Universidade do Estado do Rio de Janeiro
Centro de Tecnologia e Ciências
Faculdade de Engenharia

Alexandre Nietupski Cardoso

Implementação de Redes Neurais Convolucionais em Plataforma de Rede Intra-chip

Rio de Janeiro
2022

Alexandre Nietupski Cardoso

Implementação de Redes Neurais Convolucionais em Plataforma de Rede Intra-chip

Dissertação apresentada, como requisito parcial para obtenção do título de Mestre, ao Programa de Pós-Graduação em Engenharia Eletrônica, da Universidade do Estado do Rio de Janeiro. Área de concentração: Sistemas Inteligentes e Automação.

Orientadora: Prof^a. Dr^a. Luiza de Macedo Mourelle, Ph.D.
Orientadora: Prof^a. Dr^a. Nadia Nedjah, Ph.D.

Rio de Janeiro
2022

CATALOGAÇÃO NA FONTE
UERJ / REDE SIRIUS / BIBLIOTECA CTC/B

C268 Cardoso, Alexandre Nietupski.
Implementação de redes neurais convolucionais em plataforma de rede intra-chip / Alexandre Nietupski Cardoso. – 2022.
102 f.

Orientadoras: Luiza de Macedo Mourelle, Nadia Nedjah.
Dissertação (Mestrado) – Universidade do Estado do Rio de Janeiro, Faculdade de Engenharia.

1. Engenharia Eletrônica – Teses. 2. Aprendizado do computador – Teses. 3. Redes neurais (Computação) – Teses. 4. Sistemas programáveis em chip – Teses. I. Mourelle, Luiza de Macedo. II. Nedjah, Nadia. III. Universidade do Estado do Rio de Janeiro, Faculdade de Engenharia. IV. Título.

CDU 004.89

Bibliotecária: Júlia Vieira – CRB7/6022

Assinatura

Data

Alexandre Nietupski Cardoso

Implementação de Redes Neurais Convolucionais em Plataforma de Rede Intra-chip

Dissertação apresentada, como requisito parcial para obtenção do título de Mestre, ao Programa de Pós-Graduação em Engenharia Eletrônica, da Universidade do Estado do Rio de Janeiro. Área de concentração: Sistemas Inteligentes e Automação.

Aprovado em: 16 de Dezembro de 2022
Banca Examinadora:

Prof[a]. Dr[a]. Luiza de Macedo Mourelle, Ph.D. (Orientadora)
Faculdade de Engenharia, UERJ

Prof[a]. Dr[a]. Nadia Nedjah, Ph.D. (Orientadora)
Faculdade de Engenharia, UERJ

Prof. Dr. José Gabriel Rodrigues C. Gomez, Ph.D.
Escola Politécnica, UFRJ

Prof[a]. Dr[a]. Maria Clícia Stelling de Castro, D.Sc.
Instituto de Matemática e Estatística, UERJ

Rio de Janeiro
2022

AGRADECIMENTOS

Muitas pessoas contribuíram e contribuem, sob variadas formas, na construção e consecução de projetos, trabalhos e entregas como essa, e não é possível listar a todas em uma mísera lauda. É certo, de todo modo, que nada fazemos sozinhos e, pessoalmente, tenho a sorte de constatar isso mais e melhor, a cada tique que o relógio dá.

Agradeço à Prof^a. Dr^a. Luiza de Macedo Mourelle por todo o empenho, auxílio e paciência despendidos em orientação e aconselhamento a este aluno, ao longo deste trabalho. É certo que aquele que se posta à proa do barco para melhor ver e indicar o bom rumo, na bonança ou na tempestade, merece todo o reconhecimento e reverência.

Agradeço ao Prof. e amigo Paulo Sérgio Rodrigues Alonso, o patrocinador e conselheiro mais fiel, honrado e cavalheiro que alguém pode ter. Se é nos ombros das pessoas consistentes, obreiras e positivamente obstinadas que a mecânica do mundo se apoia, está neste Atlas o seu representante mais justo.

Agradeço ao Prof. Jorge Duarte Pires Valério, cuja aposta permanente neste aluno, seu conselho e seu patrocínio são benesses preciosas e seu valor, uma constatação endossada por milhares de outros "de sorte" como eu.

Agradeço à Prof^a. Cristiana Bentes, uma das primeiras pessoas a incentivar este discente a considerar dar um passo adiante, quando finda a Graduação.

Agradeço ao Prof. Orlando Bernardo Filho (*in memoriam*), que teve papéis centrais no auxílio a este aluno em dar seguimento a seu desejo basilar de seguir estudando.

Agradeço à Prof^a. Dr^a. Nadia Nedjah que, a seu próprio modo, se empenha em auxiliar seus alunos e orientandos a encontrar suas melhores versões, bem como em assentar sobre bases sólidas os projetos e iniciativas onde pousa sua chancela. Cada comentário ou sugestão, posso assegurar, não foi menos do que útil.

Agradeço ao pesquisador Alexandre Bazyl Zacarias de Souza, que gentilmente nos ajudou cedendo-nos um conjunto de pesos treinados para aplicarmos à nossa implementação da LeNET-5. Seu esforço com CNNs consta como uma importante referência nossa.

Agradeço a Carlos Eduardo Torres Padilha e a Márcio Albernaz de Mello. Nem todo mundo conta com gestores que imediatamente reconhecem o intento e procuram auxiliar seus colaboradores na perseguição de suas propostas pessoais de valor, como continuar seus estudos.

A Ernesto Adolfo Elias Paiva (*in memoriam*), a quem dedico este humilde trabalho. Vivia dizendo que este seu amigo era muito capaz quando, na verdade, mal entendia que era ele, sim, gigante e genial.

A meus pais e avós, por motivos óbvios.

Não foi um comando, nem um aviso... Apenas uma simples e calma pergunta:
"você planeja mesmo voar para dentro desta montanha..?"

Trecho de "O Dom de Voar", de Richard Bach

RESUMO

CARDOSO, Alexandre N. *Implementação de Redes Neurais Convolucionais em Plataforma de Rede Intra-chip*. 102f. Dissertação (Mestrado em Engenharia Eletrônica) – Faculdade de Engenharia, Universidade do Estado do Rio de Janeiro (UERJ), Rio de Janeiro, 2022.

O interesse por abordagens aceleradas para a execução de modelos de Aprendizado de Máquina é grande, posto que a aplicação de técnicas atinentes ao campo em problemas os mais diversos é um clamor de mercado e uma tendência de indústria. Por outro lado, o avanço das tecnologias de integração permite agregar aos sistemas embutidos multiprocessados uma quantidade crescente de módulos que, por sua vez, dão aos *chips* capacidade computacional estupenda, tornando-as atrativas à acomodação de aplicações de Inteligência Artificial. Mas estes módulos devem ser interconectados eficientemente e é nisso que as Redes Intra-chip vêm a contribuir, habilitando a concepção de sistemas embarcados extremamente capazes e versáteis. Neste trabalho, dissertamos sobre a implementação de uma rede neural convolucional em um sistema embutido multiprocessado. Nossa ênfase estava na organização da implementação à luz do fluxo de dados da aplicação, visando tirar o melhor proveito possível do processamento paralelo. Limitações relacionadas ao suporte à aritmética de ponto flutuante no ambiente de simulação escolhido prejudicaram o desempenho de nossa implementação enquanto modelo preditivo, porém não nos impediu de atingir nosso intento: acelerar a execução de uma rede neural convolucional.

Palavras-chave: Redes intra-chip. Redes neurais convolucionais. Aprendizado de máquina.

ABSTRACT

CARDOSO, Alexandre N. *Convolutional Neural Networks Implementation on a Network-on-Chip Platform*. 102p. Dissertation (Postgraduate in Electronics Engineering) – Faculty of Engineering, State University of Rio de Janeiro (UERJ), Rio de Janeiro, 2022.

The interest in accelerative approaches for executing Machine Learning models is intense, since applying methods from this field to a growing set of situations is a market demand and an industry tendency. On the other hand, the advance of integration technologies allows aggregating to multiprocessor embedded systems a growing number of modules and that gives to these systems enormous computational capacity, making it attractive for Artificial Intelligence solutions. However, these modules shall be connected efficiently and this is where Networks on Chip come to help, enabling the design of extremely capable and versatile embedded systems. In this work, we discuss the implementation of a convolutional neural network in a multiprocessor embedded system. Our emphasis was on organizing the implementation considering the application's data flow, pursuing the best possible use of parallel computing. Limitations related to the support for floating-point arithmetic in the chosen simulation environment restricted the performance of our implementation as a predictive model, but it did not prevent us from achieving our goal: to accelerate the execution of a convolutional neural network.

Keywords: Networks on chip. Convolutional neural networks. Machine learning.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

ALO	Ant Lion Optimization
ALU	Arithmetic-Logic Unit
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuits
BNN	Binarized Neural Network
CMP	Chip Multiprocessor
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DMA	Direct Memory Access
DOR	Dimension Order Routing
DRAM	Dynamic Randomic Access Memory
DS	Diamond Search
DSM	Deep Sub-Micron Domain
EDA	Electronic Design Automation
FEC	Forward Error Correction
FPGA	Field-Programmable Gate Arrays
GPL	General Purpose License
GPU	Graphic Processing Units
HEMPS	Hermes Multiprocessor System on Chip
HoL	Head-on-Line
IP	Internet Protocol
IPA	International Phonetic Alphabet
MEMPHIS	Many-core Modeling Platform for Heterogenous Systems on Chip
MLP	Multi-layer Perceptron
MMC	Menor Múltiplo Comum
NoC	Network on Chip
PE	Processing Element
PSO	Particle Swarm Optimization
RBF	Euclidean Radial-Basis Function
RNA	Rede Neural Artificial
SAF	Store and Forward
SIMD	Single Instruction, Multiple Data
SoC	System on Chip
SRAM	Static Random-Access Memory
STT-RAM	Spin-Torque Transfer Magnetic RAM
TCP	Transfer Control Protocol
TtM	Time to Market
VCT	Virtual Cut Through
VHDL	VSIC Hardware Description Language
VSIC	Very High Speed Integrated Circuit
WH	Wormhole
YAML	Yet Another Markup Language

SUMÁRIO

INTRODUÇÃO

A cognição humana constitui, em si, uma maravilha e um milagre. Apreciada e estudada de forma intensa, mesmo que ainda longe de exaustiva, é hoje um pouco melhor compreendida mas seus limites constituem um mistério: ela é dada e capaz de transitar por domínios complexos, como os das Artes, da Filosofia e das Ciências, assim como viabilizar competência em situações e problemas considerados triviais, como caminhar, reconhecer e manipular objetos ou conversar. Mesmo a tipificação e a antecipação de fenômenos naturais ou de consequências de um dado cenário apreensível por meio da percepção ou do pensamento abstrato parecem estar relativamente acessíveis aos seres vivos com alguma capacidade cognitiva e sensorial, manifesta de forma consciente ou como intuição ou instinto. Neste sentido, nenhuma surpresa que seja a cognição – frequentemente, a humana – uma fonte de inspiração e uma referência comparativa em áreas de estudo que têm a pretensão de construir ou inferir modelos cognitivos, sejam estes atinentes a outras espécies, sejam estes a construtos artificiais. Assim, na tentativa de abordar problemas considerados "complexos", a Ciência da Computação frequentemente busca, nas Ciências Cognitivas, respaldo e bases conceituais que permitam fazer emergir ideias que auxiliem na obtenção de soluções para questões que encontrem afinidade com estas supracitadas, de algum modo.

Em verdade, entre as décadas de 1940 e 1990, enquanto florescia e amadurecia a ciência chamada "Inteligência Artificial", havia grande interesse em abordar, por meios computacionais, problemas inerentemente difíceis de serem solucionados por seres humanos mas relativamente simples de se tratar sob uma ótica computacional (GOODFELLOW; BENGIO; COURVILLE, 2016). Ali, o escopo seria os problemas eventualmente passíveis de abordagem por meio de algum formalismo matemático bem definido. Assim, à medida em que estes problemas foram sendo progressivamente resolvidos, restava um desafio maior: dar tratamento a questões que, por sua vez, seriam absolutamente triviais a qualquer ser humano, mas de grande dificuldade de representação formal e, consequentemente, de di-

fícil tratamento por máquinas: reconhecer um rosto, interpretar uma frase dita, aprender autonomamente por experiência.

A intenção de se construir máquinas ditas "inteligentes" – aqui, como sendo as capazes de apresentar comportamento que possa ser tipificado como racional – ganha especial fôlego na década de 1960, quando surgem as ideias acerca das Redes Neurais Artificiais (SILVA; SPATTI; FLAUZINO, 2010). Inspiradas na Neurologia e motivadas pela premissa de que modelos que tivessem, em sua base, a mimetização simplificada de aspectos biológicos relacionados à cognição (não necessariamente a humana), inicia-se a proposição de soluções em torno de "neurônios artificiais", sobretudo aplicáveis em problemas de classificação e predição categórica. Porém, é somente na década de 1990 que as ditas Redes Neurais Artificiais (RNAs) começam a ganhar, rapidamente, o interesse da comunidade da Inteligência Artificial. É nesta época em que o desenvolvimento da indústria de semicondutores e o crescimento da capacidade de aquisição e compilação de grandes bases de dados, eventualmente associadas a um problema ou escopo, habilitam o surgimento de aplicações efetivamente passíveis de apreciação por meio de RNAs, porque normalmente demandam grandes quantidades de dados de entrada e, também, computação intensa. Ainda assim, é na década de 2010, quando os sistemas computacionais amplamente disponíveis detém capacidade computacional e armazenamento mais amplo, que a disseminação, estudo e adoção ampla das RNAs em aplicações variadas ganha fôlego.

As RNAs seriam modelos computacionais inspirados no sistema nervoso dos seres vivos, com capacidade de aquisição e manutenção de um corpo de conhecimento e definido por um conjunto de unidades de processamento, doravante nomeadas "neurônios artificiais", interligados por um conjunto grande de interconexões, normalmente representadas por vetores ou matrizes de pesos a elas associadas, caracterizando as chamadas "sinapses artificiais" (SILVA; SPATTI; FLAUZINO, 2010). A ideia primária, sob um contexto de Aprendizado Supervisionado, é que se possa "treinar" tal arranjo mediante a apresentação de um conjunto conhecido de fatos ou observações, visando ajustar estes pesos de maneira que o sistema possa, *a posteriori*, intuir os valores das variáveis-objetivo do corpo amostral em questão autonomamente, em casos "novos", diferentes dos apresentados durante a fase de ajuste do modelo.

Entretanto, há uma grande dificuldade no concernente à obtenção de atributos a partir de conjuntos de dados atinentes a problemas reais, sobretudo porque estão sujeitos

a variações ou modificações causadas por fatores que podem não ser de nosso interesse, ou porque colecioná-los não é realmente trivial (GOODFELLOW; BENGIO; COURVILLE, 2016). Em se tratando de imagens por exemplo, uma informação de cor pode ser modificada pelo grau de luz ali presente, de maneira que, em uma cena escura, o vermelho mais vivo pode se mostrar quase preto. É neste contexto em que começa a ganhar destaque a classe de soluções a que chamamos "Aprendizagem Profunda", onde a emersão de conhecimento, no sistema, se dá pela agregação de conceitos mais simples, em uma sistemática hierarquizada onde se pode tratar, inclusive, a questão da variabilidade dos dados de entrada, de forma, grosso modo, automática.

É sob este contexto que surgem, então, as ditas Redes Neurais Convolucionais. Por Redes Neurais Convolucionais (*Convolutional Neural Networks* - CNNs), entendemos a subclasse de Redes Neurais de Aprendizado Profundo, denominação que, em termos simples, nomeia a classe de Redes Neurais caracterizadas pela emersão autônoma de atributos de entrada e por sua arquitetura, que tipicamente envolve muitas camadas [1]. Em sua essência, as CNNs são especializadas no processamento de dados eventualmente organizados em estruturas matriciais com dimensões conhecidas *a priori*. Na figura 1 há dois exemplos de sinais de entrada que poderiam ser o escopo de uma CNN: uma imagem[2] e um espectrograma temporal de áudio[3].

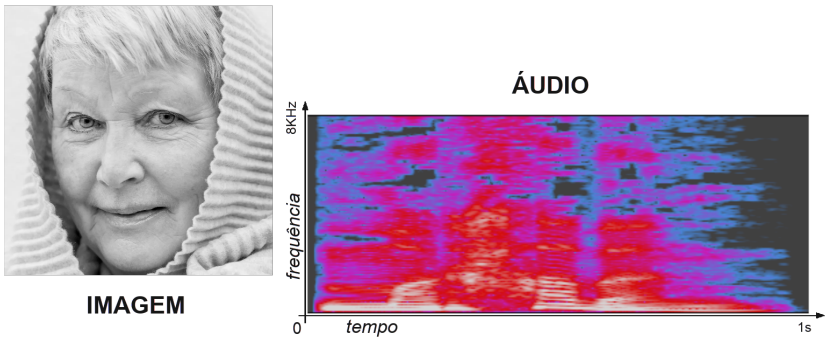


Figura 1: Exemplos de sinais de entrada aplicáveis a uma CNN

[1] Aqui, nos referimos às camadas *escondidas* ou *intermediárias* de uma RNA.

[2] A modelo é Lena Forsen, que tornou-se famosa na comunidade de Processamento de Imagens quando, em 1972, um de seus trabalhos fotográficos foi utilizado por Alexander Sawchuk e sua equipe, no Instituto de Processamento de Imagens e Sinais na Universidade do Sul da Califórnia, em testes preliminares no desenvolvimento daquilo que viria a se tornar, anos mais tarde, o padrão JPEG de compressão de imagens (KINSTLER, 2019).

[3] Representando a intensidade das componentes frequenciais, no tempo, do registro de voz de um homem adulto pronunciando o termo, em árabe, para "seja bem-vindo" (IPA: /mar'haban/).

Contextualizado o que seria uma Rede Neural Convolucional, é oportuno mencionarmos as Redes Intra-chip. Uma "rede intra-chip" (*Network on Chip* - NoC) é uma rede de comunicação orientada ao chaveamento de pacotes visando interconectar módulos em um "sistema integrado em um *chip*" (*System on Chip* - SoC) (PASRICHA; DUTT, 2008). As NoCs são uma alternativa a sistemas de interconexão amplamente utilizados em circuitos integrados desde muito tempo, como barramentos, trazendo potencial melhoria de desempenho e eficiência energética. A razão para tanto está relacionada à menor sujeição, das NoCs, a problemas comuns dos meios ditos "estáticos", como atrasos decorrentes de latência de propagação ou disputa por acesso ao meio. Este tipo de tecnologia tem sido crucial no desenvolvimento de circuitos integrados por conjugar oportunidades em processamento paralelo (e consequente aceleração) com eficiência energética; nenhuma surpresa quando as NoCs se mostram protagonistas em discussões atinentes ao tema, como a do "silício escuro" [4], por exemplo.

Assim, as NoCs possibilitam não somente a tramitação de fluxos de dados de maneira mais eficiente, mas também a observância de restrições várias impostas pelo aumento da complexidade dos circuitos integrados de maneira geral, bem como ao incremento das frequências de *clock* sob o qual operam. Sobre o primeiro, há que a tramitação de dados entre Elementos Processadores no sistema já não mais se restringe à disponibilidade do meio físico quando do momento de transmitir pois, em lugar de um arranjo do tipo barramento e *buffer*, teríamos uma estrutura a transportar os dados previamente encapsulados em pacotes por interfaces de interconexão presentes nos Elementos Processadores à malha de dispositivos de roteamento – as chaves de interconexão – ligadas por vias de transmissão curtas. Assim, não somente é possível realizar transmissões simultâneas entre módulos do sistema como, também, admitem-se hipóteses como alterações dinâmicas das rotas dos pacotes, a depender das condições operacionais das chaves de interconexão, que tramitam estes pacotes até o seu destino. Um aspecto igualmente interessante é o quanto as NoCs podem simplificar o esforço no projeto de um circuito integrado, facilitando, mesmo, sua modificação para a concepção de novas versões do produto, bem como reduzindo seu tempo de lançamento no mercado (*Time to Market* - TtM). Sobretudo em contextos sob grande pressão mercadológica e de inovação, as Redes Intra-chip consti-

[4] *Dark silicon*: temática inerente ao projeto de circuitos integrados contemplando as situações em que, por restrições sobremaneira térmicas ou energéticas, nem todas as seções do sistema podem estar energizadas e ativas simultaneamente (MODARRESSI; SARBAZI-AZAD, 2018).

tuem, crescentemente, um forte habilitador a uma resposta a esta demanda, à disposição da indústria.

O escopo deste trabalho é explorar aspectos de interesse no uso de sistemas embarcados, multiprocessados e municiados de uma rede intra-chip como subsistema de suporte à interconexão dos componentes deste sistema na implementação de Redes Neurais Convolucionais visando, sobretudo, a aceleração de sua execução. Para tanto, com a ajuda de uma plataforma de simulação, realizamos a implementação de uma CNN baseada em uma arquitetura clássica, a LeNET-5 (LECUN et al., 1998), procurando explorar computação paralela à luz do fluxo de dados inerente ao modelo. Em nossas considerações finais sobre o trabalho experimental, falaremos sobre os ganhos de desempenho obtidos, positivos no concernente à aceleração da execução do modelo preditivo porém negativo em relação à sua acurácia, em parte por conta de um aspecto relacionado ao suporte de aritmética de ponto flutuante na plataforma de simulação escolhida.

No Capítulo 1, discorremos brevemente sobre Redes Neurais Artificiais, com especial ênfase às Redes Neurais Convolucionais e à arquitetura LeNET-5, que tomamos como referência. No Capítulo 2, explicamos os princípios básicos inerentes às Redes Intra-chip, onde também apresentamos a plataforma de simulação que utilizamos em nosso trabalho experimental, a MEMPHIS (RUARO et al., 2019). O Capítulo 3 é dedicado a visitar trabalhos relacionados ao tema deste trabalho e assuntos correlatos. O Capítulo 4 é destinado ao relato de nosso trabalho experimental, onde mencionamos os métodos que utilizamos na modelagem e implementação da rede e os resultados eventualmente obtidos. Finalmente, no Capítulo 5, apresentamos algumas conclusões e proposições para trabalhos futuros.

Capítulo 1

REDES NEURAIS CONVOLUCIONAIS

O surgimento das Redes Neurais Artificiais (RNA) constituiu um marco importante na fase contemporânea do campo de estudo nomeado Aprendizado de Máquina. Entretanto, há domínios onde a aplicação de RNAs convencionais, se puder ser feita, pode se mostrar difícil de implementar ou pouco eficaz. Um exemplo é o reconhecimento de elementos em uma imagem, onde todos os *pixels* do sinal de entrada precisariam ser conectados às entradas desta RNA tradicional, e variações de luminosidade e posição dos elementos de imagem poderia afetar o desempenho do modelo. Neste contexto, surgem as Redes Neurais Convolucionais, que se baseiam na detecção de padrões geométricos neste sinal de entrada usando uma abordagem iterativa que não implica a conexão simultânea de todos os valores deste sinal. Neste capítulo, apresentamos alguns conceitos básicos de RNAs e Redes Neurais Convolucionais, para depois discorrermos sobre a arquitetura que estudamos especificamente em nosso trabalho, a LeNET-5.

1.1 Conceitos básicos

As técnicas de Aprendizado de Máquina são numerosas e se prestam a objetivos variados. Há técnicas que se prestam a encontrar valores ótimos para conjuntos de funções que modelem um sistema. Outras se propõem a determinar como um certo grupo de casos pode ser agrupado, dados alguns parâmetros de referência. E existem técnicas que se prestam à regressão, inferindo um valor para uma variável objetivo com base em relações aprendidas entre esta e outras, usadas como parâmetros de entrada.

No caso das regressões, estas podem ser do tipo linear, em que os valores a serem inferidos pelo modelo são de um domínio numérico contínuo, ou logística, quando o objetivo é atribuir ao caso de entrada uma dentre um conjunto finito de classes. Regressão

logística é também chamada regressão categórica, por seu caráter de classificação de casos de entrada.

Redes Neurais constituem uma classe de algoritmos de aprendizado supervisionado com fins de regressão. Caracteriza-se, quando comparada a outros modelos de Aprendizado de Máquina, por sua alta acurácia, baixa explicabilidade e grande potencial na identificação de padrões complexos.

Técnicas de aprendizado supervisionado são aquelas cujo treinamento do modelo preditivo é feito com base em uma coleção de observações anotadas acerca de um fenômeno de interesse. Na fase de treinamento, um conjunto de casos onde a variável objetivo já está populada, seja porque se trata de algum histórico acerca do fenômeno estudado, seja porque um especialista já o teria feito, é avaliado. Nesta fase, o modelo é ajustado de maneira a conseguir a melhor performance preditiva possível, frente padrões apreendidos deste conjunto de casos de treinamento e considerando um segundo conjunto de casos anotados, separado, usado para testes. Contrapõem-se às técnicas de aprendizado não-supervisionado, onde o algoritmo de treinamento precisa inferir padrões não com base em casos previamente anotados, mas em densidades de probabilidade relacionadas a estes padrões[1].

Acurácia é uma medida objetiva que explica o quanto um modelo preditivo é capaz de acertar um resultado antevisto dado um sinal ou caso de entrada. Explicabilidade é a característica inerente a um modelo preditivo que denota quão fácil é elicitar o conjunto de operações e decisões realizadas internamente pelo modelo para a qual uma certa inferência foi feita, dada uma entrada.

O elemento básico de uma RNA é o Perceptron, uma estrutura lógica inspirada em neurônios reais, cuja estrutura básica é mostrada na figura 2. Grosso modo, um Perceptron combina linearmente um conjunto de entradas, submetendo este resultado a uma função não-linear, gerando sua saída. É chamado "Perceptron de Rosenblatt" por conta de quem o desenvolveu e implementou primeiro, Frank Rosenblatt, baseado no trabalho em Cibernética de Walter Pitts e Warren McCulloch ainda na década de 1940 (MCCULLOCH; PITTS, 1943) (SILVA; SPATTI; FLAUZINO, 2010). Os sinais x_n que compõem a entrada são multiplicados pelos pesos w_{nj} para serem posteriormente combinados segundo uma fun-

[1]Existem ainda as técnicas de aprendizado semi-supervisionado, onde apenas uma parte dos casos de treinamento é anotada, e o aprendizado por reforço, onde o aprendizado se dá por meio da atribuição de graus numéricos às inferências feitas pelo modelo, na fase de treinamento.

ção de transferência, normalmente uma soma. O resultado desta combinação, chamado "potencial de ativação" (u), é então submetido a uma função não-linear φ. A idéia da função não-linear é produzir, como saída y_j, um "impulso" a ser propagado adiante desde que haja potencial de ativação suficiente. De outra maneira, o Perceptron "disparará" a depender da combinação dos valores ponderados de entrada. Os valores de w_{nj} são os parâmetros ditos "treináveis", definidos, como o nome sugere, na fase de treinamento.

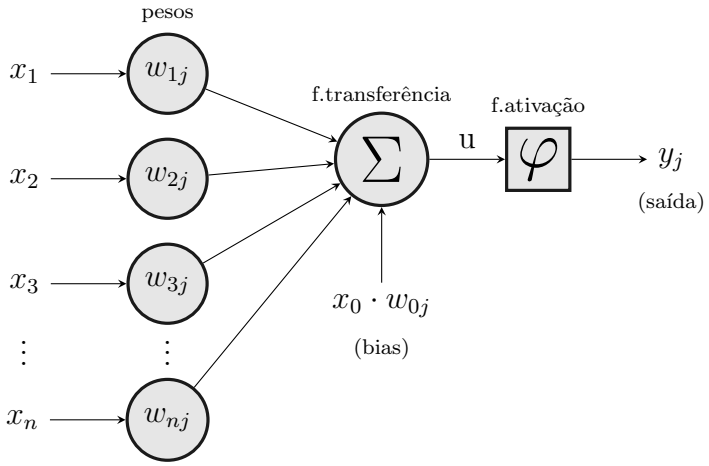


Figura 2: Um Perceptron de Rosenblatt

Há uma componente especial inserida nesta combinação linear, $x_0 \cdot w_{0j}$. Esta componente é chamada limiar de ativação. Ela se presta a ajustar o Perceptron, definindo o quanto de energia de ativação é necessário para sensibilizá-lo e produzir um "impulso" em y_j. O parâmetro treinável, ou peso, w_{0j}, recebe o nome especial de *bias*. Já a componente x_0 é frequentemente estática.

