing; SUN, Jian. Deep residual learning for image recognition. *In*: PROCEEDINGS of the IEEE conference on computer vision and pattern recognition. [*S.l.*: *s.n.*], 2016. p. 770–778.

HENNESSY, John L.; PATTERSON, David A. **Computer Architecture: A Quantitative Approach**. Waltham, MA: Morgan Kaufmann, 2017. ISBN 0128119055.

HINTON, Geoffrey; VINYALS, Oriol; DEAN, Jeff *et al.* Distilling the knowledge in a neural network. **arXiv preprint arXiv:1503.02531**, v. 2, n. 7, 2015.

HUBARA, Itay; COURBARIAUX, Matthieu; SOUDRY, Daniel; EL-YANIV, Ran; BENGIO, Yoshua. Binarized neural networks. **Advances in neural information processing systems**, v. 29, 2016.

HUBARA, Itay; COURBARIAUX, Matthieu; SOUDRY, Daniel; EL-YANIV, Ran; BENGIO, Yoshua. Quantized neural networks: Training neural networks with low precision weights and activations. **The Journal of Machine Learning Research**, JMLR. org, v. 18, n. 1, p. 6869–6898, 2017.

IOFFE, Sergey; SZEGEDY, Christian. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *In*: PMLR. INTERNATIONAL conference on machine learning. [*S.l.*: *s.n.*], 2015. p. 448–456.

JACOB, Benoit; KLIGYS, Skirmantas; CHEN, Bo; ZHU, Menglong; TANG, Matthew; HOWARD, Andrew; ADAM, Hartwig; KALENICHENKO, Dmitry. Quantization and training of neural networks for efficient integer-arithmetic-only inference. *In*: PROCEEDINGS of the IEEE conference on computer vision and pattern recognition. [*S.l.*: *s.n.*], 2018. p. 2704–2713.

KHAN, Salman; RAHMANI, Hossein; SHAH, Syed Afaq Ali; BENNAMOUN, Mohammed. A guide to convolutional neural networks for computer vision. **Synthesis Lectures on Computer Vision**, Morgan & Claypool Publishers, v. 8, n. 1, p. 1–207, 2018.

KNAG, Phil C *et al.* A 617-TOPS/W all-digital binary neural network accelerator in 10-nm FinFET CMOS. **IEEE Journal of Solid-State Circuits**, IEEE, v. 56, n. 4, p. 1082–1092, 2020.

KRIZHEVSKY, Alex; SUTSKEVER, Ilya; HINTON, Geoffrey E. Imagenet classification with deep convolutional neural networks. **Advances in neural information processing systems**, v. 25, 2012.

KROES, Mairin; PETRICA, Lucian; COTOFANA, Sorin; BLOTT, Michaela. Evolutionary bin packing for memory-efficient dataflow inference acceleration on FPGA. *In*: PROCEEDINGS of the 2020 Genetic and Evolutionary Computation Conference. [*S.l.*: *s.n.*], 2020. p. 1125–1133.

LEA, Perry. **IoT and Edge Computing for Architects: Implementing edge and IoT systems from sensors to clouds with communication systems, analytics, and security**. [*S.l.*]: Packt Publishing Ltd, 2020.

LECUN, Yann; JACKEL, Lionel D; BOSER, Brian; DENKER, John S; GRAF, Henry P; GUYON, Isabelle; HENDERSON, Don; HOWARD, Richard E; HUBBARD, William. Handwritten digit recognition: Applications of neural network chips and automatic learning. **IEEE Communications Magazine**, IEEE, v. 27, n. 11, p. 41–46, 1989.

LI, He; OTA, Kaoru; DONG, Mianxiong. Learning IoT in edge: Deep learning for the Internet of Things with edge computing. **IEEE network**, IEEE, v. 32, n. 1, p. 96–101, 2018.

LIANG, Shuang; YIN, Shouyi; LIU, Leibo; LUK, Wayne; WEI, Shaojun. FP-BNN: Binarized neural network on FPGA. **Neurocomputing**, Elsevier, v. 275, p. 1072–1086, 2018.

LIANG, Songhong; LIN, Yingcheng; HE, Wei; ZHANG, Ling; WU, Mingmei; ZHOU, Xichuan. An energy-efficient bagged binary neural network accelerator. *In*: IEEE. 2020 IEEE 3rd International Conference on Electronics Technology (ICET). [*S.l.*: *s.n.*], 2020. p. 174–179.