A função de ativação φ, por sua vez, pode ser customizada no desenvolvimento da RNA, havendo alguns tipos ditos "clássicos". As funções de ativação podem ser dos tipos definidas por partes ou totalmente diferenciáveis (SILVA; SPATTI; FLAUZINO, 2010). A tabela 1 mostra exemplos de funções totalmente diferenciáveis. A tabela 2 contém exemplos de funções definidas por partes.

Outros tipos de funções podem ser usadas como funções de ativação. Um exemplo é a Função de Base Radial Euclidiana (*Euclidean Radial Basis Function* - RBF), que serve ao cômputo de distâncias entre um conjunto de valores e um outro de referência, dada pela expressão 1. Seja D_{RBF} a distância entre os valores da amostra e a referência, x o valor de referência e w_j, um valor da amostra:

Tabela 1: Alguns exemplos de funções de ativação totalmente diferenciáveis

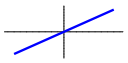
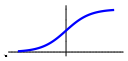
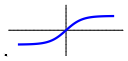
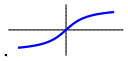
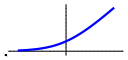
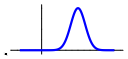
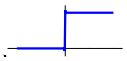
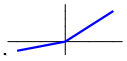
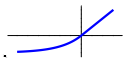
Tipo	Equação	Plotagem
Identidade	$f(u) = u$	
Logística (ou Sigmóide)	$f(u) = \frac{1}{1+e^{-u}}$	
Tangente Hiperbólica	$f(u) = tanh(u) = \frac{2}{1+e^{-2x}} - 1$	
Arcotangente	$f(u) = tan^{-1}(u)$	
Softplus	$f(u) = log_e(1+e^x)$	
Gaussiana	$f(u) = e^{\frac{-(u-c)^2}{2\sigma^2}}$	
	onde c é a média da distribuição gaussiana de probabilidade e σ, o desvio padrão.	

Tabela 2: Alguns exemplos de funções de ativação definidas por partes

Tipo	Equação	Plotagem
Degrau	$f(u) = \begin{cases} 0 \Leftarrow u < 0 \\ 1 \Leftarrow u \geq 0 \end{cases}$	
Linear Retificada (ReLu)	$f(u) = \begin{cases} 0 \Leftarrow u < 0 \\ u \Leftarrow u \geq 0 \end{cases}$	
Linear Retificada Paramétrica (PReLU)	$f(u) = \begin{cases} \alpha u \Leftarrow u < 0 \\ u \Leftarrow u \geq 0 \end{cases}$	
Exponencial Linear (ELU)	$f(u) = \begin{cases} \alpha(e^x - 1) \Leftarrow u < 0 \\ u \Leftarrow u \geq 0 \end{cases}$	

$$D_{RBF} = \sum_j (x - w_j)^2. \tag{1}$$

Uma RNA consiste em um arranjo de Perceptrons organizados em grupos ou "camadas", onde um sinal ou dado de entrada é sucessivamente processado por estas camadas até que se gere o sinal de saída. As Redes de Múltiplas Camadas (*Multi-layer Perceptron* - MLP) consistem em um tipo de organização interna das mais comuns, para RNAs. É frequentemente aplicada na construção de modelos de classificação. A figura 3 mostra o diagrama de uma MLP simples. Em uma MLP, toma-se sinais que, processados por uma

camada “de entrada”, sensibilizarão diferentes Perceptrons ali postos que, por sua vez, gerarão ou não impulsos que servirão de entrada a uma camada subsequente de perceptrons em paralelo. Essa camada é dita “escondida” porque é inerente ao modelo, e poderá haver várias delas que, sucessivamente, processarão estímulos injetados por Perceptrons da camada anterior. Finalmente, sinais de saída são gerados por uma última camada. Todo o “conhecimento” da rede está implicitamente definido pelos pesos sinápticos a ponderar os sinais de entrada de cada Perceptron e esta saída é consequência direta do estímulo inicial – o sinal de entrada – e tudo o que ele acarretou ao longo da rede. Os Perceptrons nomeados como $p0n$ constituem a camada de entrada, o Perceptron $pOut$ produziria a saída desta rede e os demais compõem as camadas escondidas.

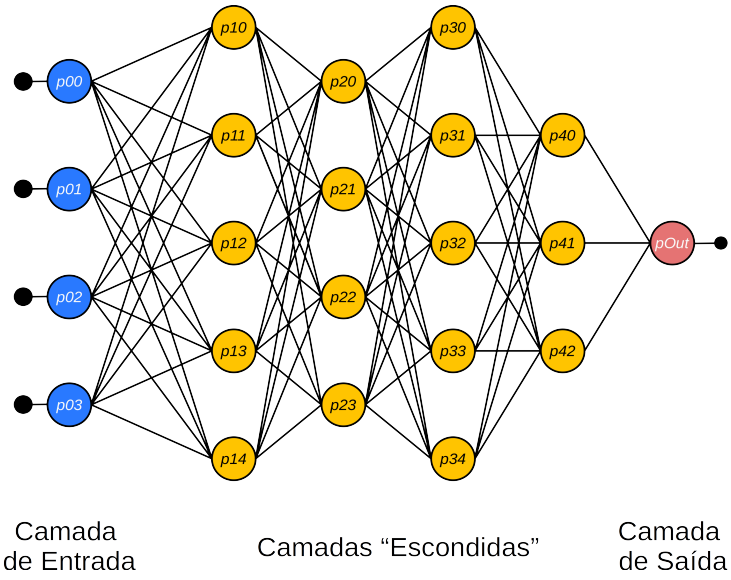


Figura 3: Uma MLP hipotética

Redes Neurais também se caracterizam pela necessidade de quantidades substanciais de casos de exemplo para o seu treinamento dada a grande quantidade de parâmetros treináveis a definir. O processamento de treinamento é iterativo e baseia-se em retropropagação de erro, implicando ajustes progressivos nos pesos sinápticos de todos os Perceptrons da rede buscando um desempenho preditivo o melhor possível em cenários de teste que visam emular condições de produção.

Há, entretanto, casos de uso onde as Redes Neurais são aplicadas em cenários que implicam dados de entrada organizados em estruturas grandes ou especiais que, por sua vez, são sujeitos a variações que podem influenciar fortemente as inferências feitas pelo modelo. Figuram aqui domínios como os problemas em Visão Computacional por exemplo,

em que os sinais ou casos de entrada são imagens que podem estar sujeitos a muitos tipos de variação em suas características sem que isso tenha qualquer importância no caráter semântico dos elementos ali retratados. De outro modo, e à semelhança da maneira bem sucedida como um cérebro humano opera com a Visão, uma laranja retratada em uma imagem pouco deveria ter a ver com as condições de iluminação, posição do objeto ou a tonalidade exata da cor da casca no dia em que a imagem foi feita, no concernente à sua condição de ser uma fruta do tipo laranja.

De fato, ao pensarmos em dados estruturados em grade, como imagens, não é possível raciocinar sobre eles como meros vetores de dados de entrada, pois sua posição relativa é importante. Sendo imagens, o exemplo, há que a adjacência dos *pixels* importa. Além disso, a depender do tamanho desta imagem de entrada, o número de parâmetros treináveis, ou "pesos" associados a esta rede neural, tende a ser bem alto, implicando a necessidade de haver uma quantidade muito grande de casos – imagens – para treinamento do modelo, e muita capacidade computacional alocada ao seu treinamento (RUSSELL; NORVIG, 2010). Neste contexto, vêm a ajudar as chamadas Redes Neurais Convolucionais.

1.2 Redes Neurais Convolucionais

Redes Neurais Convolucionais (*Convolutional Neural Network* - CNN) são um tipo especializado de rede neural voltada ao processamento de dados que têm uma organização natural em grade, para fins de classificação. Imagens seriam um exemplo típico, porque podem ser compreendidas como conjuntos de dados onde seus elementos seriam os valores de luminância associados a uma dada posição na estrutura – um *pixel*, grosso modo. Essas redes são ditas convolucionais porque, em algumas de suas camadas, o dado ali recebido como entrada será processado mediante a aplicação de um procedimento algébrico inspirado na convolução. A rigor, a operação efetivamente aplicada é o produto interno entre os valores de uma dada região do dado de entrada e uma matriz de parâmetros denominada *kernel*. Essa operação será a utilizada em lugar das multiplicações de matrizes usadas em redes neurais convencionais (GOODFELLOW; BENGIO; COURVILLE, 2016).

Convoluções são operações lineares feitas sobre duas funções de uma variável independente definida em $\mathbb{R}$ pela expressão 2. A convolução é um tipo de operação linear cujo resultado é uma integral decorrente da aplicação de uma função sobre uma outra,

combinando-as. Seja $s(t)$ a convolução das funções $x(t)$ e $w(i)$. A função x é a entrada, enquanto a função w é o *kernel* e s, o mapa de atributos resultante:

$$s(t) = (x * w)(t). \tag{2}$$

Essas operações são realizadas iterativamente, usando pequenos pedaços do dado de entrada por vez. Uma matriz de dimensões consideravelmente menores que as do dado de entrada será "movida" por sobre o dado de entrada, em um movimento de varredura. Esta matriz conteria o equivalente aos pesos sinápticos de um Perceptron comum. A ideia é que um "recorte" do dado de entrada, de dimensões compatíveis com a desta matriz – o chamado *kernel* – seja usado em operações de produto interno, de forma similar ao que acontece em uma RNA típica. De outra maneira, o *kernel* é uma janela que deslizará ao longo do dado de entrada ou resultados contribuídos por uma camada anterior do modelo, conectando as entradas de um Perceptron a diferentes pontos dessa estrutura ao longo do tempo. A saída deste Perceptron é coletada a cada iteração e compõe um elemento em uma nova estrutura bidimensional de dados, chamado mapa de atributos. Os mapas de atributos gerados constituirão dados de entrada para a camada subsequente, no modelo. A figura 4 ilustra, funcionalmente, um *kernel* genérico de tamanho 3x3.

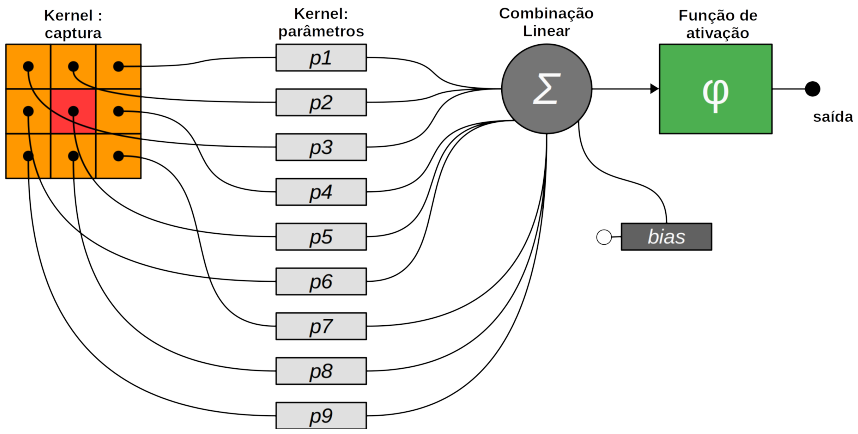


Figura 4: Um canal de detecção

Quando há mais de um *kernel* em execução em uma dada camada, a coleção de mapas de atributos produzida será o dado de entrada aos *kernels* da camada seguinte. A figura 5 mostra alguns *kernels* varrendo, simultaneamente, um dado de entrada, produzindo cada qual seu mapa de atributos.

As operações de produto interno entre uma região do dado de entrada e um *kernel* deslizante não somente flexibilizam os usos de uma CNN no concernente às dimensões da

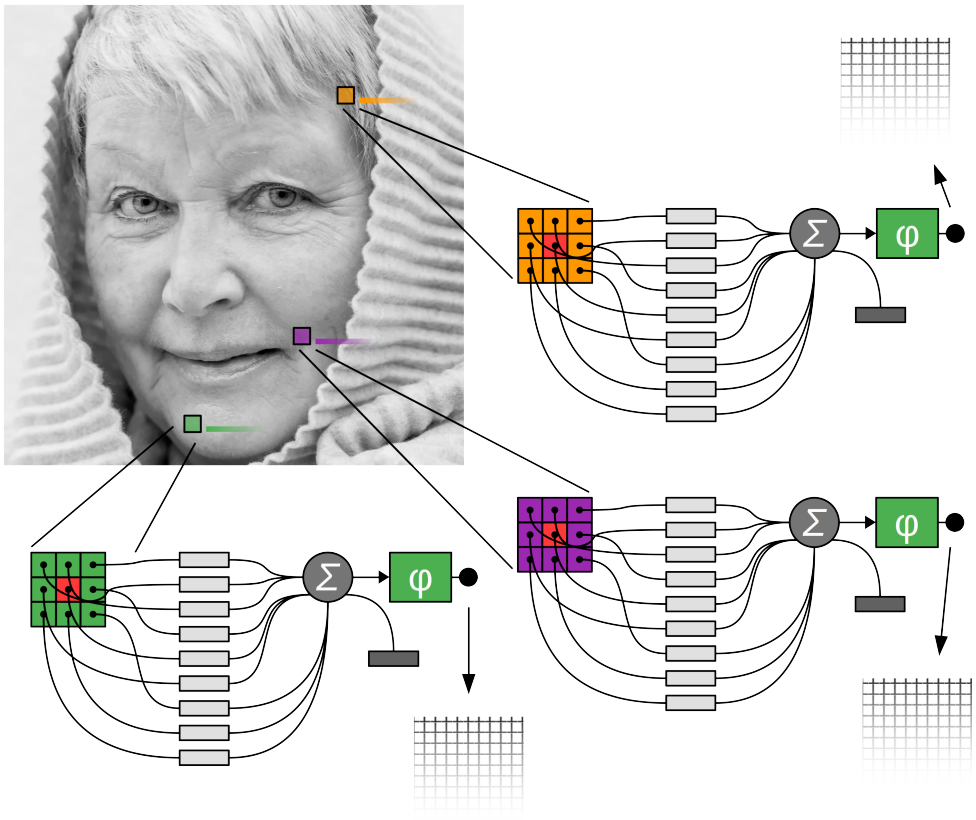


Figura 5: Um grupo de *kernels* percorrendo um dado de entrada

estrutura de dados de entrada como, também, são interessantes em relação a três outros aspectos: implicam iterações esparsas, permitem compartilhamento de parâmetros e lidam bem com representações equivariantes.

Iterações esparsas dizem respeito ao modo como o uso de um *kernel* de tamanho comparativamente menor que o dado de entrada permite redução de dimensionalidade e esforço computacional ante o necessário ao processamento do mesmo dado de entrada em uma MLP, por exemplo. Isso se dá porque trocamos, aqui, um suposto esforço de processamento de todos os *pixels* de uma imagem de entrada (que podem ser milhões) a um só tempo pela detecção de artefatos geométricos como bordas e cantos, disso fazendo os atributos efetivos a processar. Dizemos iterações esparsas porque, diferente de redes com camadas completamente conectadas, em uma camada onde se opera a convolução da entrada com um *kernel*, itera-se apenas as entradas abordadas por este *kernel*, cujas dimensões são inferiores às da entrada em algumas ordens de grandeza, tipicamente. De outra maneira, o volume de parâmetros a processar, por iteração, é limitado pelas dimensões do *kernel*. Isto traz pontos de interesse do ponto de vista computacional, por facilitar o porte dos parâmetros a processar para regiões da hierarquia de memória mais próximas

ao processador, o que normalmente implica tempos de acesso menores e, por conseguinte, melhor desempenho.

Compartilhamento de parâmetros implica o modo como os parâmetros aprendidos e associados às posições dos *kernels* são relativamente poucos em número e, por sua vez, tendem a ser aplicados sobre todos os dados de entrada, contrastando com redes neurais convencionais onde cada parâmetro aprendido é usado uma única vez e é necessariamente associado a um elemento do dado de entrada. Em uma imagem, um peso sináptico por pixel, por exemplo. Assim, por *compartilhamento de parâmetros* nos referimos ao fato de que os parâmetros usados nas transformações lineares feitas pelo *kernel* – os pesos – não serão usados apenas para uma entrada particular, mas por todo o dado de entrada. Novamente, oportunidades, do ponto de vista computacional também surgem aqui: o compartilhamento de parâmetros proporciona o uso de estratégias de *cache* que viabilizam melhor desempenho, dado que um conjunto reduzido de parâmetros será recorrentemente aplicado às entradas.

Finalmente, equivariância diz respeito ao modo como a saída de um *kernel* varia de forma solícita à entrada; considerando ser uma imagem o dado de entrada, isso faz com que a detecção de atributos seja, em princípio, menos sensível a translações destes dados na estrutura de entrada, posto que o *kernel* visitará a região onde o atributo está presente em algum momento de sua execução do que se estivéssemos usando um modelo totalmente conectado (GOODFELLOW; BENGIO; COURVILLE, 2016). Isso é crucial não somente no concernente à busca de um melhor resultado a partir do enriquecimento dos dados de entrada da CNN, durante a fase de treinamento, permitindo a síntese de novos casos de entrada mediante a submissão de outros pré-existentes a transformações lineares (translações, rotações e redimensionamentos, por exemplo). Representações equivariantes permitem que a CNN seja capaz de identificar atributos, no dado de entrada, de forma independente de sua posição específica, implicando relativa invariância espacial no processo. Assim, em uma imagem, um objeto como uma folha de grama poderá ser identificado sem importar tanto se é visto no terço superior ou inferior desta (RUSSELL; NORVIG, 2010).

Uma observação importante sobre a operação das CNNs é a de que, embora seu nome sugira que operações algébricas de convolução são realizadas em algumas de suas camadas, a operação efetivamente feita entre os parâmetros do *kernel* e uma vizinhança do mapa de atributos visitada é o produto interno. Produtos internos cumprem a função das

convoluçõe,s embora não apresentem as mesmas propriedades destas, como a comutatividade (o que não invalida esta abordagem porque esta e outras propriedades da convolução não são exploradas nas CNNs). Adicionalmente, é comum que outras funções sejam usadas em adição à operação de produto interno (GOODFELLOW; BENGIO; COURVILLE, 2016). As operações de produto interno combinam linearmente os valores de entrada, ponderadas segundo um conjunto de parâmetros e gerando um valor único, de saída. A figura 6 mostra um exemplo de produto interno.

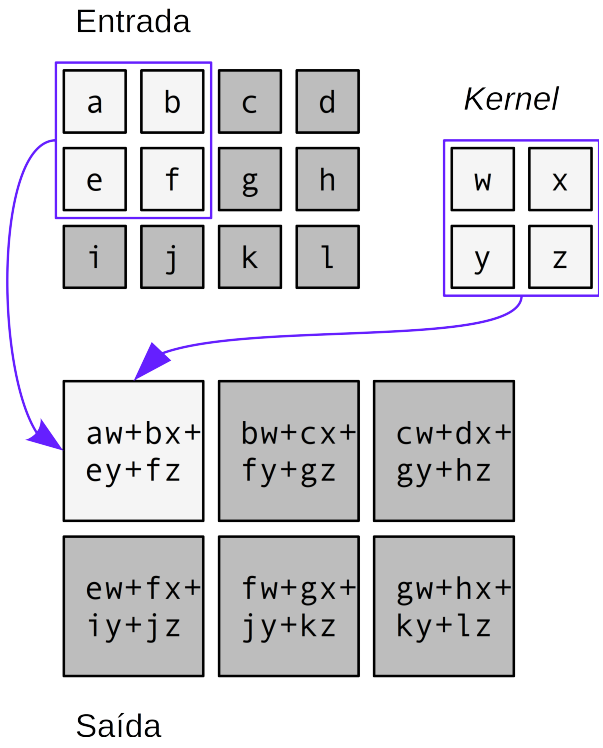


Figura 6: Um exemplo de produto interno

Para melhor ilustrar a conceituação associada ao *kernel* deslizante, pode-se fazer um paralelo entre este e os olhos de um leitor em um idioma pictográfico a varrer um texto buscando reconhecer ideogramas e imediatamente associá-los às idéias que representam. Os *kernels* varrem o dado de entrada e "sensibilizam-se" ao encontrar elementos aprendidos na fase de treinamento. Alguns termos, concernentes a esta dinâmica, são importantes. Por unidades, em um *kernel*, entendemos o valor em posição específica, dado que é uma matriz, que será combinado a um valor no dado "escaneado" de entrada, como os bastonetes nos olhos de nosso leitor fictício. É em "unidades" que as dimensões de *kernels*, nas camadas convolucionais, ou vizinhanças, em camadas de subamostragem, são dados. Há também o *padding*, que denota uma "moldura" à volta dos dados de entrada ou mapas de atributos em exame pelo *kernel* que permite que sua varredura possa tirar maior proveito

de todo o espaço denotado pela estrutura do dado em exame, com o consequente aumento das dimensões dos mapas de atributos derivados e provável enriquecimento, prevenindo também que atributos detectáveis próximo às "bordas" do dado de entrada se percam. Desta maneira, a forma específica de implementar o *padding* é relacionada às decisões de projeto inerentes ao funcionamento da CNN em implementação, sendo frequentemente criadas com molduras de valores que não intereferirão no funcionamento do modelo[2]. Igualmente importante é o *stride*, que denota o "passo" ou "deslocamento" associado ao processo de varredura, em se tratando dos *kernels* nas camadas convolucionais, ou das vizinhanças se nas camadas de subamostragem, definindo quanto do dado em exame será usado no cômputo de um elemento no mapa de atributos, em uma iteração de um *kernel* ou de subamostragem, explicada adiante.

Após as operações de produto interno, os valores gerados são submetidos a funções não-lineares, genericamente chamadas funções de ativação. Tal como nos Perceptrons, estas funções se prestam à adição de não-linearidade ao processo de identificação de atributos no dado de entrada iterado. A esta conjunção entre a aplicação do *kernel*, de maneira a realizar produtos internos, e a submissão deste resultado a uma função não-linear de ativação chamamos, genericamente, de fase de detecção.

Segue-se, à fase de detecção, uma outra etapa visando resumir seus resultados segundo uma determinada função ou procedimento de sumarização estatística envolvendo os valores sob uma dada vizinhança. A esta etapa chamamos subamostragem ou *pooling* – que, junto às outras que implementam a fase de detecção, completam o elenco de funcionalidades necessárias à implementação de uma camada convolucional. Um exemplo de procedimento de subamostragem seria a operação de MaxPooling, a preleição dos maiores valores em uma vizinhança iterada, conforme ilustrado na figura 7. A operação MaxPooling retorna o maior valor na vizinhança sob escopo da iteração.

A operação de subamostragem, ou *pooling*, consiste na aplicação de um cômputo de resumo estatístico sobre regiões do mapa de atributos gerado pelo *kernel*. Visitando regiões do mapa de atributos gerado pela fase de detecção, de forma muito semelhante ao que é feito na fase convolucional, a operação de *pooling* gera um novo mapa de atributos que constituirá a entrada de uma próxima camada, no modelo. A operação de *pooling* pode ser de tipos diversos, sendo muito comum resumos do tipo média aritmética simples

[2]Muitas vezes, valores zero, neutralizando termos do produto interno entre os dados nas molduras e as unidades no *kernel*

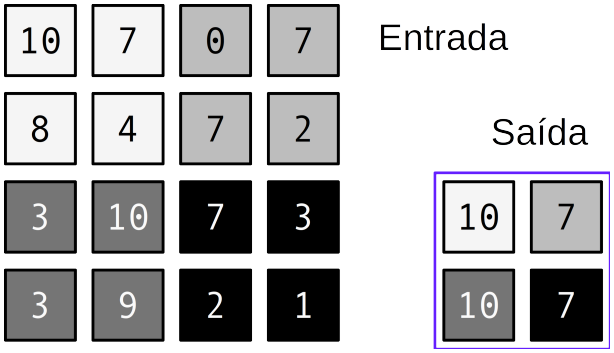


Figura 7: Um exemplo de MaxPooling

e MaxPooling. A fase de *pooling* tem, como objetivo, tornar a coleta de atributos menos suscetível a pequenas translações nos dados de entrada. Segmentar os mapas de atributos gerados pela fase de detecção e gerar resumos estatísticos produzindo, por sua vez, novos mapas de atributos tem duas vantagens imediatas: permite obter atributos mais calcados na presença de elementos de interesse na imagem e menos em sua posição exata na estrutura de dados de entrada sem, contudo, deixar de conservar alguma informação sobre sua posição relativa, nesta estrutura. Seja esta a imagem de um rosto humano: a depender da aplicação, provavelmente será mais útil saber que há dois olhos na imagem e que eles estão em posição relativa tal que implica haver uma face ali, do que identificar a posição exata do elemento, na imagem (RUSSELL; NORVIG, 2010).

As dimensões dos mapas gerados pelos canais que efetuam a detecção de atributos em uma camada convolucional e pelas operações de *pooling* podem ser calculados usando a expressão 3. Considerando-os bidimensionais e de mesma medida nas duas dimensões (quadrados), seja D a dimensão da aresta do mapa produzido; L a dimensão da aresta da estrutura dos dados de entrada (suposta quadrada); K a aresta de uma seção reta do *kernel* (considerando que, ainda que um prisma reto, sua seção reta paralela a um mapa de atributos ou dado de entrada é quadrada); P o valor de *padding* (supondo ser uniforme em todas as direções do dado ou mapa de entrada); e S, o valor de *stride* (supondo ser igualmente uniforme em quaisquer direções do dado ou mapa de entrada):

$$D = \frac{L - K + 2(P)}{S} + 1. \tag{3}$$

Usa-se, como fases finais de uma CNN, redes MLP. Assim, as camadas convolucionais solucionam a detecção de atributos no dado de entrada e a MLP subsequente se presta a operá-los com fins de classificação.

Neste trabalho, cujo escopo é a aceleração do processamento de um modelo de Aprendizado de Máquina, as CNNs se mostram particularmente interessantes enquanto objetos de estudo pela forma como operam. São calcadas em processos iterativos intensos onde, por outro lado, há grande reuso de parâmetros, operações repetitivas e, em um exame mais próximo, pouca dependência de dados entre alguns de seus elementos funcionais – *kernels* deslizantes, em uma mesma camada convolucional, podem operar de forma independente, por exemplo. Isso sugere haver extensas oportunidades para a exploração de processamento paralelo na execução de uma CNN.

1.3 A arquitetura LeNET-5

A arquitetura de CNN LeNET-5 foi proposta, inicialmente, visando resolver problemas de reconhecimento de escrita à mão – dígitos de 0 a 9, primariamente. Sua estrutura é mostrada na figura 8 (LECUN et al., 1998). A entrada é uma imagem de dimensões 32x32 *pixels*, em escala de cinza – informação de luminância somente, portanto. A arquitetura, tal como originalmente descrita, compreende sete camadas, duas delas implementando *kernels* deslizantes e seguidas, cada uma destas, por uma camada de subamostragem, ou "*pooling*".

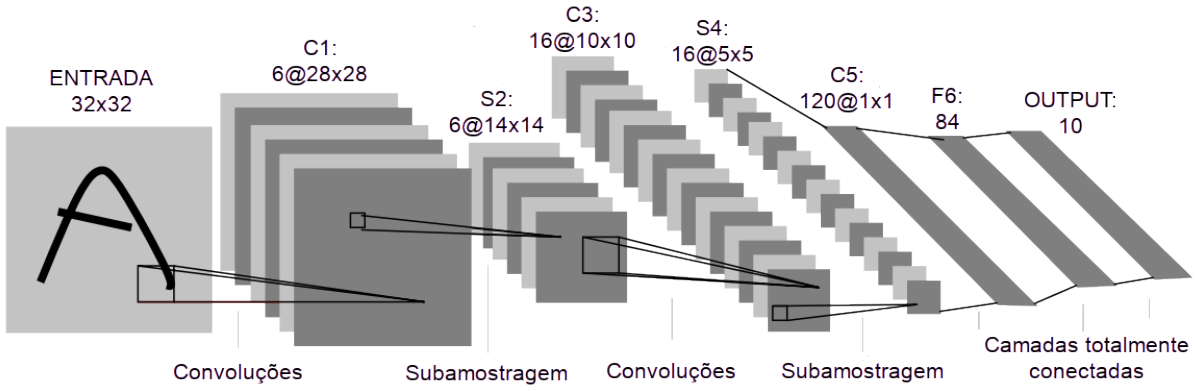


Figura 8: A arquitetura LeNET-5

C1 é uma camada convolucional com seis canais de detecção de atributos, cada qual com seu *kernel* e ativações atinentes. São *kernels* de dimensões 5x5 unidades, operando sob *stride* de 1 e *padding* 0, produzindo mapas de atributos de dimensões 28x28 unidades. Os parâmetros treináveis são, portanto, 25 por *kernel*, mais um de *bias*, perfazendo 156 ao todo.

S2 é uma camada de subamostragem que operará os seis mapas de atributos produzidos por C1, sumarizando-os em regiões 2x2, *padding* igual a 0 e *stride* 2, fazendo com que estas regiões, ou "vizinhanças", sejam disjuntas, sem sobreposição. Esses quatro valores tomados dos mapas de atributos são somados, multiplicados por um parâmetro treinável e, em seguida, somados a um valor de *bias* (implicando dizer que haveria, em S2, dois parâmetros treináveis por mapa de atributos processado, perfazendo 12 parâmetros treináveis), sendo o resultado, finalmente, submetido a uma função de ativação Sigmóide. Os seis mapas de atributos produzidos por S2 terão, dada a vizinhança e o *stride*, dimensões 14x14 unidades.

C3 é uma camada convolucional com 16 canais de detecção (0 a 15) de atributos cujos *kernels* têm dimensão de 5x5 unidades, operando com *padding* 0 e *stride* 1, produzindo 16 mapas de atributos de dimensões 10x10 unidades. A proposição original da arquitetura (LECUN et al., 1998) sugere que a conexão dos *kernels* aos mapas de atributos entregues por S2 seja deliberadamente incompleta, visando reduzir o número de conexões e tentar induzir a captura de atributos eventualmente complementares por C3, submetendo seus canais de detecção de atributos a uma assimetria deliberada. A distribuição de conexões de mapas de S2 (0 a 5) a C3 é proposta como na figura 3. C3, sob esta estrutura, tem 1516 parâmetros treináveis, contra 2496 se totalmente conectada aos mapas de S2.

Tabela 3: Mapeamento proposto de conexões entre S2 and C3

S2: → C3:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X				X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

S4 é uma camada de subamostragem que operará os dezesseis mapas de atributos gerados por C3, sob vizinhanças disjuntas de 2x2 unidades, com *stride* 2 e *padding* 0. É funcionalmente semelhante a S2, portanto, sendo a função de ativação proposta, como em S2, a Sigmóide. Gerará 16 mapas de atributos de dimensões 5x5 unidades. Havendo dois parâmetros treináveis por mapa de atributos de entrada, tal como em S2, haverá, aqui, 32 parâmetros treináveis.

C5 é uma camada convolucional com 120 canais de detecção de atributos com *kernels* de dimensões 5x5 unidades, operando os 16 mapas de atributos supridos por S4.

Uma vez que os *kernels* em C5 têm a mesma dimensão dos mapas de atributos que recebe de S4, temos que esse *kernel* não é deslizante, que o mapa consequente tem dimensões 1x1 unidade e que C5 seria, por assim dizer, "totalmente conectada a S4", ainda que tipificada como uma camada convolucional. Sobre esse último aspecto, a justificativa é a de que este caráter aparente de C5, uma "camada convolucional que se porta como um Perceptron" fazendo-a funcionalmente similar a uma camada de uma MLP cuja entrada seriam os $16 \times 5 \times 5 = 400$ atributos entregues por S4, não é mais válido se fizermos os mapas de S4 maiores em consequência de incrementos no tamanho dos dados de entrada apresentados a C1, quando os próprios mapas de atributos apresentados a C5 teriam dimensões maiores do que as de seus *kernels* (LECUN et al., 1998). De todo modo, C5 entrega 120 mapas de atributos de dimensões 1x1 unidades à camada seguinte no fluxo de dados, F6, e o número de parâmetros treináveis é $120 \times (5 \times 5 + 1) = 48.120$. A soma de 1 aos parâmetros associados ao *kernel* refere-se ao *bias*.

F6 é uma camada totalmente conectada a C5, havendo 84 Perceptrons conectados aos 120 atributos (em verdade, mapas de atributos de dimensões 1x1) recebidos desta última. Aqui, a função de ativação proposta originalmente pelo autor é a RBF. A saída da camada consiste em um conjunto de 84 atributos. O número de parâmetros treináveis será, então, $120 \times 84 + 84 = 10.164$, sendo o acréscimo de 84 parâmetros a este cálculo, referente aos *bias* dos 84 Perceptrons nesta camada. A função de ativação RBF é usada para realizar um cômputo de distância média entre os valores recebidos por C5 e versões codificadas dos caracteres de 0 a 9. Quanto menores as distâncias entre os atributos gerados por C5 e aqueles atinentes aos caracteres codificados, mais provável que o sinal de entrada corresponda àquele caractere.

Finalmente, OUTPUT é uma camada totalmente conectada aos 84 atributos gerados por F6, sendo composta por dez Perceptrons cujas funções de ativação propostas seriam a Tangente Hiperbólico. Estes dez Perceptrons gerarão as saídas que expressam a inferência feita pela CNN em relação ao dígito apresentado na imagem de entrada. Portanto, as saídas destes Perceptrons estão associadas à probabilidade inferida pela CNN de que a imagem de entrada refira-se ao dígito ou à classe especificamente associada àquele Perceptron. Em termos de número de parâmetros treináveis, teríamos aqui $84 \times 10 + 10 = 850$.

A arquitetura descrita, como explicávamos, é a originalmente proposta pelo autor. Entretanto, visto que é uma arquitetura de referência, pode se mostrar diferente, a depender de especificidades na forma como é implementada: funções de ativação, tamanhos de estruturas de dados de entrada e mapas de atributos podem mudar, a depender das características do projeto.

1.4 Considerações Finais

As Redes Neurais Convolucionais têm méritos que derivam vantagens inúmeras, habilitando aplicações em domínios onde outras técnicas não aderem bem. A natureza espacial de seu tratamento de entradas, combinada à sua capacidade em detectar atributos a partir de padrões geométricos, além da maneira como pode lidar, de forma eficiente, com dados de entrada com organização espacial e de dimensões grandes sem uma necessária contrapartida no número de parâmetros treináveis no modelo, tornam as CNNs protagonistas de casos bem-sucedidos de uso em domínios vários, sendo hoje uma ponte importante ao campo de Aprendizado de Máquina a disciplinas como Processamento de Sinais e Visão Computacional, justificando o interesse pela técnica.

Capítulo 2

REDES INTRA-CHIP

AS redes intra-chip são sistemas de interconexão de módulos em circuitos integrados, de popularidade crescente desde meados dos anos 2000, que implementam metáforas em comunicação de dados semelhantes às já conhecidas em redes de computadores. São um importante habilitador no projeto de sistemas embarcados multiprocessados modernos. Neste capítulo, discorremos brevemente sobre sua conceituação básica, topologias típicas e algoritmos de roteamento. Terminamos o capítulo apresentando a MEMPHIS, a plataforma de simulação usada em nosso trabalho experimental.

2.1 Conceitos básicos

Em meados dos anos 2000, a chamada Lei de Moore encontrou seu limiar de saturação por problemas impostos pelos limites físicos dos materiais. Aumentar o número de transistores e as frequências de *clock* aplicados aos sistemas integrados em um *chip* não configuravam mais solução suficiente para incrementar o desempenho do sistema em si. O Escalonamento de Dennard (DENNARD et al., 1974) não mais se confirmava: postulava que a potência dissipada pelos transistores componentes do sistema diminui conforme sua área é reduzida, o que sugeria que as frequências de *clock* poderiam crescer sem uma contrapartida térmica importante desde que a tecnologia garantisse a redução do tamanho dos componentes. O que houve é que barreiras materiais foram atingidas quando as frequências de *clock* alcançaram valores na faixa de 3 a 4GHz.

Considerações sobre o consumo energético e dissipação de potência também crescem em relevância: o incremento da densidade de transistores por unidade de área nos *chips*, objeto da Lei de Moore, implicou à industria um caminho onde o mero incremento das frequências de *clock* não configurava mais solução para acréscimos de desempenho. De

fato, já quando as dimensões dos transistores figuravam na ordem de dezenas de nanômetros e as frequências de *clock*, na ordem de alguns gigahertz, as operações de chaveamento desempenhadas por estes componentes eletrônicos facilmente se encontravam à mercê de mudanças no comportamento físico dos materiais semicondutores que os compõem. Fatores como a inércia associada ao trânsito de portadores de carga nestes materiais, a reatância das junções PN, indução mútua, considerações ligadas ao regime de *clock* e o calor gerado por efeito Joule mostraram-se mais importantes porque suas consequências passaram a ser experimentadas mais intensamente, dadas as modificações que infligem ao caráter semicondutor destes materiais. Deste modo, duas limitações físicas estavam postas: um teto às frequências de *clock* e a necessidade de equipar os circuitos integrados de sistemas de resfriamento capazes de dissipar o calor gerado por eles da forma mais veloz possível, sob pena de danificarem-se ou apenas sofrer mudanças em seu comportamento elétrico, com consequente mau funcionamento lógico (ILATIKHAMENEH et al., 2016).

Nesse momento em que a capacidade computacional inerente aos *chips* não pode mais ser acrescida por mero aumento das frequências de *clock* e, também, as tecnologias de integração permitiam agregar um grande conjunto de módulos e funcionalidades aos componentes, popularizam-se os sistemas integrando diversos Elementos Processadores (*Processing Elements* - PE) em um único *chip*. Inaugura-se um capítulo na história dos sistemas microprocessados em que a computação paralela passa a ser mais acessível e frequente em dispositivos ditos "de consumo". A figura 9 mostra a evolução dos microprocessadores comerciais frente a evolução das frequências de *clock* e da dissipação de potência. É importante notar a inflexão das curvas de potência dissipada e frequência de clock em meados da década de 2000 (RUPP et al., 2015).

A interligação dos PEs, por outro lado, passa a ser objeto de crescente interesse posto que, à medida em que o número de unidades disponíveis no sistema aumenta, sistemas de interconexão clássicos, como barramentos, se tornam uma limitação. Ocorre que, em esquemas de interligação por barramentos, a disputa pelo meio de transmissão é um fator limitante e importante no concernente ao dispêndio de tempo, impondo grande *overhead* por conta de transações de comunicação. Esquemas de interligação visando diminuir a disputa pelo meio, como barramentos hierarquizados, vêm a ajudar mas não resolvem por completo a questão. Desta forma, esquemas de comutação de circuitos, como matrizes de barramentos, começam a surgir.

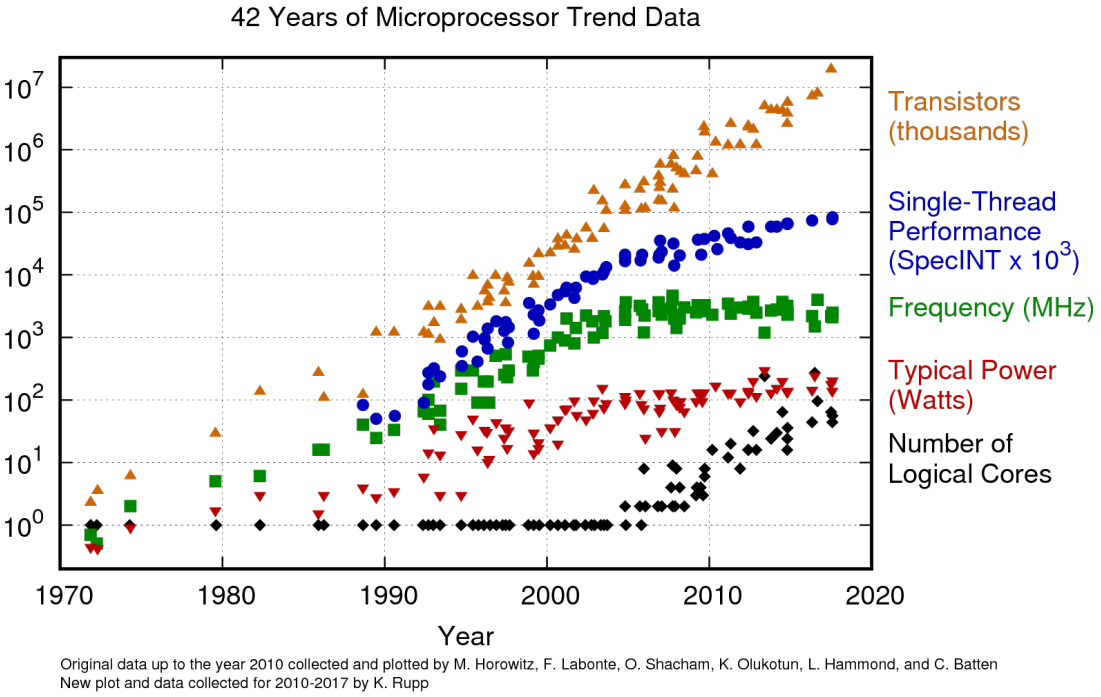


Figura 9: A evolução dos microprocessadores, em cinco décadas

Pouco depois, no entanto, começam a ser propostas, em trabalhos como (BENINI; MICHELI, 2002), as chamadas Redes Intra-Chip (*Network-on-Chip* - NoC), abordagem que implementa, no âmbito de um sistema embarcado (*System-on-Chip* - SoC), uma rede de comutação de pacotes, à semelhança das já maduras redes de comunicação de dados usadas na interconexão de computadores calcadas no protocolo TCP (*Transfer Control Protocol*). A figura 10 ilustra a evolução das tendências em sistemas de interconexão *intra-chip*, na manufatura de sistemas embarcados (PASRICHA; DUTT, 2008).

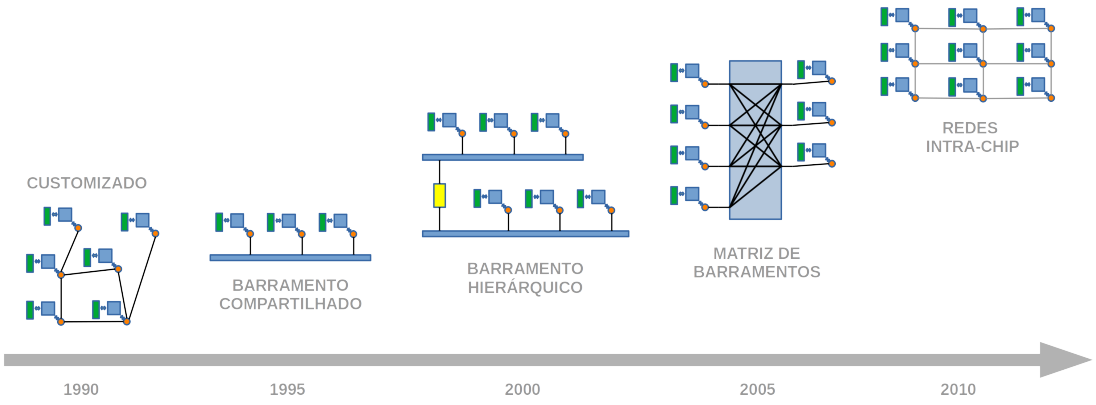


Figura 10: A evolução dos sistemas de interconexão intra-chip

É oportuno definirmos, aqui, alguns termos. Chamamos genericamente de recursos os dispositivos constituintes do SoC eventualmente dispostos nos nós da NoC, dos quais uma aplicação pode lançar mão. Por aplicação, entendemos o *software* que, utilizando-

se dos recursos, implementa ou provisiona algum tipo de solução ou funcionalidade ao usuário. Aplicações, por sua vez, decompõem-se em tarefas, blocos de *software* que implementam partes integrantes da aplicação e que constituirão, em si, as cargas de trabalho atribuídas aos PEs no SoC. As tarefas em si implementam blocos funcionais básicos necessários ao trabalho computacional realizado pela aplicação, como operações de manipulação e comunicação de dados, ou operações matemáticas.

As redes intra-chip representam não somente oportunidade na obtenção de um desempenho em comunicação viável em sistemas com grande quantidade de PEs mas, também, uma forma de simplificar o processo de desenvolvimento de um sistema. De fato, as redes intra-chip isentam parcialmente o projetista da necessidade de antecipar tipos e regimes de comunicação inerentes às aplicações a serem executadas no sistema posto que, graças à NoC, recursos como PEs e periféricos podem ser conectados com menos imposições ou considerações a fazer em tempo de *design* frente a outras abordagens ditas "tradicionais". Em um projeto calcado em barramentos, por exemplo, há que o número e a disposição, no *chip*, de recursos no sistema é fortemente influenciado por questões incidindo sobre o meio de comunicação, como banda passante. Outro exemplo são as aplicações sujeitas a forte débito de desempenho por conta da quantidade de transações de comunicação que naturalmente demandam, e problemas advindos daí como a disputa, pelos módulos do sistema, no uso do acesso ao meio. É necessário considerar também que, em sistemas embarcados, operações de comunicação são frequentemente bloqueantes, ou seja, impõem interrupção e espera por parte de um ou mais recursos envolvidos na transação de comunicação. O uso de uma rede intra-chip pode ser benéfico em todos estes casos, posto que a disputa pelo meio de comunicação é gerida no âmbito da própria rede e mesmo a buferização e encaminhamento de mensagens, administrada por seus componentes.

Em uma NoC, o nó de interconexão é o principal ator no concernente à implementação de funcionalidades de comunicação. Sua função é implementar o roteamento de pacotes de dados em tramitação na rede configurando-se, desta forma, como uma chave de interconexão. Como elementos que o compõem, temos as portas de comunicação e os respectivos *buffers* associados, conectados a uma matriz que, por sua vez, é controlada pela lógica de roteamento, conjugada a um elemento árbitro. Este elemento árbitro decide sobre a priorização na liberação das vias de interconexão quando vários pacotes, advindos

de diferentes fontes, necessitam utilizá-las. Já a matriz, é frequentemente do tipo *crossbar*. Sua função primária é realizar as conexões momentâneas entre as portas, permitindo o fluxo de dados entre elas segundo o encaminhamento que determinado pacote deve tomar, dada a sua rota para daquela chave em diante (PASRICHA; DUTT, 2008).

O nó de interconexão implementado pela plataforma de simulação que utilizamos é baseado na HERMES, uma chave para a comutação de pacotes (MORAES et al., 2004). Sua estrutura básica, mostrada na figura 11, consiste em uma lógica de controle e chaveamento e um conjunto de portas de comunicação. As portas de comunicação, bidirecionais e identificadas como Norte, Oeste, Sul e Leste, se prestam à interconexão de uma chave com outras, "vizinhas". Já a porta local é aquela à qual se conectará o recurso alocado ao nó, podendo ser um Elemento Processador ou um periférico, como um controlador de memória ou interface de comunicação. Os *buffers* ("BUF") associados a estas portas fazem a salvaguarda temporária de pacotes de dados que aguardam transmissão ou processamento, se recebidos.

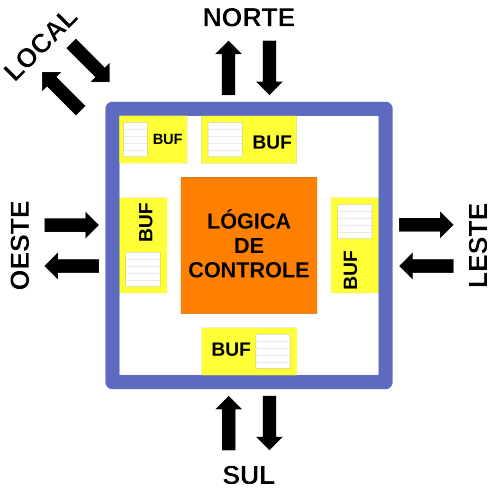


Figura 11: Diagrama básico de uma chave de interconexão HERMES

Cabe também definir alguns termos inerentes à organização dos dados no concernente à comunicação, conforme ilustrado na figura 12.

As mensagens são os corpos de dados efetivamente trocados entre as tarefas em execução nos recursos presentes no sistema. As mensagens são posteriormente quebradas em pacotes, que são blocos de dados organizados de maneira a serem transmitidos por um sistema de comunicação. Os pacotes, por sua vez, têm duas grandes seções: uma reservada ao dado que é efetivamente transmitido pela tarefa, chamada seção de carga útil ou *payload*, e uma outra com dados de controle, chamada cabeçalho. O cabeçalho

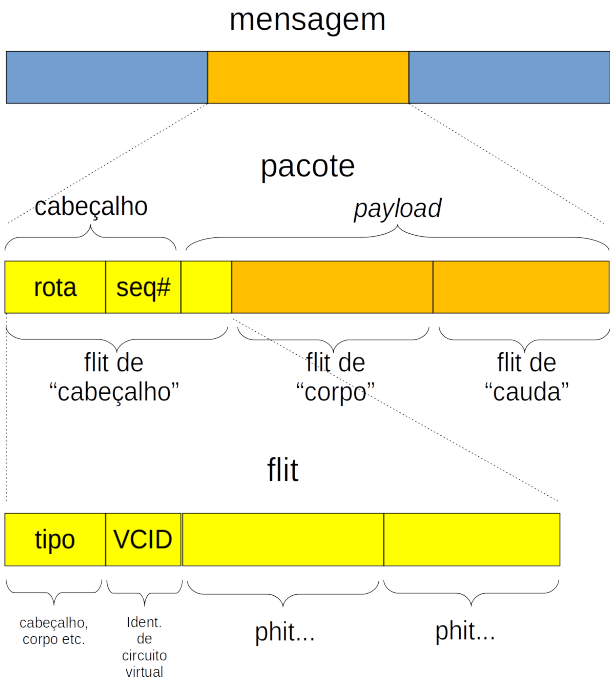


Figura 12: Mensagens, pacotes e flits

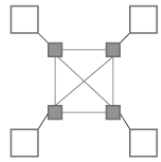
contém informações usadas na ordenação e roteamento dos pacotes de maneira a permitir a transmissão e posterior reconstrução do dado em seu destino. Um pacote, por sua vez, é dividido em *flits* (de *flow units*). Um *flit* é um bloco básico de dados sobre os quais as operações de roteamento são efetivamente implementadas, em nível de enlace. Há então os *flits* de cabeçalho, contendo informações gerais de roteamento; os *flits* de cauda, delimitando o final do pacote e, também, dados de correção prévia de erro (*Forward Error Correction* - FEC); e há os *flits* destinados ao “corpo” da mensagem, estes contendo os dados “reais” e que compõem, com os *flits* de cauda, a carga útil (*payload*) do pacote. Por sua vez, os *flits* são divididos em *phits* (de *physical units*), estes constituindo a porção de dados, em bits, encaminhada em um único ciclo físico de comunicação. O tamanho dos *phits* normalmente é definido pela largura da via de dados conectando as chaves de interconexão, na NoC (PASRICHA; DUTT, 2008).

2.2 Topologias

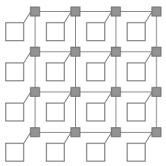
A quantidade e a configuração física das conexões da chave de interconexão com outros em sua vizinhança e ao próprio PE, realizadas por meio de conexões diretas, definem, grosso modo, a arquitetura (ou topologia) da rede. Podemos dividir as topologias de NoC em duas classes distintas, a depender se sempre haverá ou não recursos alocados nos nós

da rede. Temos as ditas redes diretas, onde cada nó da rede é, ao mesmo tempo, o lugar de um recurso como um PE, por exemplo, e o de um elemento de roteamento, a chave de interconexão. E temos as redes indiretas, onde os nós são o lugar de um recurso ou de uma chave, porém não as duas coisas simultaneamente. É importante notar que, em redes diretas, uma mensagem precisa ser veiculada por uma série de nós, na rede, onde também há recursos de computação a eles conectados. Isso é relevante na medida em que a chave de interconexão emum dado nó precisa processar não somente o roteamento de mensagens advindas de lugares diversos da rede como, ainda, as operações de comunicação do recurso a ele conectado. Em redes indiretas, haverá chaves dedicadas exclusivamente às operações de roteamento na NoC, permitindo a construção de rotas que tendem a permitir fluxos de dados maiores, podendo servir à interconexão de grupamentos de recursos, por exemplo.

A figura 13 mostra alguns exemplos de topologias de NoCs. Dentre as topologias diretas, temos: Ponto a Ponto (a), Malha 2D (b), Anel (c), Torus (d), Folded Torus (e) e Octogonal (f). Também há arquiteturas indiretas: Fat Tree (i), Borboleta (j), Clos (k) e Benes (l). E há também topologias que são ditas irregulares. Os exemplos seriam: uma malha reduzida (ou "otimizada") (g) e uma outra, dita "híbrida", baseada em grupamentos e combinando malhas e anéis (h). Topologias irregulares, tipicamente, combinam configurações diretas e indiretas, e também barramentos, às vezes.

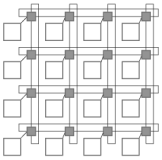


(a) Ponto a Ponto

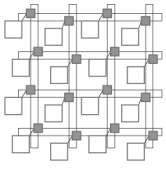


(b) Malha 2D

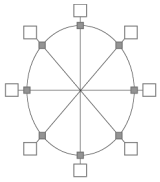
(c) Anel



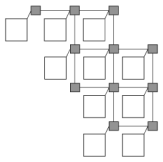
(d) Torus



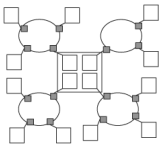
(e) Folded Torus



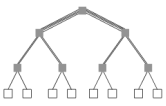
(f) Octogonal



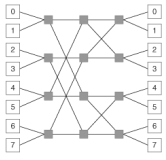
(g) Malha Reduzida



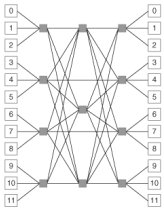
(h) Malha e Anel



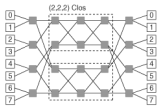
(i) Fat Tree



(j) Borboleta



(k) Clos



(l) Benes

Figura 13: Alguns exemplos de topologias de redes intra-chip

Malhas 2D são, provavelmente, o tipo de topologia de NoC do tipo direta mais popular e é a que usamos em nosso trabalho experimental. De fato, as topologias diretas em malha, notadamente ortogonais, podem ter mais de duas dimensões, embora isso imponha acréscimos importantes à complexidade do projeto, como explicado em (MILLBERG et al., 2004).

A utilização desta ou aquela topologia dependerá, sobretudo, de critérios de projeto. Topologias de malha irregulares, por exemplo, podem ser escolhas interessantes se a intenção do projetista for ter um recurso em cada nó – uma configuração direta, portanto – quando não lhe for opção dispender espaço, no *chip*, para uma NoC de malha regular.

Com o aumento do interesse pela pesquisa e desenvolvimento de arquiteturas de NoC, várias foram propostas na década de 2000, usando estas topologias como base. Uma delas é a *Nostrum*, uma arquitetura de NoC do tipo malha 2D (MILLBERG et al., 2004). A *Nostrum* prevê que as chaves de interconexão sejam dispostas em uma malha regular onde cada um destes têm vizinhança com até quatro outras chaves. Há também a Æthereal (GOOSSENS; DIELISSEN; RADULESCU, 2005), calcada em comutação de circuitos e implementada primariamente sobre uma topologia indireta e com forte ênfase em garantia de serviço; a MANGO (BJERREGAARD, 2005), baseada em circuitos assíncronos e topologias diretas; e a Spidergon (COPPOLA et al., 2004), uma NoC de topologia ponto a ponto. Há muitas outras: a própria HERMES é citada em (PASRICHA; DUTT, 2008) como exemplo da NoC calcada em malha 2D homônima da chave de interconexão com a qual é implementada – por sinal, a base do ambiente de simulação que usamos em nosso trabalho experimental.