LIU, Zechun; SHEN, Zhiqiang; LI, Shichao; HELWEGEN, Koen; HUANG, Dong; CHENG, Kwang-Ting. How do adam and training strategies help bnns optimization. *In*: PMLR. INTERNATIONAL Conference on Machine Learning. [*S.l.*: *s.n.*], 2021. p. 6936–6946.

LIU, Zechun; WU, Baoyuan; LUO, Wenhan; YANG, Xin; LIU, Wei; CHENG, Kwang-Ting. Bi-real net: Enhancing the performance of 1-bit cnns with improved representational capability and advanced training algorithm. *In*: PROCEEDINGS of the European conference on computer vision (ECCV). [*S.l.*: *s.n.*], 2018. p. 722–737.

LOWRISC. **Ibex Documentation**. 2022. Disponível em: `https://ibex-core.readthedocs.io`. Acesso em: 1 jul. 2022.

MITXPC. **Difference between Training and Inference of Deep Learning Frameworks**. 2022. Disponível em: `https://mitxpc.com/pages/ai-inference-applying-deep-neural-network-training`. Acesso em: 8 mar. 2022.

MOORE, Gordon E *et al.* **Cramming more components onto integrated circuits**. [*S.l.*]: McGraw-Hill New York, 1965.

MUCHANDI, Shruthi K. **Enabling Accelerator-soc Co-design Using RISC-V Chipyard Framework**. 2020. Tese (Doutorado) – The University of North Carolina at Charlotte.

ON SEMICONDUCTOR. **FPGA-to-ASIC Conversion**. 2021. Disponível em: `https://www.onsemi.com/products/product-taxonomy/soc-sip-custom-products/fpga-to-asic-conversion`. Acesso em: 7 mar. 2022.

PARKS, Michael. **Edge Computing and the Internet of Things**. 2018. Disponível em: `https://www.mouser.com/blog/edge-computing-and-the-internet-of-things`. Acesso em: 9 abr. 2022.

PASZKE, Adam *et al.* Pytorch: An imperative style, high-performance deep learning library. **Advances in neural information processing systems**, v. 32, 2019.

PATEL, Vishal M; GOPALAN, Raghuraman; LI, Ruonan; CHELLAPPA, Rama. Visual domain adaptation: A survey of recent advances. **IEEE signal processing magazine**, IEEE, v. 32, n. 3, p. 53–69, 2015.

PATTERSON, David A.; HENNESSY, John L. **Computer Organization and Design RISC-V Edition: The Hardware Software Interface**. Cambridge, MA: Morgan Kaufmann, 2020. ISBN 0128203315.

POEL, Jan Van de. **Deep Learning - A Definition**. 2020. Disponível em: `https://seeme.ai/blog/deep-learning-a-definition/`. Acesso em: 29 abr. 2022.

PREMSANKAR, Gopika; DI FRANCESCO, Mario; TALEB, Tarik. Edge computing for the Internet of Things: A case study. **IEEE Internet of Things Journal**, IEEE, v. 5, n. 2, p. 1275–1284, 2018.

QADEER, Wajahat; HAMEED, Rehan; SHACHAM, Ofer; VENKATESAN, Preethi; KOZYRAKIS, Christos; HOROWITZ, Mark. Convolution Engine: Balancing Efficiency and Flexibility in Specialized Computing. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 58, n. 4, p. 85–93, mar. 2015. ISSN 0001-0782.

RAJ, Vishnu; NAYAK, Nancy; KALYANI, Sheetal. **Understanding Learning Dynamics of Binary Neural Networks via Information Bottleneck**. [*S.l.*]: arXiv, 2020. Disponível em: `https://arxiv.org/abs/2006.07522`.

RASTEGARI, Mohammad; ORDONEZ, Vicente; REDMON, Joseph; FARHADI, Ali. Xnor-net: Imagenet classification using binary convolutional neural networks. *In*: SPRINGER. EUROPEAN conference on computer vision. [*S.l.*: *s.n.*], 2016. p. 525–542.

RUSSAKOVSKY, Olga *et al.* Imagenet large scale visual recognition challenge. **International journal of computer vision**, Springer, v. 115, n. 3, p. 211–252, 2015.