2.3 Algoritmos de roteamento

Os algoritmos de roteamento são responsáveis pelo encaminhamento correto e eficiente de pacotes de informação ou a formação de circuitos de comunicação entre uma fonte e um destino, em uma transação de comunicação (PASRICHA; DUTT, 2008). A escolha de um algoritmo de roteamento envolve algum compromisso entre diferentes métricas, frequentemente conflitantes, como dispêndio de energia, dispêndio de área no *chip* para lógicas de controle e tabelas de roteamento, minimização de atrasos de transmissão, uso ótimo de banda passante e adaptatividade frente a mudanças nas condições da rede. Os algoritmos de roteamento também precisam evitar condições degradantes de desempenho e funciona-

lidade, como a circulação indiscriminada de pacotes que acabam por não encontrar seus destinos (*livelocks*) ou a demora ou inibição na retransmissão de certos pacotes por serem atinentes a serviços de baixa prioridade (*starvation*).

Os algoritmos de roteamento também precisam prevenir condições que impeçam o roteamento de um pacote por tempo indeterminado mediante a impossibilidade de sua buferização adiante nas chaves envolvidas na rota (*deadlock*). A figura 14 mostra quatro chaves (a, b, c, d) procurando emitir dados mas em uma situação cíclica tal que seus *buffers* não podem ser liberados porque os dados neles contidos não podem ser encaminhados, fazendo com que as chaves bloqueiem-se – *deadlock*. As linhas tracejadas indicam a movimentação pretendida – que não acontecerá – de *flits* ou pacotes nos *buffers* das chaves. É parte do papel do algoritmo de roteamento a prevenção desse tipo de situação, ou a sua resolução, dinamicamente.

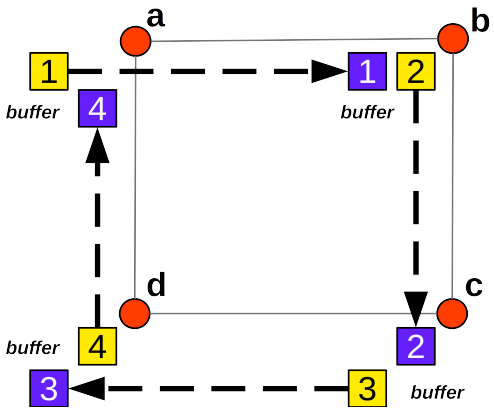


Figura 14: Chaves de interconexão em deadlock

A execução dos algoritmos de roteamento, nas chaves de interconexão, é comumente feita em lógica de controle adjunta à do árbitro (DUATO; YALAMANCHILI; NI, 2002). De fato, é nesta lógica de controle que a orquestração e gerenciamento de *buffers* e da matriz de comutação é feita e, por isso, roteamento e gestão de requisições por vezes parecem ser partes de uma mesma funcionalidade. Entretanto, e embora ambas as funções precisem operar conjuntamente, isso não quer dizer que deixam de ser distintas.

Podemos classificar os algoritmos de roteamento sob diferentes olhares. Do ponto de vista da determinação das rotas, há algoritmos ditos estáticos, em que os caminhos são fixos ou definidos *a priori*, e os dinâmicos, onde o estado instantâneo da rede é considerado e as decisões de roteamento podem mudar a depender de fatores como as condições de carga nos caminhos entre chaves de interconexão, ou mesmo sua disponibilidade. Uma

outra forma de tipificar os algoritmos de roteamento é em função do local onde as decisões de roteamento são feitas ou armazenadas, havendo aqui os algoritmos ditos distribuídos e os "predefinidos na fonte": no primeiro, os cabeçalhos dos pacotes contêm o destino e as decisões de roteamento são realizadas em cada nó, dinamicamente, enquanto que, no segundo, as rotas são previamente computadas e armazenadas em um nó especial ou "mestre", não havendo decisões sobre roteamento, para o pacote, nos nós intermediários. Um terceiro exemplo seria a tipificação dos algoritmos em relação ao "comprimento" da rota, havendo os algoritmos ditos mínimos, onde se tenta determinar a menor rota possível, e os não-mínimos, estes admitindo o uso de rotas que não seriam as de comprimento ótimo. Como dissemos, cada classe sob cada um destes parâmetros tem suas implicações no concernente ao dispêndio de recursos computacionais e de área no *chip* para cômputo e controle de rotas, consumo de energia, tempo necessário a este cômputo, *overhead* de comunicação imposto, dentre outros (PASRICHA; DUTT, 2008). Via de regra, algoritmos dinâmicos tendem a ser mais custosos, em termos energéticos e de consumo de recursos computacionais e de comunicação.

Algoritmos estáticos tendem a ser populares não somente porque são mais fáceis de implementar como, também, exigem uma quantidade menor de lógica de controle, o que também tem um impacto energético e de consumo de área associados. Além disso, com as rotas definidas *a priori*, é mais fácil fazer uso de rotas múltiplas para o encaminhamento de *flits* e, ao mesmo tempo, prevenir problemas como *deadlocks*. Nos casos em que os dados seguirão por uma única rota, também é mais fácil garantir a ordenação dos pacotes ou *flits* na chegada ao destino (PASRICHA; DUTT, 2008).

Existem muitos algoritmos de roteamento, muitos deles versões aprimoradas de outros. Seguem alguns exemplos de algoritmos de roteamento, a iniciar por alguns do tipo estático:

- Roteamento ordenado por dimensão (*Dimension Order Routing* - DOR) (DALLY; TOWLES, 2004): em topologias de malhas ortogonais (malhas 2D, cubos ou hipercubos), constrói rotas usando distâncias mínimas nas dimensões da malha, em uma ordem específica.

- XY (DEHYADGARI et al., 2005): em uma topologia de malha 2D, constrói uma rota que prevê o deslocamento em uma dimensão (X) e, depois, na dimensão ortogonal (Y). É um algoritmo simples e popular, usado como base para vários outros.

- XY Pseudoadaptativo (DEHYADGARI et al., 2005): uma variação do algoritmo XY, visando distribuir tráfego por regiões menos utilizadas da malha, evitando congestionamento.

- *Surrounding* XY (BOBDA et al., 2005): também uma variação do algoritmo XY, visa criar "contornos" quando uma chave percebe que uma outra, em sua vizinhança, está indisponível.

- *West-first* (KARINIEMI; NURMI, 2005): constrói rotas priorizando a retransmissão de pacotes usando a porta Oeste das chaves, considerando topologias em malha 2D.

- *North-last* (KARINIEMI; NURMI, 2005): constrói rotas considerando as portas Norte das chaves como as menos prioritárias, considerando topologia em malha 2D.

- *Negative-first* (KARINIEMI; NURMI, 2005): constrói rotas priorizando passos em "direções negativas", em uma malha 2D (convencionalmente, as conectadas pelas portas Oeste e Sul da chave).

- Valiant's Random (DALLY; TOWLES, 2004): constrói rotas randomizando a escolha de chaves intermediárias no caminho como forma de obter uma melhor distribuição de tráfego.

- ALOAS (KIM et al., 2005): procura fazer com que as chaves computem previamente decisões de arbítrio como forma de acelerar o encaminhamento de pacotes, visando obter melhoria em latência, na NoC.

- Inundação probabilística (PIRRETTI et al., 2004): abordagem visando obter robustez fazendo com que uma chave, em uma malha ortogonal, emita cópias dos pacotes às suas vizinhas, porém mediante sorteio a uma probabilidade arbitrária e constante.

- Inundação dirigida (PIRRETTI et al., 2004): variação do algoritmo de inundação probabilística, onde a probabilidade de que um vizinho da chave emissora seja sorteado para que se emita a ele uma cópia do pacote é função do custo estimado do caminho ao destino da mensagem, e não um valor arbitrário e fixo.

- Redundant Random Walk (PIRRETTI et al., 2004): baseado nos algoritmos de inundação probabilística e dirigida, visa diminuir o tráfego intenso criado por estes as-

sociando às portas da chave (conectadas às chaves suas vizinhas) diferentes valores de probabilidade para o sorteio para o encaminhamento de cópias do pacote.

Da mesma forma, seguem exemplos de algoritmos de roteamento dinâmico:

- *Turnaround-turnback* (ADRIAHANTENAINA et al., 2003): o algoritmo de roteamento criado para a NoC SPIN, indireta e baseada numa topologia Fat Tree onde as chaves que são antecessoras comuns das fontes e dos destinos dos pacotes têm a prerrogativa de realizar seu roteamento.

- *Turnback when possible* - TBWP (KARINIEMI; NURMI, 2005): baseado no Turnaround-turnback, o TBWP permite que níveis superiores às chaves antecessoras comuns a fontes e destinos dos pacotes sejam examinadas, caso o enlace desta chave antecessora ao ramo onde está o destino do pacote esteja ocupado.

- Q (MAJER et al., 2005): prevê comportamento adaptativo em função do estado de congestionamento das chaves vizinhas e testes de diferentes políticas de roteamento, visando determinar uma estratégia ótima no concernente à minimização da latência, na NoC.

- *Odd-even* (HU; MARCULESCU, 2004): algoritmo que procura prevenir *deadlocks* ajustando as regras para encaminhamento de pacotes a depender das coordenadas da chave, em uma malha 2D.

- *Slack-time aware* (ANDREASSON; KUMAR, 2005): algoritmo que busca otimizar o uso de banda reservada a serviços prioritários e que esteja ociosa na retransmissão de pacotes associados a serviços de prioridade mais baixa e, por isso, submetidos a políticas de "melhor esforço".

- Batata quente (FEIGE; RAGHAVAN, 1992): algoritmo que pressupõe não haver buferização em chaves intermediárias, de modo que os *flits* recebidos por uma chave devem ser imediatamente encaminhados.

2.3.1 Sobre o algoritmo XY de roteamento de pacotes

Aqui exploramos um pouco mais o algoritmo XY por ser este o implementado pela chave de interconexão HERMES. O algoritmo XY considera que as chaves estão dispostas em

topologia malha 2D, implicando dizer que uma chave pode ter até quatro vizinhas, às quais se conectará por meio das quatro portas cujos nomes convencionados seriam Norte, Oeste, Sul e Leste. Considera-se também que uma chave pode ser indicada, no SoC, por suas coordenadas em eixos que nomeamos X, paralelo a um segmento de reta imaginário ligando as portas Leste e Oeste de uma chave e com sentido positivo na direção Leste, e um eixo Y, paralelo a um outro segmento imaginário de reta ligando as portas Norte e Sul de uma chave, sentido positivo na direção Norte. A figura 15 é uma representação gráfica de uma suposta NoC com 12 chaves dispostas em uma malha 2D 4x3. Arbitramos que a origem de nosso sistema cartesiano imaginário é a posição da chave mais a sudoeste no SoC, cujo endereço grafamos como (0,0).

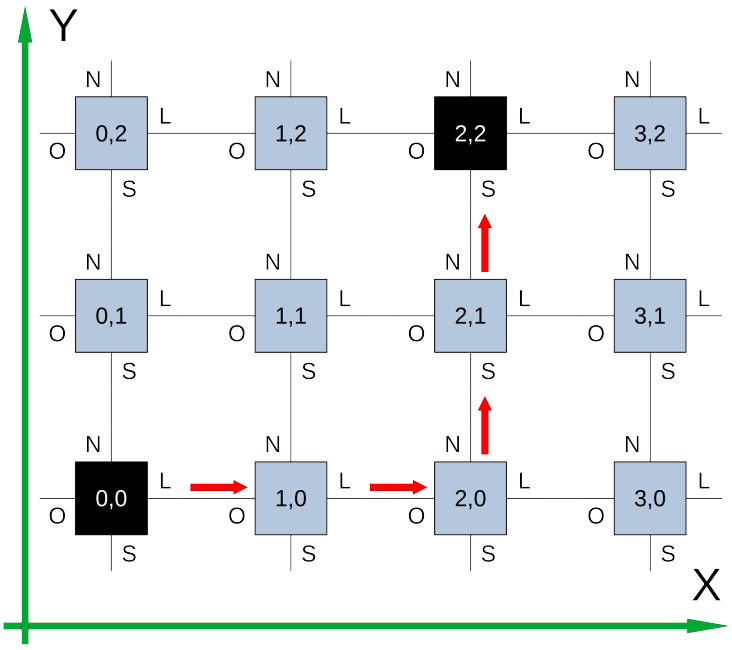


Figura 15: Representação de uma malha 2D e de um pacote sob roteamento XY

Se um recurso em (0,0) produz uma mensagem para consumo de um outro em (2,2), a rota computada *a priori* levará os pacotes inerentes à mensagem refluírem pela porta Leste da origem e serem comutados das portas Oeste a Leste da chave em (1,0) e chegando pela porta Oeste de (2,0), "fazendo uma travessia em X" até a abscissa correspondente à posição do destino. Em seguida, uma "volta" é feita e cada pacote será roteado pela porta Norte da chave em (2,0), roteado das portas Sul a Norte na chave em (2,1) e, finalmente, chegará à porta Sul de (2,2), completando a "travessia em Y" e atingindo seu destino. Uma observação importante é que estas travessias podem ser feitas nos sentidos "positivo"

ou "negativo" dos eixos, desde que observada a sua ordem: primeiro a travessia em X e, depois, em Y.

De maneira geral, o algoritmo tem como vantagem sua simplicidade, ainda que não tire proveito pleno das possibilidades de roteamento decorrentes da NoC. No algoritmo 1, seja (X_e, Y_e) a posição da chave, na malha 2D, a emitir um pacote, e (X_r, Y_r), seu destino.

Algoritmo 1 Algoritmo de roteamento XY

entrada $(X_e, Y_e), (X_r, Y_r)$ – coordenadas das chaves emissora e receptora
retorna (X_i, Y_i) – posição da chave intermediária a receber o pacote

início

$(X_i, Y_i) \leftarrow (X_e, Y_e)$

enquanto $X_i \neq X_r$
  se $X_i < X_r$
    $X_i \leftarrow X_i + 1$
    senão
      $X_i \leftarrow X_i - 1$
  Transmita o pacote a (X_i, Y_i);

Transmita o pacote a (X_i, Y_i); – chegamos à "coordenada X" da chave receptora.

enquanto $Y_i \neq Y_r$
  se $Y_i < Y_r$
    $Y_i \leftarrow Y_i + 1$
    senão
      $Y_i \leftarrow Y_i - 1$
  Transmita o pacote a (X_i, Y_i);

Transmita o pacote a (X_i, Y_i); – aqui, chegamos à chave receptora.

fim

De implementação simples, o algoritmo de roteamento XY é popular e figura como a base de outros que vieram a contornar algumas de suas deficiências, como a forte dependência das condições operacionais das chaves nas rotas pré-computadas, incluindo seu grau de congestionamento. Ocorre que, havendo recursos a produzir e despachar, via chave associada, uma quantidade relativamente grande de mensagens, bem como um outro a receber um volume de mensagens grande em relação aos demais, poderá haver tráfego substancial na rede e consequente congestionamento em chaves que lhes sejam ortogonais

porque as rotas computadas com o algoritmo XY tenderão a reusá-las. Em nosso exemplo, um suposto regime intenso de emissão de mensagens de (0,0) a (2,2) tenderá, sem considerarmos qualquer outro fator, a ocupar e, talvez, congestionar as portas Oeste e Leste da chave em (1,0), Oeste e Norte daquela em (2,0) e Sul e Norte de (2,1), além das portas Leste de (0,0) e Sul de (2,2). Sendo esta uma NoC do tipo direta, pressupõe-se ainda que as chaves se prestam à conexão do recurso à rede e, também, a roteamento em geral, na NoC, permitindo inferir que tarefas eventualmente em execução nos recursos conectados a estas chaves teriam dificuldades em produzir ou receber mensagens para outras, a depender do arbítrio feito pelas chaves.

2.4 Acerca da plataforma MEMPHIS

A implementação de nosso experimento foi feita sobre a plataforma de simulação MEMPHIS (*Many-core Modeling Platform for Heterogeneous SoCs*). A plataforma MEMPHIS é uma contribuição do Grupo de Apoio ao Projeto de Hardware (GAPH) da Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS) (RUARO et al., 2019). Distribuída de forma livre sob licença GPL, a MEMPHIS implementa um sistema *many-core* onde é possível lançar mão de *hardware* descrito em VHDL ou SystemC para compor um MPSoC (*Multiprocessor System on Chip*). Neste MPSoC, o sistema de comunicação que interconecta PEs e periféricos, se houver, é uma NoC de topologia Malha 2D. A MEMPHIS é distribuída com uma coleção de ferramentas para o apoio ao desenvolvimento de aplicações para execução na plataforma, incluindo um *debugger* gráfico, mostrado na figura 16.

A MEMPHIS, por sua vez, deriva de um outro projeto do GAPH, a plataforma HEMPS (*HERMES Multiprocessor System on Chip*). Proposta em 2009 (CARARA et al., 2009), a HEMPS foi concebida como uma plataforma de simulação que pudesse apoiar o desenvolvimento de modelos em VHDL que podem ser usados em processos de síntese de circuitos integrados. A contribuição da MEMPHIS é a possibilidade de emular um sistema heterogêneo, enquanto a HEMPS objetivava implementar SoCs homogêneos.

Os PEs podem ser considerados unidades computacionais completas, visto que dispõem de uma Unidade Central de Processamento (*Central Processing Unit* - CPU), memória dinâmica local (*Dynamic Random Access Memory* - DRAM), alguma lógica de controle e uma chave de interconexão. O PE que a MEMPHIS disponibiliza por padrão, e que foi o utilizado em nossa implementação, usa uma CPU Plasma (RHOADS, 2001),

Figura 16: Debugger, uma ferramenta de apoio ao uso da MEMPHIS

um processador de tecnologia não-proprietária, de 32 bits e cujo conjunto de instruções é compatível com o MIPS (ROWEN et al., 1984).

Sobre as chaves de interconexão agregadas aos PEs, estas são implementadas com a HERMES, interligadas em uma configuração de malha. Um bloco de lógica complementar implementa uma interface de rede, conectada à porta local da chave. De fato, este bloco contém não somente esta interface, mas também um controlador de Acesso Direto à Memória (*Direct Memory Access* - DMA). A figura 17 ilustra os blocos básicos deste PE, bem como sua conexão a uma chave de interconexão da rede intra-chip.

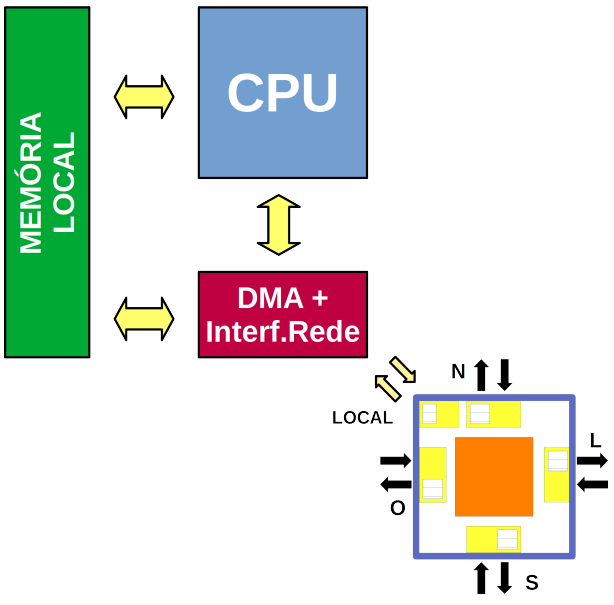


Figura 17: Um PE associado a uma chave de interconexão

A MEMPHIS simula sistemas multiprocessados equipados com uma rede intra-chip de topologia malha 2D, a exemplo do mostrado na figura 18. As chaves de interconexão conectam-se a até quatro outras vizinhas e ao recurso de computação local. A comunicação entre chaves é ponto-a-ponto, por meio de barramentos de 32 bits de largura.

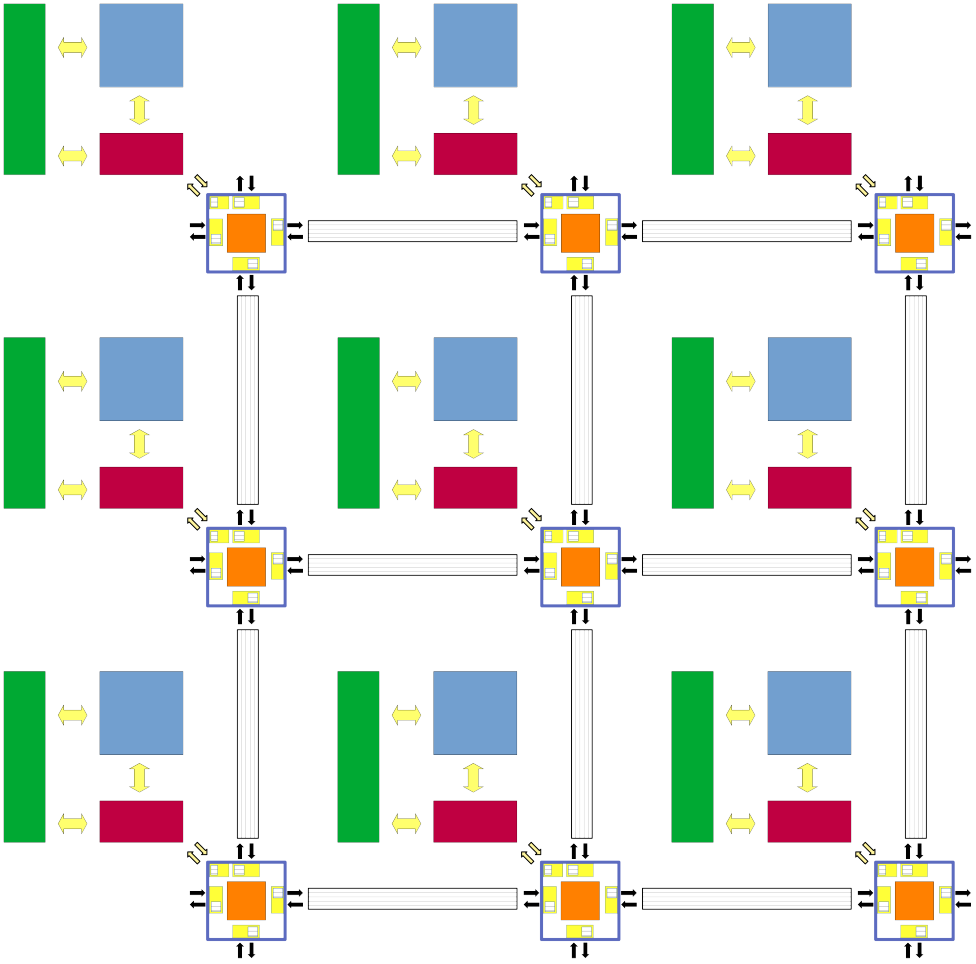


Figura 18: Um sistema com uma rede de topologia malha 2D

O roteamento de pacotes é feito segundo o algoritmo XY e, do ponto de vista da organização e transmissão dos *flits*, eles são distribuídos pela rede para posterior agregação no destino, reconstruindo o pacote. Este modo de chaveamento é chamado *wormhole*, utilizado em muitos modelos de NoC e que, apesar de permitir menor latência no encaminhamento do dado, é fortemente dependente da disponibilidade de conexões físicas disponíveis, entre chaves, ao encaminhamento de um *flit*, aumentando o risco de *deadlock* (PASRICHA; DUTT, 2008).

Dentre os parâmetros atinentes à NoC que se pode configurar, na MEMPHIS, há as dimensões da malha, a quantidade de *buffers* por chave e, também, o tamanho dos *clusters* de chaves, se o usuário quiser operar desta forma.

Além dos recursos conectados à NoC, é possível especificar (em VHDL ou SystemC) periféricos, como dispositivos de entrada e saída ou de memória externa ao MPSoC. Estes periféricos devem ser conectados, necessariamente, a uma chave nos bordos da MPSoC. Há um periférico que precisa ser obrigatoriamente conectado ao sistema, o AppInjector, cuja definição é parte da distribuição da MEMPHIS e é responsável pela injeção da carga dos recursos de processamento, os PEs. Por carga, entendemos aqui os programas que serão alocados aos PEs, chamados tarefas. As tarefas são desenvolvidas e compiladas visando os dispositivos presentes no sistema, à semelhança do que faríamos com MPSoCs físicos. Se as tarefas devem executar em PEs calcados na CPU Plasma, o compilador utilizado deve ser um *cross-compiler* para arquiteturas MIPS.

As tarefas, como explicávamos, são a carga dos PEs, os programas que efetivamente executarão como parte de uma aplicação. Uma aplicação, no escopo da MEMPHIS, é uma coleção de tarefas que implementam suas funcionalidades básicas e que podem ser executadas em paralelo. Há um mecanismo, implementado por meio de uma Interface de Programação de Aplicações (*Application Programming Interface* - API) que permite não somente à tarefa interagir com a plataforma MEMPHIS, tomando dados de relógio (o contador de pulsos de *clock* propagados na MPSoC, em verdade) ou ecoar mensagens que são registradas em arquivos de registro associados aos PEs como, e principalmente, trocar mensagens com outras tarefas da aplicação. Por *default*, as aplicações constituem domínios fechados de comunicação; significa que uma tarefa só pode trocar mensagens com outras sob a mesma aplicação. Um outro aspecto importante é que as transações de comunicação precisam ser feitas, em código, enunciando os nomes das tarefas em execução na MPSoC, o que implica dizer que é requisito, para iniciarmos uma transação de comunicação na MEMPHIS, que a tarefa consumidora esteja instanciada e rodando, não sendo possível buferizar mensagens para tarefas que ainda não estão instanciadas na MPSoC.

Em relação à alocação de tarefas na MPSoC, há dois modos de trabalho possíveis, na MEMPHIS. É possível especificar, explicitamente, que tarefa deve ser alocada a qual PE, este por sua vez conectado a um nó da rede. A alternativa é deixar que a MEMPHIS faça a alocação das tarefas por si. Neste caso, a MEMPHIS aloca as tarefas aos PEs usando, unicamente, a ordem alfabética atinente aos nomes destas tarefas. As tarefas de uma mesma aplicação são agrupadas em PEs próximos e o algoritmo usado na alocação destas tarefas aos PEs é a Busca em Diamante (*Diamond Search* - DS) (JAIN; JAIN,

1981), o que ajuda a evitar, de antemão, o alinhamento de PEs executando tarefas da mesma aplicação em colunas e linhas, algo eventualmente benéfico em cenários de uso do algoritmo de roteamento XY (porque ajuda a evitar o congestionamento de chaves de interconexão que tendem a ser muito usadas) sem, no entanto, deixá-los a mais do que uns poucos nós de distância, na rede.

2.5 Considerações finais

As redes intra-chip, objeto de considerável interesse e investimento, mais do que uma opção de interconexão de blocos em sistemas embutidos, são ferramentas que vieram a resolver problemas múltiplos que assolaram pesadamente a indústria de sistemas microprocessados há duas décadas, tornando-se ainda um importante habilitador em termos de possibilidades construtivas e no desenvolvimento de novos produtos. Grosso modo, as redes intra-chip trazem ao projeto do *chip* conceitos de conectividade já maduros porque têm grande aderência aos aplicados em sistemas como redes de computadores. Por outro lado, em um momento em que a densidade de transistores em circuitos integrados é alta a ponto de permitir trazer ao *chip* os recursos computacionais que outrora precisavam de uma sala espaçosa, refrigerada e bem servida de energia elétrica, é interessante e um alento constatar que chegamos ao ponto em que mesmo a rede, hoje, está posta em um *chip*. Melhor do que isso, que o projeto de sistemas complexos como microprocessadores é grandemente facilitado porque um de seus complicadores mais intricados, a interconexão dos módulos internos, é resolvido de forma mais simples, mais versátil e mais interessante, com uma NoC.

Capítulo 3

TRABALHOS RELACIONADOS

O interesse na aplicação de técnicas próprias da vertente probabilística da chamada "Inteligência Artificial" é crescente. Entretanto, a isso conjugam-se as dificuldades impostas pelo processamento das volumosas massas de dados que lhes são próprias, não raro também submetido a requisitos de tempo agressivos para que inferências geradas sejam úteis – um veículo autônomo, um sistema de gestão industrial ou uma aplicação militar seriam bons exemplos. Assim, abordagens aceleradas são bem-vindas, como o uso de *hardware* "especializado" na implementação de projetos em Inteligência Artificial. Um exemplo seria a implementação de uma Rede Neural Convolucional em *hardware*, buscando executá-la em menor tempo.

Há ao menos duas motivações importantes para a pesquisa sobre a implementação de sistemas de aprendizado de máquina com forte enfoque no uso ótimo do *hardware*: melhor desempenho na execução da aplicação e maior eficiência energética. A primeira decorre naturalmente do problema supracitado, onde a premissa é a de que um melhor aproveitamento do *hardware* implicaria maior desempenho. Os aspectos relacionados podem ir do projeto do *chip*, com customizações visando acelerar a execução de uma aplicação específica, até o melhor aproveitamento de um sistema preexistente, visando explorá-lo ao máximo, frequentemente lançando-se mão de processamento paralelo (HEIDARI et al., 2022). E a segunda, eficiência energética, é consequência do interesse crescente em abordagens computacionais que implicam distribuir o processamento de lotes de dados em conjuntos de sistemas computacionais de pequena capacidade. Domínios como os apelidados por "Internet das Coisas" e "Computação em Neblina" ilustram bastante bem o cenário, onde a computação se dá de forma distribuída entre os dispositivos ditos "terminais". Não raro, tais sistemas estão submetidos a requisitos de dispêndio energético

estritos (SZE et al., 2018). Deste modo, abordagens que visem um melhor aproveitamento da memória local destes dispositivos, bem como a simplificação dos processos que devem executar e, ainda, redução do *overhead* de comunicação necessário à sua interconexão, quando possível, contribuem para com o aspecto da melhoria no dispêndio energético (NABAVINEJAD et al., 2020).

De fato, a questão do dispêndio energético exigido no treinamento e aplicação de modelos de Aprendizado de Máquina, ao lado da necessidade de ampliação do acesso a tecnologias de aceleração aplicáveis ao desenvolvimento e implementação de soluções modernas em Inteligência Artificial, são vistos como pontos de atenção e interesse no sentido de garantir à área progresso e disseminação constantes, longevos e sustentáveis (STRUBELL; GANESH; MCCALLUM, 2020).

3.1 Conceitos

De (MITTAL, 2020) entendemos que, por "*hardware* especializado", tratamos de qualquer tecnologia que habilite ou produza *hardware* computacional que o distingua de implementações "de uso geral". Há, por exemplo, aspectos interessantes no uso de tecnologias que permitam a reconfiguração *a posteriori* do *hardware*. A sugestão é feita mediante a intenção econômica de permitir o reaproveitamento deste *hardware* à medida que evolui o estado da arte atinente às técnicas aplicadas na solução, bem como é aprimorada a própria implementação. O autor sugere, por exemplo, a consideração da tecnologia *Field-Programmable Gate Arrays* (FPGA) neste tipo de projeto, em detrimento do uso de *chips* especializados e não reconfiguráveis, os *Application-Specific Integrated Circuits* (ASIC). Faz ressalva, entretanto, que muitos tipos de Redes Neurais Convolucionais têm requisitos de quantidade de espaço em memória que facilmente ultrapassam a disponível em FPGA, o que precisa ser considerado quando da concepção do projeto.

Em (LI; RUAN; YANG, 2020), há uma discussão sobre a diminuição da necessidade de acessos em níveis mais custosos da hierarquia de memória, bem como de operações de processamento em si. Para tanto, tira-se proveito da organização do problema em implementação realizada sobre uma NoC. O conceito já havia sido explorado em trabalhos como (SZE et al., 2018), onde há uma proposta sobre uma arquitetura cuja finalidade é a aceleração do processamento de partes de uma dada CNN. Ela é calcada em um arranjo espacial de PEs compostos por alguns registradores, uma unidade lógico-aritmética

(*Arithmetic-Logic Unit* - ALU) e alguma lógica de controle. Os PEs seriam interligados por uma NoC, com o intento de reduzir o número de transações entre estes PEs e um *buffer* intermediário comum, ou mesmo dali à memória externa ao *chip* (*Dynamic Random Access Memory* - DRAM). O eventual aumento de desempenho seria derivado, justamente, da redução de operações de busca de dados ao longo da hierarquia de memória.

Em (YANG et al., 2019), explora-se também esta ideia, explicando que as NoCs habilitam certa flexibilidade conceitual na concepção arquitetural de *Multiprocessor Systems-on-Chip* (MPSoCs), porém incluindo os componentes da própria NoC, como as chaves de interconexão. Os autores propõem uma arquitetura de memória para os roteadores de uma NoC de tecnologia híbrida, envolvendo *Static Random-Access Memory* (SRAM) e as novas *Spin-Torque Transfer Magnetic RAM* (STT-RAM). O propósito seria tirar proveito das possibilidades ofertadas por este arranjo por habilitar o chamado "processamento em memória".

Há discussões similares, como em (OVTCHAROV et al., 2015), no contexto de uma aplicação comercial, em produção e no âmbito de um *data center*. De fato, segundo (SHAWAHNA; SAIT; EL-MALEH, 2019), já havia crescimento do interesse na implementação, em *hardware* especializado, de estruturas dedicadas ao processamento, em especial, das camadas convolucionais de uma CNN, já no início dos anos 2000.

Também há outras discussões à volta do tema, porém com especial ênfase às possibilidades decorrentes da otimização do projeto das NoCs. Em (VENKATARAMAN; KUMAR, 2019), é explicado que, por premissa, se a comunicação entre os núcleos da rede é fator determinante do grau de dispêndio energético e de consumo de área útil no *chip*, a otimização da arquitetura de *hardware*, simplificando o roteamento, pode trazer ganhos expressivos. Em projetos deste tipo, para as fases de otimização, pode-se lançar mão de algoritmos vários, muitos deles atinente ao campo de Inteligência Coletiva, como a técnica de otimização baseada no comportamento das formigas-leão (*Ant Lion Optimization* – ALO) (MIRJALILI, 2015).

3.2 Desempenho

Segundo (PASRICHA; DUTT, 2008), é natural a convergência entre a intenção de acelerar o processamento de partes computacionalmente custosas de uma aplicação em *hardware* especializado e a aplicação, neste tipo de projeto, de redes intra-chip. Isso porque a den-

sidade de transistores nos *chips* foi crescendo, habilitando a incorporação de mais e mais subsistemas aos sistemas embarcados e sua interligação, com isso, torna-se crescentemente desafiadora. Assim, um método para a interconexão destes, que seja escalável e que auxilie na sustentação do desempenho do sistema como um todo é bem-vindo. É neste contexto que as NoCs ganham maior interesse. Ocorre que as tecnologias vigentes de integração de circuitos habilitam densidades que visitam o chamado "domínio profundamente submicrométrico" (*Deep Sub-Micron domain* - DSM)[1]. Decorre daí que estes circuitos integrados possivelmente seriam mais vulneráveis a problemas de natureza física, como cargas reativas parasitárias. As consequências iriam de problemas de sincronização de sinais até forte não-determinismo do comportamento do sistema em função de fatores construtivos ou ambientais, como pequenas variações no processo de fabricação do *chip* ou mudanças de temperatura.

Há também considerações sobre a crescente complexidade dos sistemas embarcados, habilitada pelo progresso em tecnologias de integração, permitindo densidades de componentes por área no *chip* cada vez maiores. Em (CHOI et al., 2018) usa-se o termo *Datacenter-on-Chip* (DoC) para referenciar cenários em que vários sistemas *many-core* poderiam eventualmente coexistir em um único *chip*. De fato, o treinamento de CNNs, que frequentemente envolve grandes quantidades de dados, é, não-raro, implementado em sistemas *many-core* híbridos, frequentemente aplicando usando processadores gráficos (*Graphic Processing Unit* - GPU) conjugadas a CPUs. Ocorre que, dada a natureza destas operações, pode-se tirar muito proveito de processamento paralelo e, por isso, GPUs, ou a combinação destas com CPUs, são frequente eleitas como tecnologia a suportar implementações com este fim. Entretanto, latência e dispêndio energético impostos pelo barramento que conecta CPUs e GPUs consistem bom motivo para considerar combinar a ambos em um único e heterogêneo sistema embutido multiprocessado (*Chip Multiprocessor* - CMP). A ideia é, por meio de interconexão interna ao *chip*, diminuir, tanto quanto possível, a incidência destes problemas quando do uso de barramentos externos a ele.

Da mesma forma em que há a busca de formas de implementar, de forma eficiente, modelos de aprendizado de máquina em plataformas especiais de *hardware*, também há esforço em buscar modelos cuja computação tire proveito deste *hardware*. É o caso das Redes Neurais Binarizadas (*Binarized Neural Networks* - BNNs), onde pesos e parâmetros

[1] As tecnologias de integração DSM implicam densidades de $90\eta m$ por transistor, patamar atingido por volta de 2003. Em 2020, este valor já teria diminuído de uma ordem de grandeza.

são representados e manipulados em sua forma binária, o que inclui toda a aritmética necessária à sua operação (COURBARIAUX et al., 2016). Além de menor consumo de recursos computacionais como memória, operações aritméticas em nível de bits tendem a ser mais velozes e menos dispendiosas em termos energéticos. De fato, as BNNs têm, como vantagem imediata, a isenção da necessidade de suporte aritmético extenso e, por vezes, complexo e dispendioso em termos de recursos computacionais, como os necessários ao suporte de operações de ponto fixo ou flutuante, na maioria das vezes essencial para realizar o processamento de modelos de Aprendizado de Máquina. O próprio dispêndio de memória usado no armazenamento de pesos e atributos é ponto de atenção, aqui, porque a representação de números não inteiros, como o são estes parâmetros, pode demandar muitos *bytes* porque se guarda mantissas e multiplicadores, por exemplo. (UMUROGLU et al., 2017).

É importante notar que, em uma CNN, diferentes camadas tendem a ser intensas, por premissa, no concernente à produção e consumo de fluxos de dados, bem como ao consumo de memória e tempo de processamento. Na arquitetura AlexNet, por exemplo, as camadas convolucionais deterão cerca de 5% dos pesos sinápticos apenas, enquanto as camadas totalmente conectadas respondem por 95% destes. Em contrapartida, estas mesmas camadas responderiam por cerca de 93% e 7% do tempo despendido em computação propriamente dita (LI et al., 2018). Camadas de subamostragem, por outro lado, tendem a ser menos dispendiosas, sob estes parâmetros (MOTAMEDI; GYSEL; GHIASI, 2017). Estas considerações são importantes na medida em que habilitam um olhar visando a agregação de funcionalidades em camadas de diferentes tipos, objetivando maior desempenho e eficiência no uso de recursos computacionais.

Outra consideração importante a fazer é no concernente ao uso da hierarquia de memória. Há interesse em abordagens que tirem o máximo proveito das memórias locais dos Elementos Processadores no sistema, evitando o dispêndio em tempo e energia a empenhar no uso de sistemas de memória externos ao SoC (ABDELOUAHAB et al., 2017). A busca por estratégias que permitam o uso eficiente de níveis hierárquicos de memória mais próximos dos Elementos Processadores (PARK; SUNG, 2016) e de modelos que facilitem a obtenção deste objetivo, como as BNNs (UMUROGLU et al., 2017) são considerados basilares e objeto de interesse e estudo, enquanto habilitadores ao porte de modelos de Aprendizado de Máquina a sistemas embarcados.

Em suma, estratégias que visem reduzir transferências de dados em níveis hierárquicos de memória de maior tempo de acesso parecem ser cruciais na implementação de modelos como as CNNs em sistemas embutidos. De fato, idealmente, níveis hierárquicos de memória mais altos, frequentemente externos ao SoC, só deveriam ser utilizados na aquisição e posterior entrega de dados pelo modelo em execução. Assim, considerar cuidadosamente os fluxos de dados no projeto, de maneira a permitir que mapas de atributos possam ser tramitados entre camadas utilizando apenas sistemas de memória hierarquicamente mais baixos, por vezes realizando a fusão de camadas em fluxos de dados que lhes sejam transversais, habilitam, potencialmente, ganhos em desempenho importantes, dada a natureza intensiva dos fluxos de dados gerados pelas camadas convolucionais de uma CNN (ALWANI et al., 2016).

Finalmente, e no contexto desta dissertação, é importante citar a discussão em (FRANÇA et al., 2021) por conta de sua afinidade com o projeto experimental que desenvolvemos. Ali, explora-se a implementação de CNNs em um contexto de Redes Neurais em *Hardware* (*Hardware Neural Networks* - HNN) explorando forma de obter maior desempenho e menor custo de implementação mediante a adaptação dos fluxos de dados de maneira a procurar evitar o uso de sistemas de memória dinâmica externas ao SoC. Aspectos cruciais na implementação de HNNs são discutidos pelo autor, como os compromissos entre acurácia, desempenho e eficiência energética, o que é explorado de forma bem-sucedida em seu trabalho experimental.

3.3 Tendências

Embora as NoCs possam operar com o chaveamento de circuitos ou de pacotes, menciona-se em (M.WANJARI; AGRAWAL; V. Kshirsagar, 2015) e (CARARA; CALAZANS; MORAES, 2008) que a tramitação de dados em sistemas intra-chip, suportados frequentemente por NoCs, seria calcada, cada vez mais, no chaveamento de pacotes. Isso implica interesse crescente em técnicas de roteamento, eventualmente tipificadas sob três categorias: *Store and Forward* (SAF), *Wormhole* (WH) e *Virtual Cut Through* (VCT). Entretanto, pontua-se que todas estas classes de técnicas sofreriam do problema de bloqueio *Head-on-Line* (HoL), resultante da retenção de dados nos *buffers* de entrada do elemento roteador. Em (DALLY, 1992) já havia sido proposto o uso de canais virtuais como maneira de evitar *deadlocks* no roteamento de pacotes na NoC, um exemplo entre várias abordagens diferentes

para a alocação de *buffers* já propostas. A ideia é retomada em (SUSEELA; MUTHUKUMAR, 2012), que sugere o uso de canais virtuais como vias de *loopback* associados aos *buffers* de entrada como maneira de reduzir a sobrecarga sobre estes *buffers* e, ainda, evitar bloqueios do tipo HoL.

Em (ABBA; LEE, 2017) há uma discussão interessante sobre a construção de NoCs "conscientes", capazes de se adaptar de forma autônoma em presença de uma falha ou gargalo. Este trabalho discute a aplicação de técnicas de otimização como a *Particle Swarm Optimization* (PSO) no desenvolvimento do algoritmo de roteamento, apontando ainda a forte tendência e interesse no uso de técnicas de similar inspiração biológica. O motivo não seria outro que não a forte aderência deste tipo de abordagem a cenários complexos de roteamento, como as NoCs.

3.4 Considerações Finais

O interesse no desenvolvimento de métodos e tecnologias que visem acelerar o processamento de modelos de aprendizado de máquina como as CNNs se dá pela confluência de fatores vários, dos quais ressaltam-se: a popularização do uso de técnicas de Inteligência Artificial em uma gama crescente de problemas e aplicações do cotidiano de todos, a igual popularização do uso de dispositivos embarcados na execução destas aplicações, a consequente necessidade de buscar formas de executar tais aplicações com o melhor desempenho computacional e energético possível e a disponibilidade mais ampla de dispositivos embarcados que integram mais recursos computacionais. Nesse sentido, expedientes tecnológicos que contribuam para com a simplificação do projeto de *chips*, reuso destes projetos e artefatos derivados e, ainda, redução do tempo para seu lançamento no mercado são de todo o interesse. Assim, as redes intra-chip figuram como habilitadores importantes ao intento, permitindo não somente atingir mais facilmente estes objetivos de ordem econômica e de mercado como, também, o incremento da capacidade computacional de sistemas embarcados, vital à indústria de semicondutores e particularmente importante no segmento de aceleradores.

Capítulo 4

IMPLEMENTAÇÃO

NESTE capítulo, explicamos como o problema que abordamos foi considerado e organizado de maneira a permitir-nos gerar a implementação de uma Rede Neural Convolucional (*Convolutional Neural Network* - CNN) em uma plataforma que simula um sistema multiprocessado *on chip*. Entenderemos como esta implementação foi concebida a partir da análise visando identificar as funções básicas que constituem uma CNN, seguida de avaliação sobre o fluxo de dados segundo a arquitetura preleita de referência (a LeNET-5). Depois disso, compreenderemos como, feito esse estudo preliminar e considerando estas funções e o fluxo de dados, chegamos à forma como estes elementos ou blocos funcionais da CNN poderiam ser reorganizados de forma a constituir tarefas. Essas tarefas, em um segundo momento, se tornariam as cargas de trabalho a serem distribuídas pelos Elementos Processadores no sistema embarcado.

Explicamos também os detalhes de implementação de nosso experimento principal no âmbito da MEMPHIS (*Many-core Modeling Platform for Heterogeneous Systems on Chip*), desde o planejamento destas tarefas até a sua codificação propriamente dita, incluindo a estratégia de comunicação adotada para conectá-las por meio do uso da Interface de Programação de Aplicações (*Application Programming Interface* - API) da plataforma com este fim.

Por fim, detalhamos algumas abordagens utilizadas no intuito de contornarmos dificuldades decorrentes de limitações que se impuseram por conta de peculiaridades atinentes à plataforma utilizada, sobretudo no concernente às tratativas numéricas necessárias às operações algébricas próprias ao processamento de um modelo de Aprendizado de Máquina, como uma CNN.

4.1 Análise preliminar do problema

Em face de nosso intento de produzir uma implementação de uma CNN de arquitetura baseada na LeNET-5 explorando paralelismo, iniciamos por uma análise do problema visando verificar a maneira como o fluxo de dados se dá no âmbito da arquitetura de referência, procurando entender onde não haveria co-dependência de dados, o que implicaria oportunidades para paralelização. A abordagem utilizada foi, em primeiro lugar, levantar que funções, ou blocos funcionais, constituiriam as partes de uma CNN. Compreender isto *a priori* seria importante porque, por premissa, se há uma torrente de dados em fluxo pelas diferentes camadas de uma CNN, então há de se ter, em cada camada, blocos funcionais que operem estes de dados, produzindo novos corpos de dados como suas saídas, subsidiando uma camada subsequente, do modelo. Com isso, compreender esse fluxo nos ajudaria a entender como tirar dele proveito visando encontrar, como mencionamos, oportunidades de paralelização. Assim, uma vez identificados estes blocos funcionais, seria possível entender que operações estariam ocorrendo ali especificamente e de que modo estariam postos enquanto receptores dos dados produzidos por outros blocos, elicitando que blocos seriam produtores e/ou consumidores de dados em relação aos demais. De outro modo, procuramos transpor nosso entendimento sobre a CNN como uma sucessão de camadas a processar um dado de entrada em uma certa sequência no tempo para uma coleção de canais por onde fluiriam estes dados. Posto este entendimento, seria possível avaliar como e quando estes blocos funcionais deteriam, entre si, dependência de dados (e quais não). A partir disso, seria possível dizer que blocos funcionais poderiam ser operados em paralelo, a depender dos recursos computacionais à disposição no sistema embutido emulado pela MEMPHIS.

4.1.1 Sobre as funções identificadas na arquitetura de referência

Embora uma dada arquitetura de CNN possa conter especificidades em termos de blocos funcionais e operações atinentes, de maneira geral (e para a LeNET-5, em particular), os seguintes blocos funcionais estariam presentes, todos eles especializados segundo a camada ou ponto, no processo, onde são aplicados:

- ***Kernels* deslizantes**: percorrem espacialmente o dado de entrada produzindo um mapa de atributos decorrente da submissão destes dados a um Perceptron.

- **Subamostragem**: sua saída decorre da submissão de um mapa de atributos de entrada a uma função de subamostragem.

- **Perceptrons**: o bloco funcional que operará os produtos internos ao longo da CNN.

- **Aquisição de dados**: blocos de apoio destinados ao consumo e organização de dados despachados por tarefas produtoras. Fazem uso da API de comunicação da plataforma de simulação (MEMPHIS), reconstituindo e organizando os dados recebidos na memória local do Elemento Processador (PE).

- **Emissão de dados**: blocos de apoio com a função de enviar dados a outras tarefas. Operarão o dado a ser despachado, disposto em uma estrutura de dados posta na memória local do Elemento Processador, de maneira a transformá-lo em um vetor unidimensional para, em seguida, enviá-los às tarefas consumidoras. Lançam mão da API de comunicação da plataforma de simulação (MEMPHIS).

É importante ressaltar que os Perceptrons podem existir como blocos funcionais em si mesmos, quando consideramos a porção totalmente conectada da CNN (notadamente, uma *Multi-layer Perceptron* - MLP) mas, também, como um elemento constituinte de um *kernel*. Outro ponto importante é que funções de ativação e operações de *pooling* poderiam ser, do ponto de vista de implementação, considerados blocos funcionais em si; porém abstraímos esta noção em favor do entendimento de que estes elementos seriam uma implicação destes que citamos: funções de ativação são parte constituinte de Perceptrons, por exemplo. A tabela 4 mostra o mapa de presença destes blocos pelas camadas da arquitetura LeNET-5.

Tabela 4: O mapeamento de blocos funcionais pelas camadas da arquitetura LeNET-5

Blocos funcionais / Camadas	C1	S2	C3	S4	C5	F6	OUTPUT
kernel deslizante	✓		✓				
Subamostragem		✓		✓			
Perceptron	✓		✓		✓	✓	✓
Aquisição de Dados		✓	✓	✓	✓	✓	✓
Emissão de Dados	✓	✓	✓	✓	✓	✓	

Cabe observar que não há um *kernel* deslizante em C5, na arquitetura LeNET-5 canônica. Isso se dá porque, usando as especificações originais da arquitetura no concernente às dimensões do dado de entrada e mapas de atributos consequentes, a entrada desta camada já tem as dimensões do *kernel* – 5x5 – de maneira que não haveria, a rigor, comportamento de varredura nesta camada. De outro modo, podemos entender a camada C5 como totalmente conectada, neste contexto.

Outro aspecto importante a pontuar em nossa implementação é o de que, nesta versão do experimento, nenhum periférico de entrada e saída foi conectado ao MPSoC no âmbito do simulador (além do AppInjector, dispositivo padrão necessário à inicialização do sistema e injeção das cargas dos PEs no sistema). Significa que todos os dados são mantidos e processados no âmbito das memórias locais dos Elementos Processadores e, também, que a apresentação de uma imagem de entrada à CNN e a transferência de aprendizado ao experimento mediante a adição dos pesos treinados são feitos diretamente via código-fonte das tarefas atinentes à aplicação. É por esta razão que não há um bloco de Aquisição de Dados associado a C1. Do mesmo modo, a emissão da classe associada à imagem de entrada, feita por OUTPUT, também é feita por escrita direta em arquivo de registro utilizando a API da MEMPHIS, que detém um método para este fim (a saber, o comando `Echo()`). Desta forma, também não há um periférico de saída e, por isso, não temos a presença de um bloco de Emissão de Dados associado à camada OUTPUT.

4.2 Sobre os dados de entrada e a carga de parâmetros treináveis da CNN

O banco de dados MNIST (*Modified National Institute of Standards and Technology*) reúne imagens de dígitos de 0 a 9 escritos à mão, a exemplo dos mostrados na figura 19, servindo, primariamente, ao desenvolvimento e teste de algoritmos e técnicas de reconhecimento de imagens e padrões ou outras aplicações afins (LECUN; CORTES; BURGES, 1998). Originalmente, reúne 60 mil imagens para fins de *design* e 10 mil para testes. Embora não restrita a aplicações orientadas ao reconhecimento de caracteres, a arquitetura de CNN LeNET-5 foi originalmente proposta usando o MNIST como base de dados, como comentado em (LECUN et al., 1998).

Em nosso experimento, utilizamos 200 imagens do conjunto de testes do MNIST, 20 exemplares por dígito (de 0 a 9). As imagens foram tratadas de forma a serem dispostas

Figura 19: Exemplo de casos contidos do conjunto de dados MNIST

em um retículo de 32x32 *pixels*, em atenção às especificações originais da arquitetura LeNET-5. As imagens são codificadas a um único canal de cor (*grayscale*), um *byte* por *pixel*. Assim, o nível de luminância associado ao *pixel* será codificado com um valor variando entre 0 (luminância neutra) e 255 (luminância máxima).

Uma vez que não utilizamos periféricos de entrada em nosso experimento, as imagens foram apresentadas à CNN embutindo-as diretamente no código (em linguagem de programação C) das tarefas responsáveis pelo processamento da primeira camada convolucional, na forma de um vetor de inteiros. Os valores dos *pixels*, para tanto, foram previamente extraídos e organizados em arquivos de cabeçalho, incluídos quando da compilação da aplicação. A alternância dos cabeçalhos com os dados das imagens a cada teste foi feita com um *script* de apoio, também automatizando outras tarefas na execução dos experimentos.

Os pesos associados aos parâmetros da CNN foram portados de conjuntos de dados gentilmente cedidos pelo pesquisador Alexandre Bazyl, do PEE/COPPE/UFRJ, ativo gerado como parte do processo de desenvolvimento do experimento descrito em (FRANÇA et al., 2021). A extração dos valores de seu contêiner original (a saber, arquivos HDF5), assim como sua posterior notação em forma de frações e organização na forma de arquivos de cabeçalho, foi feita por meio de *scripts* de apoio. Estes scripts, em linguagem Python e utilizando o *framework* Keras (CHOLLET et al., 2018), visavam permitir a carga dos pesos codificados nestes arquivos em um modelo definido tal como no experimento que origina-

mente os produziu, reexportando-os novamente a um arquivo de dados JSON permitindo, finalmente, realizar o *parsing* dos valores de peso e o seu porte a nosso experimento. Decorre também disso que, da maneira como os pesos foram gerados durante o treinamento original, pressupõe-se conexão plena entre os mapas de atributos de S2 e C3, o que difere do proposto na arquitetura LeNET-5 canônica (o que acatamos em nosso experimento, realizando o mesmo mapeamento completo). Também aqui optamos por embutir os parâmetros da CNN em tempo de compilação, diretamente no código, por simplicidade na implementação do experimento. Houve a necessidade de notação dos valores na forma de frações (na forma de tuplas de inteiros definindo um numerador e um denominador) por conta de limitações em nosso ambiente experimental, que suporta apenas tipos numéricos inteiros. A sugestão de operar com frações como forma de tramitar números fracionários como um numerador e um denominador inteiros como forma de conservar a precisão advém de (KNUTH, 1981).

4.3 Construção do experimento

Passada a fase inicial de compreensão do problema, fizemos uma implementação paralela do experimento agregando os blocos funcionais atinentes a cada camada da arquitetura de CNN LeNET-5 em tarefas. As tarefas são componentes de software que consituirão, em si, corpos funcionais completos com os quais o sistema interagirá por meio de suas interfaces de comunicação, e cada instância de uma certa tarefa constituirá a carga de trabalho associada a um PE.

Para avaliações de *speed-up*, uma implementação serial da aplicação foi construída, de forma a permitir a extração de tempos de execução executando-a integralmente em um único PE. Entretanto, a implementação paralela foi feita primeiro porque naturalmente adere a dois aspectos de nosso interesse neste trabalho, a saber: aproveitar a disponibilidade de um sistema multiprocessado e dotado de uma NoC como plataforma de execução, e o fato de uma CNN ser um problema que pode ser facilmente decomposto em partes embaraçosamente paralelas, ou seja, naturalmente dividido em blocos funcionais que podem ser executados simultaneamente por não serem interdependentes, no concernente ao fluxo de dados na aplicação.