RUSSO, Enrico; PALESI, Maurizio; MONTELEONE, Salvatore; PATTI, Davide; MINEO, Andrea; ASCIA, Giuseppe; CATANIA, Vincenzo. DNN Model Compression for IoT Domain Specific Hardware Accelerators. **IEEE Internet of Things Journal**, IEEE, 2021.

SAINATH, Tara N; KINGSBURY, Brian; SINDHWANI, Vikas; ARISOY, Ebru; RAMABHADRAN, Bhuvana. Low-rank matrix factorization for deep neural network training with high-dimensional output targets. *In*: IEEE. 2013 IEEE international conference on acoustics, speech and signal processing. [*S.l.*: *s.n.*], 2013. p. 6655–6659.

SAMIE, Farzad; BAUER, Lars; HENKEL, Jörg. From cloud down to things: An overview of machine learning in internet of things. **IEEE Internet of Things Journal**, IEEE, v. 6, n. 3, p. 4921–4934, 2019.

SCHIAVONE, Pasquale Davide; CONTI, Francesco; ROSSI, Davide; GAUTSCHI, Michael; PULLINI, Antonio; FLAMAND, Eric; BENINI, Luca. Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications. *In*: IEEE. 2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS). [*S.l.*: *s.n.*], 2017. p. 1–8.

SHAWAHNA, Ahmad; SAIT, Sadiq M; EL-MALEH, Aiman. FPGA-based accelerators of deep learning networks for learning and classification: A review. **ieee Access**, IEEE, v. 7, p. 7823–7859, 2018.

SHEPARD, Jeff. **Domain specific accelerators for RISC-V**. 2021. Disponível em: `https://www.microcontrollertips.com/domain-specific-accelerators-for-risc-v-faq/`. Acesso em: 29 mar. 2022.

SHI, Weisong; DUSTDAR, Schahram. The promise of edge computing. **Computer**, IEEE, v. 49, n. 5, p. 78–81, 2016.

SIMONYAN, Karen; ZISSERMAN, Andrew. Very deep convolutional networks for large-scale image recognition. **arXiv preprint arXiv:1409.1556**, 2014.

SORO, Stanislava. Tinyml for ubiquitous edge ai. **arXiv preprint arXiv:2102.01255**, 2021.

SPACELAB. **FloripaSat-2 Documentation**. Space Technology Research Laboratory. 2022. Disponível em: `https://github.com/spacelab-ufsc/floripasat2-doc`. Acesso em: 8 ago. 2022.

SUN, Baohua; YANG, Lin; DONG, Patrick; ZHANG, Wenhan; DONG, Jason; YOUNG, Charles. Ultra power-efficient cnn domain specific accelerator with 9.3 tops/watt for mobile and embedded applications. *In*: PROCEEDINGS of the IEEE Conference on Computer Vision and Pattern Recognition Workshops. [*S.l.*: *s.n.*], 2018. p. 1677–1685.

SZE, Vivienne; CHEN, Yu-Hsin; EMER, Joel; SULEIMAN, Amr; ZHANG, Zhengdong. Hardware for machine learning: Challenges and opportunities. **2017 IEEE Custom Integrated Circuits Conference (CICC)**, IEEE, abr. 2017.

SZE, Vivienne; CHEN, Yu-Hsin; YANG, Tien-Ju; EMER, Joel S. Efficient processing of deep neural networks: A tutorial and survey. **Proceedings of the IEEE**, IEEE, v. 105, n. 12, p. 2295–2329, 2017.

SZEGEDY, Christian; LIU, Wei; JIA, Yangqing; SERMANET, Pierre; REED, Scott; ANGUELOV, Dragomir; ERHAN, Dumitru; VANHOUCKE, Vincent; RABINOVICH, Andrew. Going deeper with convolutions. *In*: PROCEEDINGS of the IEEE conference on computer vision and pattern recognition. [*S.l.*: *s.n.*], 2015. p. 1–9.

THEIS, Thomas N; WONG, H-S Philip. The end of moore's law: A new beginning for information technology. **Computing in Science & Engineering**, IEEE, v. 19, n. 2, p. 41–50, 2017.

UMUROGLU, Yaman; FRASER, Nicholas J.; GAMBARDELLA, Giulio; BLOTT, Michaela; LEONG, Philip; JAHRE, Magnus; VISSERS, Kees. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. *In*: PROCEEDINGS of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. [*S.l.*]: ACM, 2017. (FPGA '17), p. 65–74.