4.3.1 Sobre a agregação de blocos funcionais em tarefas

Uma vez identificados os blocos funcionais relacionados à nossa arquitetura de referência, pudemos buscar rearranjá-los de forma a agregá-los em tarefas que constituiriam, por sua vez, a carga de trabalho a ser mapeada aos PEs da SoC, na MEMPHIS. Pode-se compreender por tarefa um programa a ser executado em um PE, abrangendo uma coleção de blocos funcionais, incluindo os necessários à operações de comunicação para a troca de dados com outras tarefas. Assim, uma tarefa também pressupõe unidade funcional porque, através da combinação dos blocos que a constituem, também implementam uma funcionalidade completa, com a qual o resto do sistema interage por meio de suas interfaces de comunicação.

Outro ponto é que, sob parâmetros e dados de entrada diferentes, uma certa tarefa pode ser instanciada muitas vezes, o que acomoda-se bem às premissas de uso de um sistema multiprocessado. Por conta disso, foi necessário entender como estes blocos se configuram no concernente às dependências mútuas de dados pois partiria daí a compreensão sobre que blocos deveriam ser arranjados de maneira a operarem como produtores e consumidores de fluxos de dados entre si e quais poderiam executar de forma paralela, por não deterem interdependência de dados.

Cabe definir o conceito de canais, no contexto de blocos funcionais em uma CNN. Chamamos por canais os arranjos de blocos funcionais em série que englobam operações visando a extração de atributos de um mapa de entrada, sendo ainda arranjos que abarcam a blocos funcionais em camadas convolucionais e as camadas de subamostragem que as seguem. Desta maneira, um canal deterá, minimamente, um *kernel* deslizante e um bloco de subamostragem, além de blocos de aquisição e emissão de dados, para interfaceamento. Entendemos que um canal será uma estrutura transversal às camadas da CNN, porque implica agrupamento de blocos funcionais em uma camada convolucional e, também, em uma camada de subamostragem subsequente – camadas diferentes, portanto. Além disso, um canal pode, potencialmente, operar como um *pipeline*, pois seus blocos funcionais constituintes não necessariamente precisam esperar a conclusão de todo o trabalho de outros blocos que os antecedem se os dados atinentes a resultados parciais puderem efluir destes ao longo do processo (um conceito que exploramos em experimentos preliminares, mas não chegamos a usar em nossa implementação da arquitetura LeNET-5). A figura 20

mostra o arranjo de blocos de *kernel* deslizante, subamostragem e de interface (aquisição e emissão de dados) em canais.

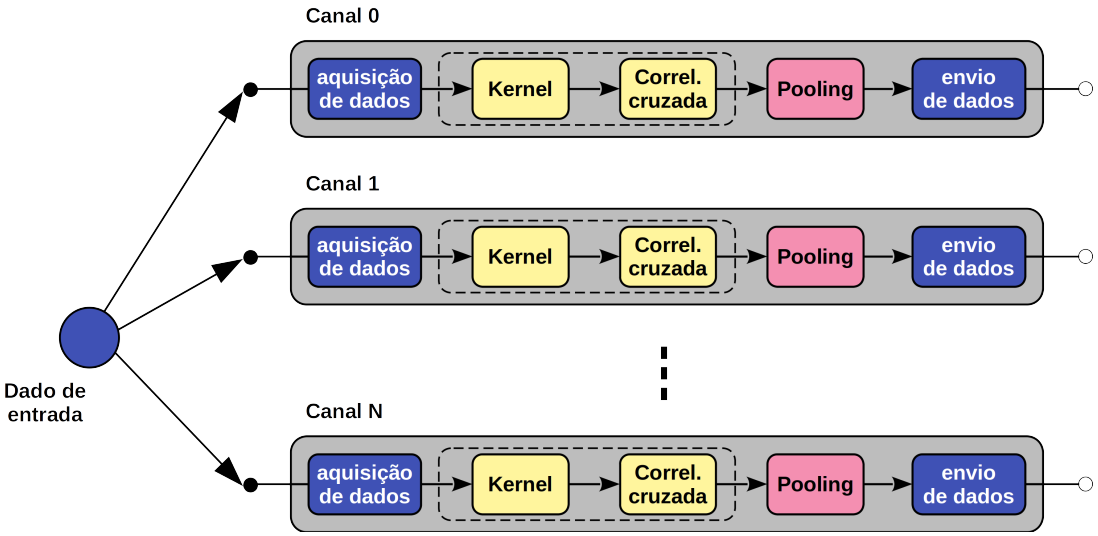


Figura 20: Blocos funcionais estruturados na forma de canais

Em resumo, blocos funcionais que têm dependências de dados com outros estão agregados e em série, mas estes arranjos constituiriam unidades de trabalho independentes porque não detém interdependência de dados entre si, habilitando paralelismo. Ocorre que uma camada convolucional (e uma camada de subamostragem subsequente, quando existente) pode implicar não apenas em um, mas em vários canais que, no entanto, por deterem, cada um destes, suas próprias instâncias dos blocos funcionais mencionados e, também, por não haver relação de produção e consumo de dados entre instâncias de blocos funcionais em canais diferentes, permite-nos intuir que é possível executar a estrutura constituinte de um canal na forma de uma tarefa independente. Isto nos induziu ao entendimento de que seria possível, então, executar os canais em paralelo. Os canais, por sua vez, seriam consumidores e/ou produtores de dados entre si, cabendo aos blocos de aquisição e emissão de dados o interfaceamento entre eles. Os blocos de aquisição de dados se responsabilizariam, ainda, pela organização dos dados emitidos pelos canais que lhe são produtores de fluxos de dados, na CNN.

Na figura 21, vemos como os principais blocos funcionais da CNN se relacionam, entre camadas, no concernente à dependência de dados (um importante balizador no projeto de uma versão do experimento que explore paralelismo). De fato, o entendimento é que a dependência de dados entre blocos funcionais como os de *kernel* deslizante e de subamostragem se deve ao fato de que os *kernels* em C3 e C5 necessitam dos mapas de

atributos gerados por todas as instâncias de S2 e S4, respectivamente. O mesmo acontece com F6, onde cada Perceptron ali instanciado utiliza atributos contribuídos por todas as instâncias de *kernels* de C5.

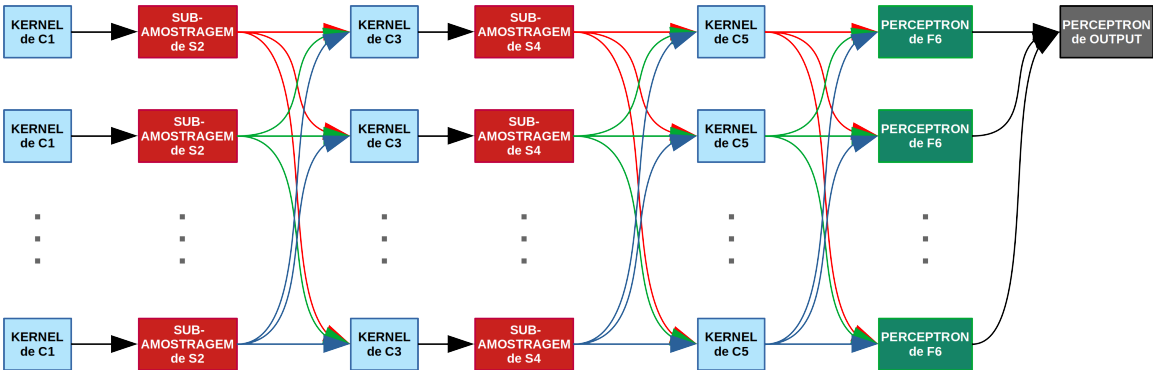


Figura 21: Dependência de dados entre blocos funcionais

A serialização de blocos funcionais não implica que precisem ser usados estritamente desta forma no tempo, fazendo o consumo, processamento e produção de dados apenas quando um bloco que o antecede encerra seu trabalho. Os blocos de emissão e aquisição de dados em tarefas com relação de produção-consumo precisam operar simultaneamente posto que as transações de comunicação, na MEMPHIS, embora assíncronas, necessitam que o produtor e o consumidor de uma mensagem estejam instanciados e ativos na SoC.

Algumas oportunidades de otimização decorrem do fato de que canais com uma relação de produção e consumo executam em tarefas distintas: o bloco funcional de *kernel* deslizante em uma tarefa não precisa ser executado apenas quando o mapa de atributos que lhe serve de entrada estiver completo, dado que apenas um pequeno grupo de valores, do mapa, será processado pelo *kernel* por vez. A tarefa produtora, por sua vez, também não precisa aguardar que todo o mapa de atributos a ser transmitido esteja completo, para iniciar a emissão de dados às tarefas que farão uso dele. A figura 22 ilustra essa ideia, para um kernel hipotético de dimensões 3x3.

Desta forma, transações de comunicação visando transferir parcelas de mapas de atributos entre tarefas podem ser feitas tão logo haja dados suficientes em *cache* para que se inicie o procedimento, o que auxilia no ocultamento da latência eventualmente imposta pelas transações de comunicação. A ideia é conseguir fazer com que as tarefas consumidoras não fiquem completamente ociosas enquanto aguardam dados para processar, buscando uma operação paralela que tenda a um *pipeline*. Nos valemos do fato de que, ao

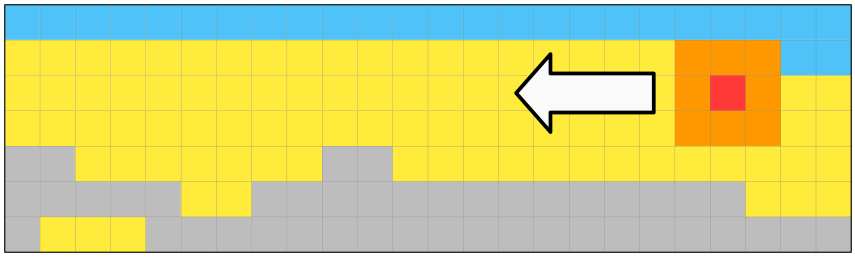


Figura 22: O *kernel* pode ser executado sobre um mapa de atributos ainda incompleto

menos em nosso ambiente de experimentação, métodos de emissão de mensagens não são bloqueantes e, por outro lado, a buferização em nível de NoC nos permite despachar dados de forma assíncrona às tarefas consumidoras. De fato, em experimentos preliminares implementando uma arquitetura arbitrária ilustrada na figura 23, e usando a abordagem de comunicação descrita, constatou-se o efeito prático, nos tempos tomados, em face de uma virtual redução de latência, inerente ao aproveitamento do tempo gasto nas transações de comunicação; iniciamos a aplicação do *kernel* sobre o mapa de atributos tão logo pudemos dispor de um número de linhas, do mapa, correspondente à altura do *kernel*, usando este fragmento do mapa como um tipo de trilho à execução do *kernel*, dando seguimento à operação tão logo houvesse mais dados que permitissem outro deslizamento horizontal, condicionado aos valores de *stride*. Em suma, vislumbra-se oportunidade para manter atividade de computação em tarefas com relação de dependência de dados, posto que a tarefa consumidora pode realizar trabalho sem ter ainda recebido todos os seus dados de entrada das tarefas produtoras.

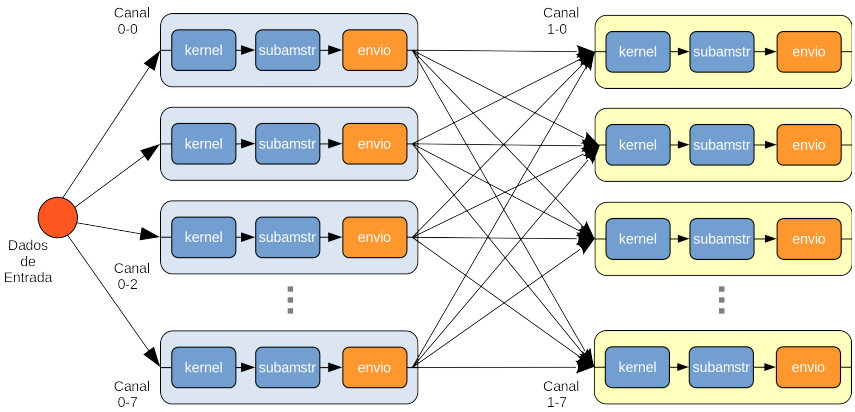


Figura 23: A arquitetura de referência do ensaio preliminar

A figura 24 mostra um diagrama de tempos ilustrando a forma como usamos a abordagem descrita em experimentos preliminares visando estudar como diminuir o tempo de execução minimizando o tempo de espera das tarefas consumidoras fazendo-as proces-

sar os lotes já disponíveis dos seus dados de entrada. No diagrama a abscissa é o tempo em *ticks* (pulsos de *clock*) e cada linha representa a atividade de computação associada a uma tarefa. Os nomes das tarefas estão indicados à esquerda, onde `fint_0_0_*` são tarefas atinentes a canais a uma primeira camada convolucional e sua camada de subamostragem subsequente e `fint_0_1_*`, a um segundo par de camada convolucional e posterior subamostragem. Neste diagrama, até o instante na marca 233794 *ticks*, temos a inicialização da aplicação, incluindo a distribuição das cargas do PEs. Depois disso, as partes marcadas na linha de tempo e nas raias associadas a cada tarefa indicam computação efetiva, ou seja, trabalho das CPUs onde as tarefas foram mapeadas. O tempo que não é computação é de espera, ociosidade que visamos minimizar. O digrama representa apenas o início do processo, porém permite notar que as tarefas `fint_0_1_*`, após trabalharem no recebimento de alguns dados, começam a trabalhar mais intensamente, antes que as tarefas `fint_0_0_*` encerrem a sua entrega. Isso é porque, de posse de parte dos mapas de atributos gerados pelas suas tarefas produtoras, as tarefas `fint_0_1_*` podem iniciar seu trabalho, executando *kernels* deslizantes, conforme a ideia que ilustrávamos na figura 22. Esta abordagem foi explorada em experimentos preliminares, mas não chegamos a usá-la em nosso experimento final, usando a arquitetura LeNET-5.

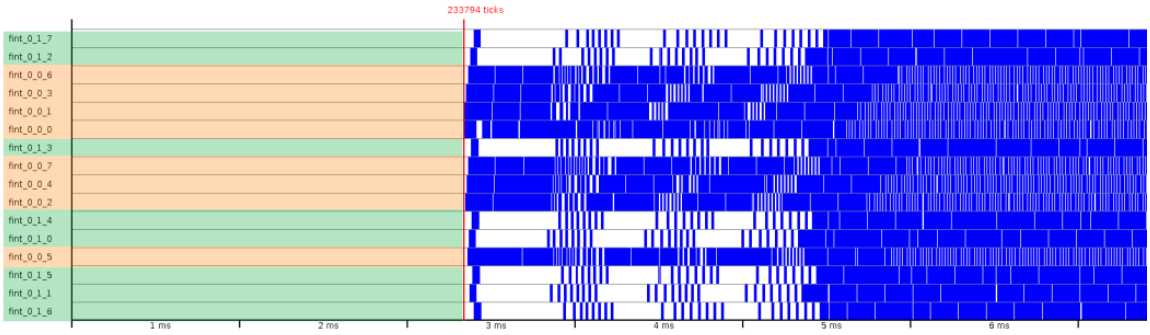


Figura 24: Diagrama de tempo inerente a um ensaio preliminar usando mapas de atributos incompletos

Desta maneira, vê-se que há vantagens em agregar alguns blocos funcionais em uma única tarefa desde que, e sobretudo, haja interdependência de dados entre blocos agregados, justificando a agregação, mas também observando o compromisso entre o crescimento do tempo de execução da tarefa e o comportamento do *overhead* de comunicação. Isso é importante porque a premissa de que paralelização sempre implica aceleração pode se mostrar falaciosa se o eventual crescimento do tempo despendido em transações de comunicação não for considerado. De outra maneira e em nosso contexto, a paralelização

de tarefas é tão bem-sucedida quanto menor for o tempo comprometido por esperas, por conta de transações de comunicação.

Com base nas lições adquiridas com os ensaios preliminares, foi-nos possível organizar os blocos funcionais encontrados em cada camada ou fase de processamento, na arquitetura de referência LeNET-5, nas tarefas notadas na tabela 5, a serem instanciadas em um número de vezes, no SoC, correspondente ao número de canais em cada camada da CNN. Cabe notar que um bloco de Emissão de Dados pressupõe a existência de um bloco de Aquisição de Dados na tarefa consumidora. Outro ponto importante é o número de tarefas instanciadas na SoC, 149 ao todo, onde as instâncias de C5 respondem pelo custo de 120 PEs. Essa é uma consideração importante porque, da forma como realizamos a implementação da arquitetura LeNET-5, cada instância implicou no consumo de um PE, fazendo desta organização um escopo óbvio de otimização futura. O número grande de PEs utilizados na implementação de C5 contra seu tempo relativamente curto de computação efetiva e, ainda, o *overhead* imposto de comunicação, em especial às tarefas C3S4 por precisarem enviar dados para muitas tarefas C5 com perda de *speed-up* de até 75% para C3S4, sugerem que deveríamos tentar agregar o trabalho destas instâncias em um número reduzido de tarefas, à semelhança do que fizemos com os Perceptrons de F6.

Tabela 5: As tarefas elicitadas agregando blocos funcionais na CNN

Nome da tarefa	Blocos Funcionais	Número previsto de instâncias
C1S2	*kernel* de C1, Subamostragem de S2, Emissão de Dados a C3S4	6
C3S4	*kernel* de C3, Subamostragem de S4, Emissão de Dados a C5	16
C5	*kernel* de C5, Emissão de Dados a F6	120
F6	Perceptrons de F6 (grupos de 14 unidades), Emissão de Dados a OUTPUT	6
OUTPUT	Perceptrons de OUTPUT	1

A organização final da aplicação ficou, então, como mostrado na figura 25. Os blocos funcionais, agregados em tarefas associadas às camadas da arquitetura de referência, já aglutinadas quando possível, executam serialmente no âmbito destas, e em paralelo, da perspectiva de camadas aglutinadas. Considera-se que uma otimização possível é explorar paralelismo no âmbito das tarefas, procurando organizar os blocos funcionais de maneira a operar como um *pipeline*.

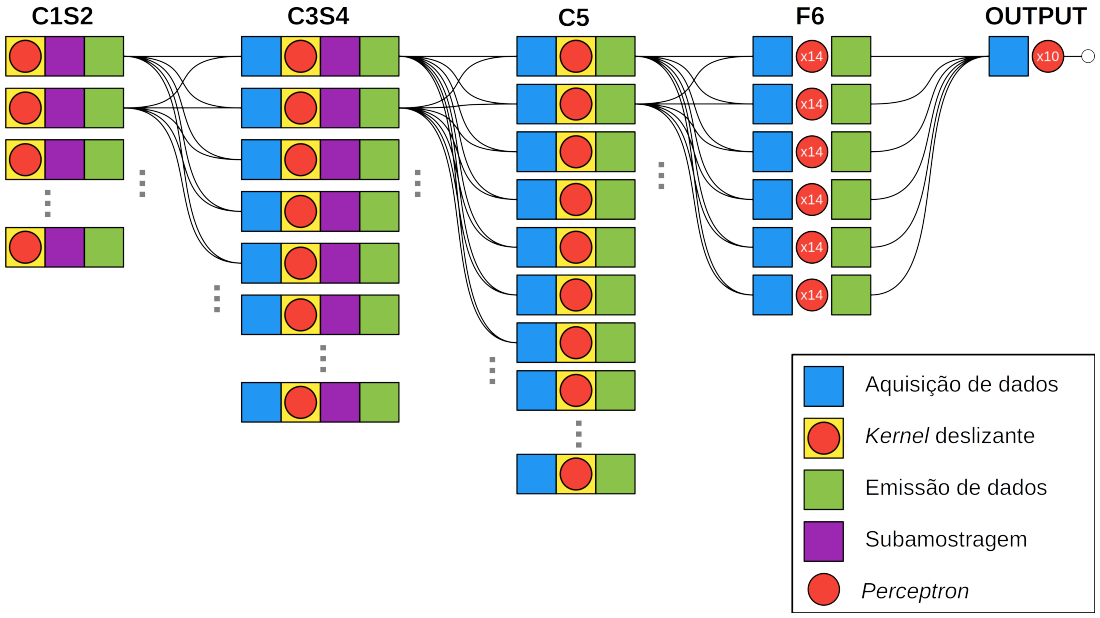


Figura 25: Organização da aplicação

4.3.2 Sobre fluxos de dados e comunicação entre tarefas

A API da MEMPHIS detém métodos para o envio e recebimento de mensagens entre tarefas de uma mesma aplicação, além de uma estrutura de dados continente para estas mensagens. Esse mecanismo de comunicação foi usado para transferir, entre tarefas, os valores dos mapas de atributos gerados como resultados do processamento dos dados recebidos e processados pelos canais de cada camada da CNN, despachando estes mapas aos canais na camada subsequente. Aproveitamos o *payload* subdividindo-o conforme diagramado na figura 26, apenas reservando os oito primeiros *bytes* para criarmos um tipo de cabeçalho de controle de uso no nível da aplicação.

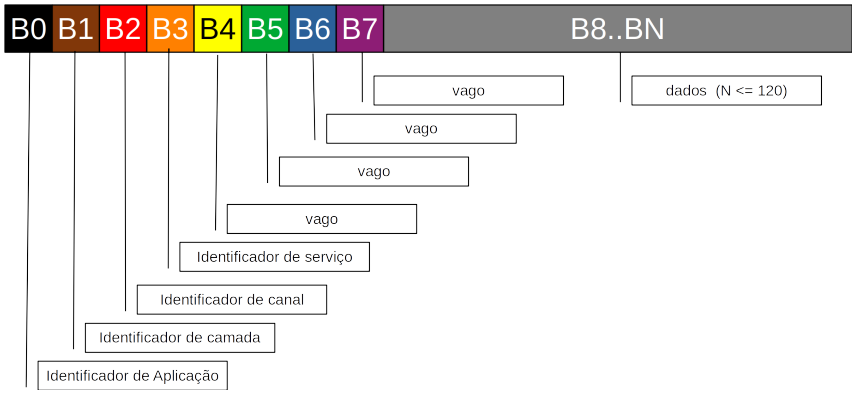


Figura 26: Aproveitamento do *payload* das mensagens pelas tarefas na aplicação

B0 foi originalmente reservado como um identificador de aplicação (que terminamos por não usar, posto que, na MEMPHIS, as aplicações constituem domínios fechados de co-

municação, não emitindo diretamente mensagens a outras, no SoC). B1, usamos como um identificador de camada e B2, um identificador de canal. Este campo foi usado primariamente em auxílio à organização de dados na implementação, posto que, na MEMPHIS, as mensagens são despachadas diretamente para a tarefa consumidora, sendo assim explicitamente conhecido *a priori* quais seriam as tarefas produtora e consumidora, na transação de comunicação. De outro modo, as transações de comunicação entre tarefas são ponto a ponto. B3 é um identificador de serviço, permitindo dizer à tarefa consumidora se o conteúdo da mensagem consiste em dados a processar ou se a mensagem seria para mera sinalização, o que usamos para indicar o término do envio de dados por parte da tarefa produtora. B4 a B7, não chegamos a utilizar. E o resto do *payload*, usamos para dados em si, cuja organização dependerá do par de tarefas implicado pela transação de comunicação mas sempre acomodando-o no vetor de inteiros iniciando em B8. Por *default*, na MEMPHIS, o *payload* de uma mensagem é um vetor unidimensional para até 128 números inteiros, codificados a 31 bits mais um bit de sinal.

Naturalmente, a primeira camada convolucional não recebe aporte de dados via mecanismos de comunicação da MEMPHIS porque as imagens de entrada, como explicávamos, são apresentadas à rede mediante o porte direto dos valores dos *pixels* ao código-fonte das tarefas atinentes àquela camada. Portanto, seu comportamento pressupõe que os dados de entrada sempre estarão disponíveis. Do mesmo modo, a camada de saída não despachará dados para ninguém, posto que os resultados de classificação são escritos diretamente em um arquivo de registro, novamente utilizando um método da API da MEMPHIS (a saber, o comando `Echo()`). A justificativa, em ambos os casos, para operarmos desta forma é que optamos por não implementar dispositivos de entrada e saída para o sistema emulado pela MEMPHIS, no decurso do projeto.

Para a implementação da arquitetura LeNET-5, as decisões tomadas em relação ao aproveitamento do *payload* das mensagens produzidas pelas tarefas atinentes a cada camada da CNN estão resumidas na tabela 6. Notar que, no caso de F6, as mensagens produzidas têm 14 valores porque a tarefa foi projetada para processar 14 Perceptrons, serialmente (dos 84 constantes da arquitetura LeNET-5, nesta camada). Também é por isso que o número de tarefas associadas à camada F6 é de seis instâncias. Similarmente, os dez Perceptrons previstos, na arquitetura LeNET-5, na camada OUTPUT são processados em uma única instância da tarefa atinente.

Tabela 6: Usos das mensagens pelas tarefas da aplicação

Tarefas produtoras	Tarefas consumidoras	*Payload* das mensagens	Quantidade de valores vs. Mensagens	Número de tarefas consumidoras
C1S2	C3S4	Uma linha de cada mapa de atributos	14 x 14	16
C3S4	C5	Um mapa de atributos	25 x 1	120
C5	F6	Um valor	1 x 1	6
F6	OUTPUT	Um valor por Perceptron	14 x 1	1

É oportuno mencionar que as transações de comunicação são realizadas por dois métodos da API da MEMPHIS: o método `Send()`, para o envio de uma mensagem a um receptor definido, e o método `Receive()`, que aguardará uma mensagem de um emissor definido, sendo esta uma operação bloqueante, ou seja, que interrompe a tarefa em execução até que uma mensagem seja recebida. Assim, e sob a premissa de que nem sempre todo o dado a ser utilizado por um bloco funcional ou tarefa consumidora caberá no *payload* de uma única mensagem, é às vezes necessário que esta tarefa consumidora opere a aquisição de dados em partes, até que todo o conteúdo que sua produtora tem a despachar-lhe tenha sido consumido, implicando dizer que os blocos funcionais de interfaceamento – emissão e aquisição de dados – sempre executarão simultaneamente. Um exemplo de produção e consumo de mensagem entre duas tarefas usando a API da MEMPHIS é mostrado na figura 27. Aqui, uma instância de C3S4, de nome `fint_1_3`, prepara-se para consumir uma mensagem, que espera ser produzida por uma instância de C1S2 chamada `fint_0_2`. A tarefa `fint_1_3` permanecerá bloqueada ao atingir a instrução `Receive()` até o recebimento de mensagem enviada por `fint_0_2`, o que é feito por meio da instrução `Send()`.

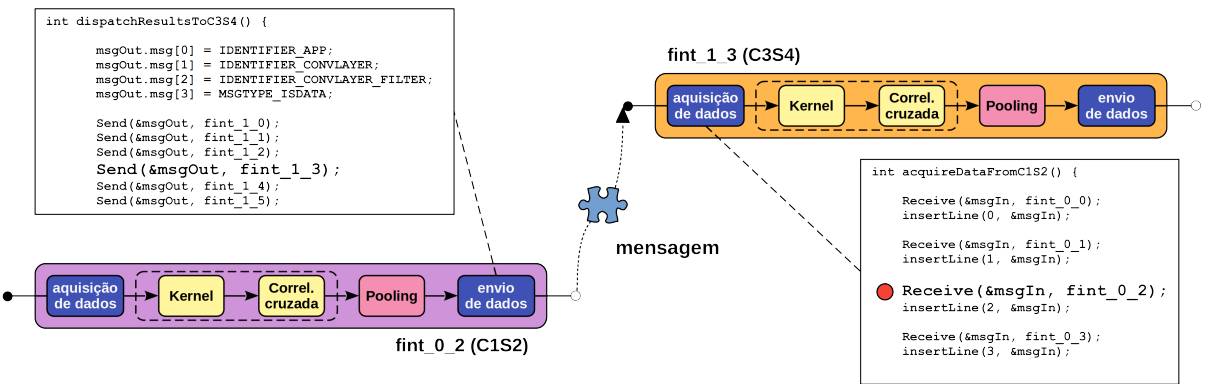


Figura 27: Exemplo de troca de mensagens entre tarefas usando a API da MEMPHIS

4.3.3 Sobre o número de tarefas por etapa

A rigor, se as tarefas agregam blocos funcionais que têm interdependências do ponto de vista de dados e, contudo, não as têm entre diferentes *instâncias* destes blocos presentes nos canais, então nossa primeira hipótese seria a de que poderíamos ter um único canal executando por tarefa, e esta seria a carga de trabalho a ser mapeada em cada PE utilizado na SoC. A premissa era a de que esta abordagem nos garantiria ótima exploração de paralelismo, resultando na maior aceleração possível – uma hipótese ingênua, posto que não consideramos o impacto do *overhead* de comunicação, além do próprio dispêndio de PEs, aos quais mapeamos uma tarefa por unidade, não reciclando-os ao longo do experimento. Explicando:

1. O tempo de execução atinente a cada tarefa e o consequente uso de tempo de processamento no PE precisam ser comparativamente maior do que o tempo despendido em transações de comunicação. De outra maneira, para que a distribuição de determinada carga de trabalho por diferentes recursos computacionais da SoC se mostre compensadora, é necessário fazer com que o tempo gasto em transações de comunicação não cresça de maneira a suplantar o ganho, em redução de tempo de execução, obtido a partir da distribuição desta carga de trabalho de forma a operá-la de forma paralela em diferentes PEs. Uma estratégia que experimentamos foi a agregação de trabalho de várias instâncias de certos blocos funcionais em uma tarefa, como o que fizemos em F6, onde cada tarefa processa 14 Perceptrons (contra nossa abordagem para C5, onde cada tarefa processa apenas um *kernel*).

2. A depender do número de operações que se deseja executar em paralelo, usar mais PEs tem implicação direta no custo em área do SoC, assim como prováveis impactos na mídia de comunicação posto que o roteamento de mensagens na NoC tende a se tornar mais trabalhoso e o tráfego de dados, mais intenso.

3. Especificamente sobre a MEMPHIS, há um aumento do tempo de simulação associado ao aumento das dimensões do SoC. Aumentar o número de PEs no SoC de uma ordem de grandeza elevou o tempo necessário à execução de uma simulação completa consideravelmente.

Uma consideração importante, ainda no concernente à agregação de trabalho por instância de tarefa, como mencionado no primeiro item acima, é a de que esta precisa

ser feita buscando contrapartidas explícitas no concernente à comunicação. Novamente mencionando F6, se uma instância da tarefa executará, serialmente, 14 Perceptrons da camada F6 da arquitetura LeNET-5, há que os dados de entrada despachados pelas 120 instâncias de C5 não precisariam estar distribuídos em 84 mensagens, uma para cada Perceptron de F6, mas em apenas 6 mensagens por instância de C5, procurando agregar dados para 14 Perceptrons por instância de F6 em cada mensagem de C5 a F6. Deste modo, adotamos como critério de projeto procurar agregar à carga de trabalho inerente a uma tarefa o quanto fosse possível em trabalho computacional, porém observando o compromisso disto com o dispêndio de tempo em operações de comunicação. Nas operações inerentes às camadas C5 e F6, o tempo associado a processamento efetivo, para um canal individual – em verdade, um Perceptron – é comparativamente menor que o tempo usado em comunicação. A estratégia para F6, como dissemos, foi agregar o processamento de 14 Perceptrons por instância da tarefa F6.

Na versão vigente do experimento, de fato, todas as tarefas estão instanciadas em número correspondente ao número de canais ou Perceptrons associados à camada da CNN relacionadas a estes, exceto F6 e OUTPUT, que agregam os Perceptrons das camadas homônimas em grupos de 14 (implicando seis instâncias) e 10 (implicando uma instância), respectivamente.

4.3.4 Construção de uma versão serial do experimento

Uma versão serial do experimento foi construída no intuito de permitir-nos examinar o tempo de execução associado a uma implementação que não explorasse paralelismo, bem como avaliar tratativas numéricas e o próprio funcionamento da CNN.

A versão serial é um derivado direto de sua equivalente paralela. De fato, a versão serial foi construída com base nesta outra, apenas removendo os blocos funcionais de aquisição e emissão de dados. Adaptações na estrutura dos arquivos de cabeçalho, no código fonte, utilizados para o porte dos parâmetros da CNN (a saber, os pesos), também foram feitas. A conexão de blocos funcionais foi realizada com o apoio de novas estruturas de dados, fazendo com que a troca de dados entre estes blocos, agora todos reunidos em uma única tarefa, se desse usando a memória local do PE que a executava.

Sobre o *testcase*, utilizamos o menor tamanho de SoC que conseguimos operar na MEMPHIS (a saber, uma malha 2x2). O tamanho das páginas de memória na DRAM

local dos PEs também foi aumentada, em parte para permitir a acomodação de todos os 60 mil parâmetros treináveis atinentes à LeNET-5. A representação destes parâmetros foi feita por um par de inteiros do tipo `long int` da linguagem C (32 bits) perfazendo um total de 480KB para seu armazenamento.

4.4 Resultados experimentais

Na MEMPHIS, existem duas formas de realizar o mapeamento de tarefas no SoC: definí-lo manualmente, adicionando à configuração de cenário o PE onde determinada tarefa executará, e utilizar um modo automático inerente à plataforma, onde a MEMPHIS mapeará as tarefas pelos PEs disponíveis no SoC autonomamente. Na versão paralela de nossa aplicação, para o mapeamento das instâncias das tarefas no SoC, fizemos experimentos preliminares usando o mapeamento manual, visando entender se o posicionamento das instâncias das tarefas influenciaria o tempo de execução do experimento. Entretanto, as diferenças de tempo que encontramos mapeando as tarefas de diferentes formas foram muito pequenas. De fato, posto que não existe dependência de dados entre as tarefas de um mesmo tipo e que estão executando em paralelo, só haverá trocas de mensagens no momento em que estas tarefas precisarem comunicar dados às tarefas subsequentes, considerando o fluxo de dados do modelo e explicando a diferença pequena nos tempos totais de execução associados aos diferentes mapeamentos manuais experimentados.

Percebemos, então, que não precisaríamos utilizar mapeamento manual de tarefas mediante a estrutura que consideramos dar à aplicação. Por isso, passamos a utilizar, somente, o sistema de mapeamento automático da própria plataforma, cujo algoritmo mapeia as instâncias das tarefas nos PEs segundo o nome destas instâncias, seguindo sua ordem alfabética e utilizando uma Busca em Diamante (*Diamond Search* - DS) (JAIN; JAIN, 1981). Por distâncias, nos referimos a distâncias de Manhattan (RIESZ, 1910), tomando o número de chaves de roteamento na rota das mensagens como métrica, ainda considerando que o algoritmo de roteamento é o XY.

A figura 28 mostra um mapa gerado por uma ferramenta que acompanha a MEMPHIS onde se vê as tarefas mapeadas em PEs no SoC. Os PEs são nomeados de acordo com a posição da chave de interconexão a que se conectam, na NoC. O nome das tarefas instanciadas denota de quem a tarefa mapeada é instância, mais um identificador numérico associado àquela instância em particular. Assim, temos que `fint_0_*` são instâncias de

C1S2, `fint_1_*`, instâncias de C3S4, `fint_2_*` para instâncias de C5, `f6_*` para instâncias de F6 e, finalmente, `out_*` como instâncias de OUTPUT (esta última, com apenas uma instância). O `PE0x0` não é usado no mapeamento porque trata-se de um PE com funções administrativas no SoC, como orquestrar o mapeamento das tarefas no sistema. Os 19 PEs sem tarefas, no canto inferior direito da malha, não foram utilizados por nosso experimento[1].

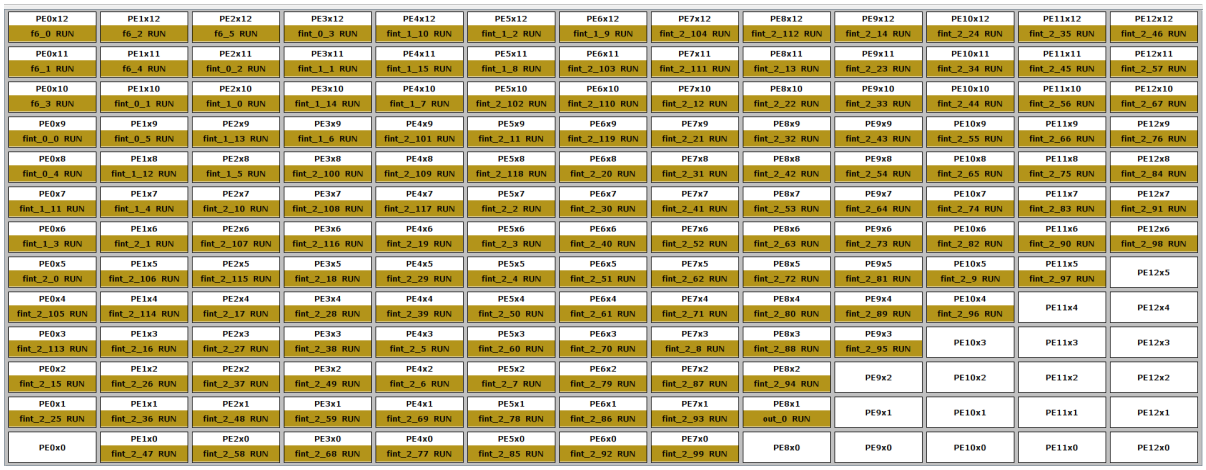


Figura 28: Tarefas mapeadas em PEs na MEMPHIS

Sobre a acurácia, os testes que realizamos utilizaram uma versão de nossa aplicação que ainda não dispõe de todas as tratativas numéricas necessárias a uma apropriada execução de uma CNN LeNET-5 na MEMPHIS, porém é capaz de ler um dado de entrada e emitir alguns resultados. A MEMPHIS, em sua distribuição padrão, não dá suporte a aritmética de ponto flutuante, sendo necessário implementar os expedientes necessários ao cômputo de números fracionários em nível de aplicação. Desta forma, nosso experimento pôde ser validado à luz de fluxos de dados, mas apresenta, em relação ao desempenho enquanto modelo preditivo, acurácia comparável a um sorteio aleatório considerando as dez classes associadas às amostras dos dez dígitos manuscritos do conjunto de dados de entrada (11,5%). Os testes foram feitos com um conjunto de 200 imagens do conjunto MNIST, como explicado na seção 4.2.

Sobre o tempo de execução, temos que a versão paralela do experimento executa em um total de 141,7ms, contra 314,6ms observados na execução da versão serial, perfazendo um *speed-up* de 2,22 vezes. Esse tempo contempla a execução integral do ex-

[1]Optamos por usar malhas quadradas. Neste caso, temos uma malha 13x13, perfazendo 169 nós, cada um deles conectando um PE. Nosso experimento, entretanto, usa 149 PEs apenas. O `PE0x0`, que não é usado para o processamento de tarefas da aplicação, não é contabilizado. Assim, há um excedente de 19 PEs.

perimento na MEMPHIS, do instante em que é inicializado até o momento em que todas as tarefas mapeadas no SoC estão encerradas e os PEs utilizados, liberados. O resultado nos é importante porque a versão paralela utilizada neste teste ainda não continha nenhuma melhoria específica, como das que tirassem proveito de ocultamento ou redução da latência decorrente das operações de comunicação, cujo *overhead* pode ser parcialmente mascarado se pudermos fazer computação durante a tramitação de mensagens entre tarefas, sugerindo que este valor de *speed-up* ainda pode ser melhorado. Outra razão cogitada para a obtenção de um *speed-up* tão baixo frente o grande consumo de PEs para computação paralela reside no fato de que, em C5, cada tarefa processa apenas um *kernel* deslizante. Isso não somente nos leva a usar 120 PEs para o processamento das instâncias de C5 como, ainda, impõe numerosas transações de comunicação por parte das instâncias de C3S4, além do uso empobrecido destes 120 PEs, cujo tempo efetivo em computação é muito próximo do consumido em transações de comunicação. Isso sugere que poderia haver melhor uso destes PEs usados no processamento de C5 se pudéssemos agregar, em uma mesma tarefa, o cômputo de mais de um *kernel*, à semelhança do conceito usado na implementação de F6 e OUTPUT, cujas instâncias operam vários Perceptrons.

A função primordial da versão serial é gerar valores que permitam avaliar ganhos de desempenho decorrentes da paralelização de partes do problema. Por outro lado, ela ajuda a explicitar oportunidades de aumento de desempenho procurando identificar as partes dos problemas que impõem maior dispêndio de tempo. Em nosso caso, a versão serial foi construída usando a organização que havíamos criado para a geração de tarefas, de forma a processar a rede. Isso nos permite fazer um comparativo entre as versões serial e paralela de nossa implementação considerando a agregação de blocos funcionais em tarefas tal como originalmente concebida para a versão paralela. Este comparativo é visto na tabela 7, onde os tempos médios despendidos com computação em ambas as versões são mostrados. Por custo, entendemos o número de PEs empregados na implementação das tarefas atinentes ao tipo enunciado, na versão paralela. É necessário entender também que os valores de tempo, para a versão paralela, referem-se apenas aos tempos despendidos com computação, desconsiderando os inerentes às operações de comunicação, discutidos adiante.

Constatamos uma relação aproximada de 1:1 entre o *speed-up* obtido na execução do problema decomposto em tarefas, na versão paralela, e o custo em PEs utilizados. É

Tabela 7: Tempos para computação por tarefa nas versões paralela e serial do experimento

Tarefa	Versão serial (ms)	Versão paralela (ms)	*Speed-up*	Custo (número de PEs)
C1S2	152,04	25,10	6,1	6
C3S4	85,37	5,31	16,1	16
C5	63,08	0,54	116,5	120
F6	12,82	2,41	5,3	6
OUTPUT	1,25	1,44	0,9	1

particularmente interessante notar o comportamento específico das tarefas C5 e F6 em nossa implementação. Em C5, optamos por usar um PE por filtro, implicando alto custo em recursos, porém com um rendimento de 116,5:120, ou 97% de aproveitamento do tempo usado em computação, nestes PEs, no concernente a *speed-up*. Já em F6, agregamos a computação de 14 Perceptrons por tarefa, implicando em aproveitamento de 5,3:6, ou 88% do tempo despendido em computação, nos PEs utilizados. Pondera-se que, no caso de F6, seções seriais da tarefa, como as necessárias à iteração dos Perceptrons por ela processada, possam ter influenciado negativamente o rendimento por PE, frente o observado em C5 e à luz do Argumento de Amdahl, que postula que o tempo mínimo necessário à execução de uma certa aplicação ou programa é limitado pelas seções não-paralelizáveis deste (SUN; CHEN, 2010). É importante lembrar que cada tarefa instanciada, de quaisquer dos tipos enumerados, foi mapeada em um PE exclusivo a si.

A discussão acerca dos tempos associados a computação e o *speed-up* consequente da paralelização precisa ser complementado considerando-se os tempos necessários às transações de comunicação. A tabela 8 mostra o tempo médio despendido por tarefa instanciada na emissão de resultados às suas respectivas consumidoras. É importante notar que, nesta implementação da versão paralela do experimento, as tarefas consumidoras permanecem em espera até o recebimento de todos os dados que utilizarão em seu processamento; de outra maneira, não realizarão computação enquanto suas tarefas produtoras estiverem transmitindo-lhes dados.

Nota-se o impacto das operações de comunicação no *speed-up*, em especial nas tarefas C3S4 e C5. As instâncias de C3S4 precisa transmitir dados a 120 instâncias de C5, o que acaba respondendo por 3/4 do tempo em que a instância não está em espera;

Tabela 8: Tempos despendidos pelas tarefas na versão paralela

Tarefa	Consumidoras	Computação apenas (ms)	Emissão de dados (ms)	Tempo total (ms)	Versão serial (ms)	*Speed-up* revisado
C1S2	16x C3S4	25,10	3,32	28,42	152,04	5,4
C3S4	120x C5	5,31	16,01	21,32	85,37	4,0
C5	6x F6	0,54	0,57	1,11	63,08	56,8
F6	1x OUTPUT	2,41	0,05	2,46	12,82	5,2

algo similar acontece com as instâncias de C5, que realizam computação por pouquíssimo tempo que, por outro lado, pareia com o tempo empregado em operações de emissão de dados. Estes resultados, quando confrontados com o consumo de PEs feito para cada grupo de tarefas, sugerem que, ao menos em relação a C5, a utilização de um PE por instância foi uma decisão pobre em custo-eficiência, prejudicando o *speed-up* obtido ao paralelizarmos C3S4 ao impor-lhe maior *overhead* de comunicação. A premissa é que as mensagens emitidas entre tarefas podem ter melhor aproveitamento do *payload* se as consumidoras agregam várias instâncias de uma mesma função, como as tarefas F6, onde cada uma das seis instâncias executará 14 dos 84 Perceptrons previstos na arquitetura LeNET-5, para esta camada.

Finalmente, examinamos o consumo médio de tempo imposto por cada função nas diferentes tarefas, tomando como base a implementação paralela. Estes tempos estão notados na tabela 9. É fácil perceber que os *kernels* são as funções mais demoradas, porém parecem ser menos dispendiosas em tempo à medida que avançamos pelas camadas convolucionais da arquitetura. Vale contrapor os tempos para os *kernels* de C1 e C5 para rapidamente percebermos que, a despeito de terem o mesmo tamanho e realizarem as mesmas operações algébricas, em C1 há um mapa de atributos de dimensões 28x28 a construir, enquanto C5 produz um mapa 1x1 (não é deslizante).

4.5 Oportunidades de melhoria

Neste ponto, é útil discorrer sobre oportunidades de melhoria que vislumbramos, dado o estado atual de nosso trabalho experimental. Nossa pretensão é contextualizá-las de forma a facilitar o fomento de trabalhos futuros.

Tabela 9: Tempos despendidos em funções diversas e em diferentes tarefas

Tarefa	Função	Tempo médio (ms)
C1S2	*Kernel* de C1	24,98
C1S2	Subamostragem de S2	0,10
C1S2	Emissão de Dados a C3S4	3,32
C3S4	*Kernel* de C3	5,26
C3S4	Subamostragem de S4	0,03
C3S4	Emissão de Dados a C5	16,01
C5	*Kernel* de C5	0,54
C5	Emissão de Dados a F6	0,57
F6	Perceptrons de F6	0,17
F6	Emissão de dados a OUTPUT	0,05
OUTPUT	Perceptrons de OUTPUT	0,13

4.5.1 Tipos numéricos e precisão na MEMPHIS

A plataforma de simulação MEMPHIS, em sua versão vigente (a saber, 1.2) e distribição padrão, não suporta, *a priori*, operações com números de ponto flutuante. Com isso, queremos dizer que, para se ter suporte a operações com tipos numéricos não inteiros sem o uso explícito de componentes que não são os originalmente distribuídos com a plataforma, é necessário provisionar, em software, funções e expedientes aritméticos que subsidiem a aplicação de tratativas numéricas no domínio dos números reais. Assim, e embora seja possível especificar como `float` o tipo numérico de variáveis usadas no código-fonte das tarefas e obtermos compilação aparentemente bem-sucedida com o compilador para sistemas-objetivo baseados na arquitetura MIPS, sua execução acarreta exceções na plataforma MEMPHIS em si, que interrompem o processamento do experimento. Fato é que, da forma como é originalmente distribuída (a que usamos em nosso trabalho experimental), a MEMPHIS efetivamente suporta números inteiros codificados por palavras de 32 bits, havendo a opção de operarmos com inteiros sinalizados, reservando um bit para sinal. Deste modo, as opções seriam: contornar essa dificuldade em nível de codificação das tarefas, implementando o suporte matemático necessário à sua execução, ou procurar utilizar CPUs (e seus compiladores) que já detivessem melhor suporte a outros tipos nu-

méricos. Nossa opção foi tentarmos a primeira alternativa, porque gostaríamos de tirar o melhor proveito possível da plataforma de simulação da forma como é originalmente distribuída.

Como as operações aritméticas envolvendo as transformações lineares necessárias ao processamento de uma CNN exigem a manipulação de valores não-inteiros, tivemos que adaptar nosso experimento de forma a conseguirmos portar estas operações para domínios numéricos de inteiros, unicamente. De fato, todos os parâmetros, treináveis e não treináveis, aplicados à nossa implementação eram, originalmente, números de ponto flutuante normalizados entre -1.0 e 1.0, de forma que só haveria como realizarmos uma tentativa de uso destes se pudéssemos transpô-los a um domínio numérico de inteiros e ali operá-los, deste modo.

Na tentativa de contornarmos esta dificuldade, foi criada uma abordagem baseada em sugestões presentes em (KNUTH, 1981). O autor pontua que, para sistemas que não suportam números de ponto flutuante, uma abordagem que poderia mostrar-se útil sem um comprometimento importante em precisão seria operar com os valores na forma de frações, observando os limites de representação para numeradores e denominadores. Cada valor originalmente fracionário seria, então, operado em código, nas tarefas, como um par de inteiros, um numerador e um denominador. É importante notar que, a despeito de operações de multiplicação e divisão de frações não representarem desafio importante desde que os limites de representação sejam observados, somas e subtrações demandariam lidar com cômputos de Menores Múltiplos Comuns (MMC). Cômputos mais complexos fatalmente demandariam abordagens emprestadas do campo da Análise Matemática, como a aplicação de séries de funções e outros expedientes.

Nossa abordagem inicial foi, portanto, notar todos os parâmetros treináveis da CNN para pares de inteiros, um numerador e um denominador. Como as transformações lineares efetuadas pela CNN, operadas com multiplicações de matrizes de valores de entrada por outras de pesos, implicariam somas de frações, cogitamos implementar rotinas de cômputo de MMC que seriam aplicadas em todas as operações de soma necessárias ao longo do processamento dos dados. A abordagem é ilustrada pela expressão 4, sendo V_e o valor de entrada, K um parâmetro definido em termos da razão entre $K_{numerador}$ e $K_{denominador}$, A um fator de ganho e V_r o valor resultante:

$$V_r = \frac{K_{numerador}}{K_{denominador}} \times A \times V_e. \quad (4)$$

4.5.2 Outras oportunidades de melhoria

Quatro simplificações, em relação à arquitetura LeNET-5 canônica, foram feitas em nossa implementação ao longo do processo de desenvolvimento. A primeira diz respeito às funções de subamostragem, onde a operação utilizada foi a MaxPooling, em lugar da média aritmética. A segunda, sobre as funções de ativação, onde utilizamos a ReLu em qualquer situação, em lugar das funções sigmoide e RBS. A terceira seria sobre os valores dos *pixels* de entrada, onde sua normalização a valores entre 0 e 1 seria uma operação preliminar comumente feita, mas a título de compatibilização com os pesos sinápticos utilizados no modelo, originalmente obtidos, em treinamento, mediante o uso da mesma abordagem. E a quarta diz respeito à conexão dos mapas de atributos gerados pela camada S2, na arquitetura LeNET-5, para a camada C3, em que nem todos os mapas são conectados entre todas as unidades de uma camada e outra, mas apenas um grupo de três ou quatro mapas por *kernel* em C3.

As três primeiras simplificações enunciadas foram adotadas como decisão preliminar para o contorno à dificuldade com a ausência de suporte a operações de ponto flutuante nativamente, na plataforma de simulação. Antevíamos seu impacto negativo sobre o desempenho de classificação do modelo, porém a estratégia permitiu-nos seguir com o trabalho de implementação do experimento, validando o fluxo de dados e habilitando o esforço de obtenção de aceleração de execução. A quarta simplificação diferencia-se das outras por sua natureza operacional em lugar de numérica, relacionando-se, sobretudo, à forma como os conjuntos de pesos sinápticos utilizados foram gerados – a saber, a partir de um modelo implementado utilizando o *framework* Keras, mencionado no trabalho relacionado (FRANÇA et al., 2021).

Finalmente, em face dos resultados de dispêndio de tempo, vê-se que há um relativo desbalanço no uso de PEs frente as contrapartidas em *speed-up*, algo que provavelmente pode ser revisto e otimizado. O melhor exemplo são as instâncias de C5: têm o custo de 120 PEs (um por instância) e executam por tempo comparativamente baixo frente às demais tarefas e ao próprio dispêndio de tempo em operações de comunicação, ainda impondo *overhead* de comunicação às instâncias de C3S4, que precisam enviar dados a

cada instância individual de C5. Reorganizar o experimento de forma a reunir o trabalho de tarefas como as C5 em menos instâncias pode contribuir com o aumento do *speed-up* local, se isso puder contrabalancear o aumento do *overhead* de comunicação que um número maior de instâncias impõe. Um menor dispêndio de PEs é uma vantagem óbvia e imediata.

4.6 Considerações finais

A solução de um problema computacional qualquer, quando a intenção é a busca por desempenho ótimo, passa por um processo iterativo de planejamento, implementação e testes que pode iniciar de um protótipo simples e destituído de otimizações. Isso porém implica subsequentes e constantes questionamentos visando identificar onde há recursos de computação ociosos, tarefas paralelizáveis e operações que poderiam ser simplificadas, trocadas ou eliminadas.

O exame do problema à luz do fluxo de dados é uma estratégia eficaz na identificação de oportunidades de melhoria e otimização. Em uma CNN, por exemplo, é possível tirar proveito do fato de que *kernels*, produtos internos associados e as subamostragens que se seguem podem ser organizados como um canal onde haverá um fluxo de dados operados sem dependências de outros, em uma mesma camada convolucional. Isso faz de uma CNN, em alguma medida, um problema embaraçosamente paralelo.

Por outro lado, o trabalho de elicitação de blocos funcionais atinentes à CNN e seu eventual reagrupamento na forma de tarefas permitiu-nos ver que há um caminho, em paralelização e na implementação de uma Rede Neural Convolucional em uma plataforma multiprocessada com uma NoC que preveja a buferização e roteamento de mensagens, para a emersão de um *pipeline*. Isso porque, uma vez que as funções de comunicação eventualmente permitem que o processamento no PE que executa a tarefa produtora pode seguir após o despacho de uma mensagem através do subsistema de comunicação, há chances de que se possa ocultar parcialmente a latência inerente a essas transações. Para tanto, haveria a premissa de que operações de envio de mensagens não poderiam bloquear a tarefa produtora, aguardando confirmação de recebimento pela tarefa consumidora, por exemplo. Em nossos experimentos observamos que uma tarefa em execução em um PE pode emitir mensagens a outras tarefas de sua aplicação e imediatamente seguir seu trabalho, se uma resposta imediata não for exigida. No caso de uma CNN, as tarefas

podem emitir mensagens contendo os dados necessários às tarefas subsequentes valendo-se de haver um fluxo de dados bem definido e unidirecional. Obviamente, considera-se que o contexto do meio de transmissão implica segurança e qualidade a ponto ser desnecessário à tarefa produtora receber confirmação de seu recebimento pelo destino.

Finalmente, é importante considerar aspectos laterais do problema, buscando clareza sobre o que é necessário implementar de modo a subsidiar a solução de todo o ferramental necessário à computação que lhe é necessária, como o suporte a operações matemáticas definidas em domínios numéricos reais. No caso de modelos em Aprendizado de Máquina, o suporte a tipos numéricos de ponto flutuante e aspectos variados de cálculo infinitesimal se tornam importantes e, quando não há suporte em *hardware* ou uma camada de *software* que provisione esse suporte *a priori*, essas tratativas precisam ser consideradas e implementadas ainda em fases preliminares do trabalho, com rigor no concernente à precisão e atenção a compromissos de desempenho, posto que serão basilares ao longo de todo o projeto.

Capítulo 5

CONCLUSÃO E TRABALHOS FUTUROS

ESTA dissertação é sobre a implementação de Redes Neurais Convolucionais em plataformas de *hardware* do tipo MPSoC, sistemas multiprocessados e integrados em um *chip*. A justificativa primeira desse trabalho é investigar aspectos de interesse na implementação de soluções em Aprendizado de Máquina usando *hardware* especializado, visando, sobremaneira, aceleração de execução. Procuramos sintetizar nossas conclusões e deixamos sugestões para trabalhos futuros.