URQUHART, Roddy. **The Road To Domain-Specific Accelerators**. 2021. Disponível em: `https://semiengineering.com/the-road-to-domain-specific-accelerators/`. Acesso em: 29 mar. 2022.

VÉSTIAS, Mário P; DUARTE, Rui Policarpo; SOUSA, José T de; NETO, Horácio C. Moving deep learning to the edge. **Algorithms**, Multidisciplinary Digital Publishing Institute, v. 13, n. 5, p. 125, 2020.

VILLASENOR, John; MANGIONE-SMITH, William H. Configurable computing. **Scientific American**, JSTOR, v. 276, n. 6, p. 66–71, 1997.

WANG, Xiaofei; HAN, Yiwen; LEUNG, Victor CM; NIYATO, Dusit; YAN, Xueqiang; CHEN, Xu. **Edge AI: Convergence of edge computing and artificial intelligence**. [*S.l.*]: Springer, 2020.

WARREN, Henry S. **Hacker's Delight**. 2nd. [*S.l.*]: Addison-Wesley Professional, 2012. ISBN 0321842685.

WATERMAN, Andrew; LEE, Yunsup; PATTERSON, David A.; ASANOVIC, Krste; ISA, Volume I User-level; WATERMAN, Andrew; LEE, Yunsup; PATTERSON, David. **The RISC-V Instruction Set Manual**. [*S.l.*: *s.n.*], 2014.

WEN, Wei; WU, Chunpeng; WANG, Yandan; CHEN, Yiran; LI, Hai. Learning structured sparsity in deep neural networks. **Advances in neural information processing systems**, v. 29, 2016.

WU, Lisa; LOTTARINI, Andrea; PAINE, Timothy K; KIM, Martha A; ROSS, Kenneth A. Q100: The architecture and design of a database processing unit. **ACM SIGARCH Computer Architecture News**, ACM New York, NY, USA, v. 42, n. 1, p. 255–268, 2014.

WU, Lisa; WEAVER, Chris; AUSTIN, Todd. CryptoManiac: A fast flexible architecture for secure communication. *In*: IEEE. PROCEEDINGS 28th Annual International Symposium on Computer Architecture. [*S.l.*: *s.n.*], 2001. p. 110–119.

XILINX. **7 Series FPGAs Data Sheet: Overview (DS180)**. [*S.l.*], set. 2020.

YANG, Haojin; FRITZSCHE, Martin; BARTZ, Christian; MEINEL, Christoph. Bmxnet: An open-source binary neural network implementation based on mxnet. *In*: PROCEEDINGS of the 25th ACM international conference on Multimedia. [*S.l.*: *s.n.*], 2017. p. 1209–1212.

YUAN, Chunyu; AGAIAN, Sos S. A comprehensive review of Binary Neural Network. **arXiv preprint arXiv:2110.06804**, 2021.

ZEILER, Matthew D; FERGUS, Rob. Visualizing and understanding convolutional networks. *In*: SPRINGER. EUROPEAN conference on computer vision. [*S.l.*: *s.n.*], 2014. p. 818–833.

ZHANG, Chen; LI, Peng; SUN, Guangyu; GUAN, Yijin; XIAO, Bingjun; CONG, Jason. Optimizing FPGA-based accelerator design for deep convolutional neural networks. *In*: PROCEEDINGS of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays. [*S.l.*: *s.n.*], 2015. p. 161–170.

ZHANG, Yichi; ZHANG, Zhiru; LEW, Lukasz. PokeBNN: A Binary Pursuit of Lightweight Accuracy. **arXiv preprint arXiv:2112.00133**, 2021.

ANEXO A – SCRIPT BIN-CONV.PY

```
#!/bin/env python3

import numpy as np

# params to specify
iw = 7
ih = iw
ic = 64
oc = 64
padding = False
input_mem_file = "input.vmem"
weight_mem_file = "weights.vmem"
output_mem_file = "output-script.vmem"

# constants
ks = 3


# util functions
def hex2int(i):
    return int(i, 16)


def popcount(i):
    return sum(b == '1' for b in i)


def normalize(i):
    return (i*2) - 32


def int2hex16b(number):
    """ Return the 2'complement hexadecimal representation of a number
        """
    if number < 0:
        return f'{(1 << 16) + number:04X}'
    else:
        return f'{number:04X}'


hex2int_vec = np.vectorize(hex2int)
popcount_vec = np.vectorize(popcount)
normalize_vec = np.vectorize(normalize)
bin_repr_vec = np.vectorize(lambda x: np.binary_repr(
    x, width=32)[-32:])  # enforce width
```

```
hex_and_merge_str_vec = np.vectorize(lambda a, b: int2hex16b(a)+
    int2hex16b(b))


def binconv(a, b):
    x = np.bitwise_xor(hex2int_vec(a), hex2int_vec(b))
    y = np.bitwise_not(x)
    z = bin_repr_vec(y)
    return normalize_vec(popcount_vec(z)).sum()


input = []
with open(input_mem_file) as f:
    input = [line[:-1] for line in f.readlines()]

weight = []
with open(weight_mem_file) as f:
    weight = [line[:-1] for line in f.readlines()]

input_matrix = np.reshape(
    input[:iw*ih*int(ic/32)], (iw, ih, int(ic/32)), order='F')
weight_matrix = np.reshape(
    weight[:ks*ks*int(ic/32)*oc], (ks, ks, int(ic/32), oc), order='F')


if padding:
    oh = ih
    ow = iw

    vertical_fill = np.array(
        ["0" for i in range(0, (iw+2)*int(ic/32))]).reshape((iw+2, 1,
            int(ic/32)))
    horizontal_fill = np.array(
        ["0" for i in range(0, ih*int(ic/32))]).reshape((1, ih, int(ic
            /32)))
    hpadded_matrix = np.concatenate(
        (horizontal_fill, input_matrix, horizontal_fill), axis=0)
    padded_matrix = np.concatenate(
        (vertical_fill, hpadded_matrix, vertical_fill), axis=1)
    input_matrix = padded_matrix
else:
    oh = ih-2
    ow = iw-2


out = []
for n in range(0, oc):
```

```
    for j in range(0, oh):
        for i in range(0, ow):
            out.append(
                binconv(input_matrix[i:i+ks, j:j+ks, :], weight_matrix
                    [:, :, :, n]))

out_matrix = np.reshape(out, (ow, oh, oc),  order='F')

output_array_hex = hex_and_merge_str_vec(
    out_matrix[:, :,  1::2], out_matrix[:, :,  0::2])

with open(output_mem_file, 'w') as f:
    f.write('\n'.join(output_array_hex.reshape(-1, order='F')) + '\n')
```

ANEXO B – SCRIPT GEN-DATA.PY

```
#!/bin/env python3

from random import randint

# params to specify
iw = 56
ih = iw
ic = 64
oc = 64
padding = False
c_header_file = "sw/simple-mmio/test-data.h"
input_mem_file = "input.vmem"
weight_mem_file = "weights.vmem"

# constants
ks = 3

# util functions


def int2hex(number, bits):
    """ Return the 2'complement hexadecimal representation of a number
        """
    if number < 0:
        return hex((1 << bits) + number)
    else:
        return hex(number)


ow = iw if padding else iw-2
oh = ih if padding else ih-2

input_size = iw*ih*int(ic/32)
weight_size = ks*ks*int(ic/32)*oc
output_size = ow*oh*int(oc/2)


def gen_c_array_text(name, str_array):
    data = []
    data.append(f'uint32_t {name}[] = {{')
    data += [(f"0x{str}" + ", ") for str in str_array]
    data.append('};\n')
    return '\n'.join(data)
```

```
def gen_c_empty_array_text(name, size):
    data = []
    data.append(f'uint32_t {name}[{size}] = {{}};')
    data.append(f'uint32_t output_size = {size};\n')
    return '\n'.join(data)


input_array_str = [f"{randint(0, 2**32-1):08X}" for x in range(0,
   input_size)]
weight_array_str = [f"{randint(0, 2**32-1):08X}"
                    for x in range(0, weight_size)]

with open(c_header_file, 'w') as f:
    input_text = gen_c_array_text("input", input_array_str)
    weight_text = gen_c_array_text("weights", weight_array_str)
    output_text = gen_c_empty_array_text("output", output_size)
    file_text = input_text + weight_text + output_text
    f.write(file_text)

with open(input_mem_file, 'w') as f:
    f.write('\n'.join(input_array_str) + '\n')
with open(weight_mem_file, 'w') as f:
    f.write('\n'.join(weight_array_str) + '\n')
```