5.1 Conclusão

Neste trabalho, abordamos aspectos inerentes à implementação de uma Rede Neural Convolucional (CNN) em um sistema embarcado cujos recursos são conectados por meio de uma Rede Intra-chip (NoC). Primeiramente, discutimos o tema Redes Neurais Convolucionais: o que são, como se diferenciam de Redes Neurais comuns, quais seriam suas aplicações mais frequentes e quais seus aspectos funcionais básicos. Discorremos sobre princípios básicos relacionados às NoCs, assim como sobre sua importância à indústria de semicondutores desde sua popularização em projetos de sistemas multiprocessados, em meados dos anos 2000. Em seguida, dissertamos sobre nossa investigação experimental, no intuito de implementar uma CNN baseada na arquitetura clássica LeNET-5: descrevemos a maneira como decompusemos a arquitetura em blocos funcionais básicos, explicamos a forma como os organizamos de maneira a lançar mão de computação paralela como forma de acelerar a execução da CNN, dissertamos sobre os detalhes da implementação dessa CNN e experimentos relacionados usando a plataforma de simulação MEMPHIS,

expusemos os resultados que conseguimos obter e comentamos rapidamente sobre o que aprendemos, no processo.

As Redes Intra-chip são um potente habilitador e facilitador na concepção de sistemas embarcados por serem soluções potenciais para problemas que, não-raro, se mostram limitadores importantes em projetos em Microeletrônica. Mesmo não sendo o escopo estrito deste trabalho, vale pontuar que, quando contrapostas a esquemas anteriores de interligação de módulos no *chip*, como barramentos, resolvem questões relacionadas à disputa por meios de transmissão, banda passante, dispêndio energético e outros fatores confinantes de desempenho. Em nosso trabalho experimental, serviu-nos a NoC como um simplificador de projeto pela forma como permitiu-nos abstrair de preocupações sobre a gestão da comunicação entre tarefas, além de ajudar, por conta de especificidades relacionadas à buferização de *flits* e a gestão de seu roteamento realizado pelas chaves de interconexão, na implementação de estratégias visando realizar ocultamento de latência, auxiliando-nos na busca por ganhos de desempenho em termos de tempo de execução. Aqui, nos referimos ao caráter ativo das chaves na NoC, armazenando *flits* e processando seu encaminhamento, liberando os PEs para realizar computação relacionada às tarefas em si.

Sobre o problema estudado, uma CNN, entendemos que sua implementação não somente é viável sem o suporte de uma larga coleção de camadas de *software* a subsidiá-la como, ainda, que há oportunidades tremendas na execução de modelos calcados em Redes Neurais em *hardware* especializado, sobretudo pela grande quantidade de operações algébricas que, ao mesmo tempo que simples, são de natureza que justifica considerar tais modelos como "embaraçosamente paralelos", justificando a investigação e o desenvolvimento de soluções em Aprendizado de Máquina calcadas em *hardware* especializado.

5.2 Trabalhos futuros

Vislumbramos, posto o nosso esforço em retrospectiva, ao menos três oportunidades para trabalhos futuros.

A primeira delas diz respeito à plataforma sobre a qual o experimento é construído. Posto que as CNNs, assim como outros modelos em Aprendizado de Máquina (desde que previamente treinados) envolvem numerosas, porém simples operações algébricas em sua maioria, incentiva-se a experimentação com plataformas de *hardware* reais, como as

eventualmente disponíveis na forma de *kits* de prototipação usando tecnologias diversas, a exemplo de *hardware* reconfigurável, como *chips* FPGA. Mesmo no âmbito de simuladores, como a MEMPHIS, é útil considerar experimentos complementares como, por exemplo, exames envolvendo mapeamento manual e medição de tempos de execução de maneira a investigar como a disposição das tarefas no SoC podem, potencialmente, influenciar o tempo de execução. Esse tipo de exame, entretanto, é útil se, em lugar de agregarmos os blocos funcionais com dependência de dados em uma única tarefa, a ser executada em um único recurso de computação, como um PE, distribuirmos esses blocos em tarefas diferentes, mapeadas em PEs diversos, modificando-os visando implementar um *pipeline*. Isso porque, na estrutura que demos a nosso experimento, transações de comunicação só se dão na transição entre grandes fases, o que o torna menos suscetível a variações no mapeamento das tarefas no SoC.

A segunda é sobre o potencial interesse associado ao uso de *hardware* especializado como forma de acelerar o processo de treinamento de modelos de Aprendizado de Máquina. Ocorre que é justamente esta a fase mais "custosa" em termos computacionais no desenvolvimento destes modelos, e procurar migrar este trabalho para *hardware* visando aceleração, assim como entendemos, constituiria contribuição relevante.

Finalmente, cogita-se que, deste trabalho, além de seu aperfeiçoamento, a via natural para dar-lhe continuidade seria a investigação e implementação de outros modelos de Aprendizado de Máquina no mesmo contexto que o nosso. Mesmo sob a classe de Redes Neurais Artificiais, há uma miríade de subclasses onde, tal como percebemos ao trabalhar com CNNs, encontrariam nesta proposta ressonância, pela afinidade que há entre o modelo estudado e estes outros. Por outro lado, há modelos de Aprendizado de Máquina concebidos para, especificamente, tirar o melhor proveito possível dos recursos de computação, talvez do *hardware* em si. Um exemplo imediato seriam as BNNs, cuja natureza adere imediatamente a esta premissa.

REFERÊNCIAS

ABBA, S.; LEE, J. A. Bio-inspired self-aware fault-tolerant routing protocol for network-on-chip architectures using Particle Swarm Optimization. *Microprocess. Microsyst.*, Elsevier B.V., v. 51, p. 1339–1351, 2017. ISSN 01419331.

ABDELOUAHAB, K. et al. Tactics to directly map cnn graphs on embedded fpgas. *IEEE Embedded Systems Letters*, IEEE, v. 9, n. 4, p. 113–116, 2017.

ADRIAHANTENAINA, A. et al. Spin: a scalable, packet switched, on-chip micro-network. In: IEEE. *2003 Design, Automation and Test in Europe Conference and Exhibition*. [S.l.], 2003. p. 70–73.

ALWANI, M. et al. Fused-layer cnn accelerators. In: IEEE. *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. [S.l.], 2016. p. 1–12.

ANDREASSON, D.; KUMAR, S. Slack-time aware routing in noc systems. In: IEEE. *2005 IEEE International Symposium on Circuits and Systems*. [S.l.], 2005. p. 2353–2356.

BENINI, L.; MICHELI, G. Networks on chip: A new paradigm for systems on chip design. *Proc. -Design, Autom. Test Eur. DATE*, p. 418–419, 2002. ISSN 15301591.

BJERREGAARD, T. *The MANGO clockless network-on-chip: Concepts and implementation*. [S.l.]: IMM, Informatik og Matematisk Modellering, Danmarks Tekniske Universitet, 2005.

BOBDA, C. et al. Dynoc: A dynamic infrastructure for communication in dynamically reconfugurable devices. In: IEEE. *International Conference on Field Programmable Logic and Applications, 2005.* [S.l.], 2005. p. 153–158.

CARARA, E.; CALAZANS, N.; MORAES, F. A new router architecture for high-performance intrachip networks. *J. Integr. Circuits Syst.*, v. 3, n. 1, p. 23–31, 2008. ISSN 18071953.

CARARA, E. A. et al. HeMPS - A framework for NoC-based MPSoC generation. *Proc. - IEEE Int. Symp. Circuits Syst.*, IEEE, p. 1345–1348, 2009. ISSN 02714310.

CHOI, W. et al. On-Chip Communication Network for Efficient Training of Deep Convolutional Networks on Heterogeneous Manycore Systems. *IEEE Trans. Comput.*, v. 67, n. 5, p. 672–686, 2018. ISSN 0018-9340. Disponível em: <https://ieeexplore.ieee.org/document/8119941/>.

CHOLLET, F. et al. Keras: The python deep learning library. *Astrophysics source code library*, p. ascl–1806, 2018.

COPPOLA, M. et al. Spidergon: a novel on-chip communication network. In: *2004 International Symposium on System-on-Chip, 2004. Proceedings.* [S.l.: s.n.], 2004. p. 15–.

COURBARIAUX, M. et al. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

DALLY, W. J. Virtual-Channel Flow Control. *IEEE Trans. Parallel Distrib. Syst.*, 1992. ISSN 10459219.

DALLY, W. J.; TOWLES, B. P. *Principles and practices of interconnection networks*. [S.l.]: Elsevier, 2004.

DEHYADGARI, M. et al. Evaluation of pseudo adaptive xy routing using an object oriented model for noc. In: IEEE. *2005 International Conference on Microelectronics*. [S.l.], 2005. p. 5–pp.

DENNARD, R. et al. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, v. 9, n. 5, p. 256–268, 1974.

DUATO, J.; YALAMANCHILI, S.; NI, L. *Interconnection Networks*: An engineering approach. [s.l.]: Elsevier textbooks, 2002. (The Morgan Kaufmann Series in Computer Architecture and Design).

FEIGE, U.; RAGHAVAN, P. Exact analysis of hot-potato routing. In: IEEE COMPUTER SOCIETY. *Proceedings., 33rd Annual Symposium on Foundations of Computer Science*. [S.l.], 1992. p. 553–562.

FRANÇA, A. B. Z. et al. Non-Memoryless vs. Memoryless Hardware Architectures for Convolutional Neural Networks. *2021 IEEE 12th Lat. Am. Symp. Circuits Syst. LASCAS 2021*, p. 21–24, 2021.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. *Deep Learning*. [S.l.]: MIT Press, 2016.

GOOSSENS, K.; DIELISSEN, J.; RADULESCU, A. Aethereal network on chip: concepts, architectures, and implementations. *IEEE Design and Test of Computers*, v. 22, n. 5, p. 414–421, 2005.

HEIDARI, S. et al. CAMDNN: Content-Aware Mapping of a Network of Deep Neural Networks on Edge MPSoCs. *IEEE Trans. Comput.*, IEEE, n. June, p. 1–12, 2022. ISSN 0018-9340.

HU, J.; MARCULESCU, R. Dyad: smart routing for networks-on-chip. In: *Proceedings of the 41st annual Design Automation Conference*. [S.l.: s.n.], 2004. p. 260–263.

ILATIKHAMENEH, H. et al. Saving Moore's Law Down to 1 nm Channels with Anisotropic Effective Mass. *Sci. Rep.*, v. 6, n. August, p. 1–6, 2016. ISSN 20452322.

JAIN, J.; JAIN, A. Displacement measurement and its application in interframe image coding. *IEEE Transactions on Communications*, v. 29, n. 12, p. 1799–1808, 1981.

KARINIEMI, H.; NURMI, J. Arbitration and routing schemes for on-chip packet networks. In: *Interconnect-centric design for advanced SoC and NoC*. [S.l.]: Springer, 2005. p. 253–282.

KIM, K. et al. An arbitration look-ahead scheme for reducing end-to-end latency in networks on chip. In: IEEE. *2005 IEEE International Symposium on Circuits and Systems (ISCAS)*. [S.l.], 2005. p. 2357–2360.

KINSTLER, L. *Finding Lena Forsen, the Patron Saint of JPEGs | WIRED*. 2019. Disponível em: <https://www.wired.com/story/finding-lena-the-patron-saint-of-jpegs/>.

KNUTH, D. E. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition*. [S.l.]: Addison-Wesley, 1981. ISBN 0-201-03822-6.

LECUN, Y. et al. Gradient-Based Learning Applied to Document Recognition. *Proc. IEEE*, v. 86, n. 11, p. 1–46, 1998. ISSN 00189219.

LECUN, Y.; CORTES, C.; BURGES, C. *The MNIST Database*. 1998. Disponível em: <http://yann.lecun.com/exdb/mnist/>.

LI, W. J.; RUAN, S. J.; YANG, D. S. Implementation of energy-efficient fast convolution algorithm for deep convolutional neural networks based on FPGA. *Electron. Lett.*, v. 56, n. 10, p. 485–488, 2020. ISSN 00135194.

LI, Y. et al. A gpu-outperforming fpga accelerator architecture for binary convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, ACM New York, NY, USA, v. 14, n. 2, p. 1–16, 2018.

MAJER, M. et al. Packet routing in dynamically changing networks on chip. In: IEEE. *19th IEEE International Parallel and Distributed Processing Symposium*. [S.l.], 2005. p. 8–pp.

MCCULLOCH, W. S.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, Springer, v. 5, n. 4, p. 115–133, 1943.

MILLBERG, M. et al. The Nostrum backbone - A communication protocol stack for networks on chip. *Proc. IEEE Int. Conf. VLSI Des.*, IEEE, v. 17, p. 693–696, 2004. ISSN 10639667.

MIRJALILI, S. The Ant Lion Optimizer. *Adv. Eng. Softw.*, v. 83, p. 80–98, 2015. ISSN 0965-9978. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0965997815000113>.

MITTAL, S. A survey of FPGA-based accelerators for convolutional neural networks. *Neural Comput. Appl.*, v. 32, n. 4, p. 1109–1139, 2020. ISSN 14333058.

MODARRESSI, M.; SARBAZI-AZAD, H. Topology Specialization for Networks-on-Chip in the Dark Silicon Era. In: *Adv. Comput.* [s.n.], 2018. p. 217–258. ISBN 9780128153581. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S0065245818300226>.

MORAES, F. et al. HERMES: An infrastructure for low area overhead packet-switching networks on chip. *Integr. VLSI J.*, v. 38, n. 1, p. 69–93, 2004. ISSN 01679260. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S0167926004000185>.

MOTAMEDI, M.; GYSEL, P.; GHIASI, S. PLACID: A platform for FPGA-based accelerator creation for DCNNs. *ACM Trans. Multimed. Comput. Commun. Appl.*, 2017. ISSN 15516865.

M.WANJARI, M.; AGRAWAL, P.; V. Kshirsagar, R. Design of NoC Router Architecture using VHDL. *Int. J. Comput. Appl.*, v. 115, n. 4, p. 18–21, 2015. ISSN 09758887. Disponível em: <http://research.ijcaonline.org/volume115/number4/pxc3902238.pdf>.

NABAVINEJAD, S. M. et al. An Overview of Efficient Interconnection Networks for Deep Neural Network Accelerators. *IEEE J. Emerg. Sel. Top. Circuits Syst.*, v. 10, n. 3, p. 268–282, 2020. ISSN 21563365.

OVTCHAROV, K. et al. Accelerating Deep Convolutional Neural Networks Using Specialized Hardware. *Microsoft Res. Whitepaper*, p. 3–6, 2015. Disponível em: <http://research-srv.microsoft.com/pubs/240715/CNN Whitepaper.pdf>.

PARK, J.; SUNG, W. Fpga based implementation of deep neural networks using on-chip memory only. In: IEEE. *2016 IEEE International conference on acoustics, speech and signal processing (ICASSP)*. [S.l.], 2016. p. 1011–1015.

PASRICHA, S.; DUTT, N. *On-Chip Communication Architectures*. [S.l.]: Elsevier, 2008.

PIRRETTI, M. et al. Fault tolerant algorithms for network-on-chip interconnect. In: IEEE. *IEEE computer society annual symposium on VLSI*. [S.l.], 2004. p. 46–51.

RHOADS, S. *Plasma - MIPS I compatible Processor*. 2001. Disponível em: <https://opencores.org/projects/plasma>.

RIESZ, F. Untersuchungen uber systeme integrierbarer funktionen. *Mathematische Annalen*, v. 69, n. 4, p. 449–497, Dec 1910. ISSN 1432-1807. Disponível em: <https://doi.org/10.1007/BF01457637>.

ROWEN, C. et al. A pipelined 32b nmos microprocessor. In: *1984 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*. [S.l.: s.n.], 1984. XXVII, p. 180–181.

RUARO, M. et al. Memphis: a framework for heterogeneous many-core SoCs generation and validation. *Des. Autom. Embed. Syst.*, Springer US, v. 23, n. 3-4, p. 103–122, 2019. ISSN 15728080. Disponível em: <https://doi.org/10.1007/s10617-019-09223-4>.

RUPP, K. et al. Years of microprocessor trend data. *Figure available on webpage http://www. karlrupp. net/wp-content/uploads/2015*, v. 6, p. 40, 2015.

RUSSELL, S.; NORVIG, P. *Artificial Intelligence: A Modern Approach*. 3. ed. [S.l.]: Prentice Hall, 2010.

SHAWAHNA, A.; SAIT, S. M.; EL-MALEH, A. FPGA-Based accelerators of deep learning networks for learning and classification: A review. *IEEE Access*, v. 7, p. 7823–7859, 2019. ISSN 21693536. Disponível em: <https://ieeexplore.ieee.org/document/8594633/>.

SILVA, I. N.; SPATTI, D. H.; FLAUZINO, R. A. *Redes Neurais Artificiais Para Engenharia e Ciências Aplicadas*. [S.l.: s.n.], 2010. ISBN 9788588098534.

STRUBELL, E.; GANESH, A.; MCCALLUM, A. Energy and Policy Considerations for Modern Deep Learning Research. *Proc. AAAI Conf. Artif. Intell.*, v. 34, n. 09, p. 13693–13696, apr 2020. ISSN 2374-3468. Disponível em: <http://arxiv.org/abs/1906.02243 https://ojs.aaai.org/index.php/AAAI/article/view/7123>.

SUN, X.-H.; CHEN, Y. Reevaluating amdahl's law in the multicore era. *Journal of Parallel and Distributed Computing*, v. 70, n. 2, p. 183–188, 2010. ISSN 0743-7315. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0743731509000884>.

SUSEELA, J.; MUTHUKUMAR, V. Loopback Virtual Channel Router Architecture for Network on Chip. In: *2012 Ninth Int. Conf. Inf. Technol. - New Gener.* IEEE, 2012. p. 534–539. ISBN 978-1-4673-0798-7. Disponível em: <http://ieeexplore.ieee.org/document/6209228/>.

SZE, V. et al. Hardware for machine learning: Challenges and opportunities. In: *2018 IEEE Cust. Integr. Circuits Conf.* [S.l.: s.n.], 2018. v. 2017-April, p. 1–8. ISBN 9781509051915. ISSN 08865930.

UMUROGLU, Y. et al. 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.: s.n.], 2017. p. 65–74.

VENKATARAMAN, N. L.; KUMAR, R. Design and analysis of application specific network on chip for reliable custom topology. *Comput. Networks*, Elsevier B.V., v. 158, p. 69–76, 2019. ISSN 13891286.

YANG, L. et al. Optimal Application Mapping and Scheduling for Network-on-Chips with Computation in STT-RAM Based Router. *IEEE Trans. Comput.*, v. 68, n. 8, p. 1174–1189, 2019. ISSN 0018-9340. Disponível em: <https://ieeexplore.ieee.org/document/8432053/>.

Apêndice A

Código-fonte

Listamos aqui partes do código-fonte do experimento que julgamos serem representativas em relação à forma como o construímos.

A.1 A versão serial do experimento com a arquitetura LeNET-5

A íntegra da implementação serial do experimento feito com a arquitetura de referência.

```
/*
    serial.c
    A serial version of the LeNET-5 experiment.
    Alexandre N. Cardoso, 2022.
*/

#include <api.h>

#include <stdlib.h>

#include <stdbool.h>

#include "./extra/inputdata.h"

#include "./extra/functions.c"

#include "./extra/serial_weights.h"

#define SIZE_OF_RESULTS_VECTOR 128 // from the size limit of the Message structure payload

#define C1_NUMBER_OF_FILTERS 6
#define C3_NUMBER_OF_FILTERS 16
#define C5_NUMBER_OF_FILTERS 120
#define F6_NUMBER_OF_PERCEPTRONS 84
#define OUTPUT_NUMBER_OF_PERCEPTRONS 10

#define C1_DIM_SQUARE_KERNEL 5
#define C1_DIM_SQUARE_PADDED_IMAGE 32
#define C1_DIM_SQUARE_PADDED_IMAGE_PADDING 2
#define C1_DIM_SQUARE_POOLING 2
#define C1_MUL_DENOMINATOR 1
#define C1_MUL_NUMERATOR 1000000
#define C1_POOLING_STRIDE 2
#define C1_RESULT_SIDE_SIZE 14

#define C3_DIM_INPUT_IMAGE_PADDING 2
#define C3_DIM_SQUARE_KERNEL 5
#define C3_DIM_SQUARE_PADDED_IMAGE 10
#define C3_DIM_SQUARE_POOLING_RESULT 5
#define C3_MUL_DENOMINATOR 1
#define C3_MUL_NUMERATOR 100000
#define C3_POOLING_STRIDE 2
#define C3_RESULT_SIDE_SIZE 5
#define C3_RESULT_SIZE 1

#define C5_DIM_SQUARE_KERNEL 5
#define C5_DIM_SQUARE_PADDED_IMAGE 5
#define C5_DIM_SQUARE_PADDED_IMAGE_PADDING 0
#define C5_MUL_DENOMINATOR 1
#define C5_MUL_NUMERATOR 10000

#define F6_MUL_DENOMINATOR 1
#define F6_MUL_NUMERATOR 1000

#define OUTPUT_MUL_DENOMINATOR 1
#define OUTPUT_MUL_NUMERATOR 100
```

```
int j = 0, k = 0, index_image = 0, lin = 0, col = 0;
unsigned int c1_value_from_activation = 0;

// Intermediary structures, to receive data
int c1_conv_result[C1_NUMBER_OF_FILTERS][C1_DIM_SQUARE_PADDED_IMAGE][C1_DIM_SQUARE_PADDED_IMAGE];
int c1_pooling_result[C1_NUMBER_OF_FILTERS]
  [(C1_DIM_SQUARE_PADDED_IMAGE - C1_DIM_SQUARE_PADDED_IMAGE_PADDING * 2) / C1_POOLING_STRIDE]
  [(C1_DIM_SQUARE_PADDED_IMAGE - C1_DIM_SQUARE_PADDED_IMAGE_PADDING * 2) / C1_POOLING_STRIDE];

int c3_conv_result[C3_NUMBER_OF_FILTERS][C3_DIM_SQUARE_PADDED_IMAGE * C3_DIM_SQUARE_PADDED_IMAGE];
int c3_pooling_result[C3_NUMBER_OF_FILTERS][C3_DIM_SQUARE_POOLING_RESULT][C3_DIM_SQUARE_POOLING_RESULT];

int c5_conv_result[C5_NUMBER_OF_FILTERS];

int f6_output[F6_NUMBER_OF_PERCEPTRONS];

int output_output[OUTPUT_NUMBER_OF_PERCEPTRONS];

// Functions.

int c1_kernel(int identifier_convlayer_filter) {
  int result = 0;
  for (lin = 0; lin < C1_DIM_SQUARE_PADDED_IMAGE - C1_DIM_SQUARE_KERNEL; lin++) {
    for (col = 0; col < C1_DIM_SQUARE_PADDED_IMAGE - C1_DIM_SQUARE_KERNEL; col++) {
      result = 0;
      for (j = 0; j < C1_DIM_SQUARE_KERNEL; j++) {
        for (k = 0; k < C1_DIM_SQUARE_KERNEL; k++) {
          index_image = (C1_DIM_SQUARE_PADDED_IMAGE * (lin + j)) + (col + k);
          result +=
            aSimpleIntegerDivision(serialized_input_data[index_image] * C1_MUL_NUMERATOR,
              1 * C1_MUL_DENOMINATOR,
              weights_C1[identifier_convlayer_filter][C1_DIM_SQUARE_KERNEL * j + k][0],
              weights_C1[identifier_convlayer_filter][C1_DIM_SQUARE_KERNEL * j + k][1]);
        }
      }
      c1_conv_result[identifier_convlayer_filter][lin][col] =
        aSimpleReLu(result + aSimpleIntegerDivision(1 * C1_MUL_NUMERATOR,
                                                    1 * C1_MUL_DENOMINATOR,
                                                    bias_C1[identifier_convlayer_filter][0],
                                                    bias_C1[identifier_convlayer_filter][1]));
    }
  }
  return 0;
}

int c3_kernel(int identifier_convlayer_filter) {
  int result = 0;
  // line, on input
  for (int lin = 0; lin < C3_DIM_SQUARE_PADDED_IMAGE - C3_DIM_SQUARE_KERNEL; lin++) {
    // column, on input
    for (int col = 0; col < C3_DIM_SQUARE_PADDED_IMAGE - C3_DIM_SQUARE_KERNEL; col++) {
      // line, on kernel
      for (int j = 0; j < C3_DIM_SQUARE_KERNEL; j++) {
        // column, on kernel
        for (int k = 0; k < C3_DIM_SQUARE_KERNEL; k++) {
          // number of channels on C0, which determines the depth of kernel and the convolution data
          for (int c = 0; c < C1_NUMBER_OF_FILTERS; c++) {
            result += aSimpleIntegerDivision(c1_pooling_result[c][lin + j][col + k] * C3_MUL_NUMERATOR,
              1 * C3_MUL_DENOMINATOR, * weights_C3[c][C3_DIM_SQUARE_KERNEL * j + k][0],
              * weights_C3[c][C3_DIM_SQUARE_KERNEL * j + k][1]);
          }
        }
      }
      c3_conv_result[identifier_convlayer_filter][C3_DIM_SQUARE_PADDED_IMAGE * lin + col] =
        aSimpleReLu(result + aSimpleIntegerDivision(1 * C3_MUL_NUMERATOR,
                                                    1 * C3_MUL_DENOMINATOR,
                                                    bias_C3[identifier_convlayer_filter][0],
                                                    bias_C3[identifier_convlayer_filter][1]));
    }
  }
  return 0;
}

int c5_kernel(int identifier_convlayer_filter) {
  int result = 0;
  // line, on input. In LeNet, the kernel will have the size of the input image.
  for (int lin = 0; lin <= C5_DIM_SQUARE_PADDED_IMAGE - C5_DIM_SQUARE_KERNEL; lin++) {
    // column, on input. In LeNet, the kernel will have the size of the input image.
    for (int col = 0; col <= C5_DIM_SQUARE_PADDED_IMAGE - C5_DIM_SQUARE_KERNEL; col++) {
      // line, on kernel
      for (int j = 0; j < C5_DIM_SQUARE_KERNEL; j++) {
        // column, on kernel
        for (int k = 0; k < C5_DIM_SQUARE_KERNEL; k++) {
          // number of channels on C3, which determines the depth of kernel and the convolution data
          for (int c = 0; c < C3_NUMBER_OF_FILTERS; c++) {
            result += aSimpleIntegerDivision(c3_pooling_result[c][lin + j][col + k] * C5_MUL_NUMERATOR,
              1 * C5_MUL_DENOMINATOR, * weights_C5[c][C5_DIM_SQUARE_KERNEL * j + k][0],
              * weights_C5[c][C5_DIM_SQUARE_KERNEL * j + k][1]);
          }
        }
      }
      c5_conv_result[identifier_convlayer_filter] = aSimpleReLu(result + aSimpleIntegerDivision(C5_MUL_NUMERATOR,
        C5_MUL_DENOMINATOR, bias_C5[identifier_convlayer_filter][0],
        bias_C5[identifier_convlayer_filter][1]));
    }
  }
  return 0;
}
```

```
int c1_maxpooling(int identifier_convlayer_filter) {
  int poolingresult_lin = 0;
  int poolingresult_col = 0;
  for (lin = C1_DIM_SQUARE_PADDED_IMAGE_PADDING;
       lin <= C1_DIM_SQUARE_PADDED_IMAGE - C1_DIM_SQUARE_PADDED_IMAGE_PADDING * 2;
       lin += C1_POOLING_STRIDE) {
    poolingresult_col = 0;
    for (col = C1_DIM_SQUARE_PADDED_IMAGE_PADDING;
         col <= C1_DIM_SQUARE_PADDED_IMAGE - C1_DIM_SQUARE_PADDED_IMAGE_PADDING * 2;
         col += C1_POOLING_STRIDE) {
      c1_value_from_activation = 0;
      for (j = 0; j < C1_POOLING_STRIDE; j++) {
        for (k = 0; k < C1_POOLING_STRIDE; k++) {
          if ( * c1_conv_result[lin + j][col + k] > c1_value_from_activation)
            c1_value_from_activation = * c1_conv_result[lin + j][col + k];
        }
      }
      c1_pooling_result[identifier_convlayer_filter][poolingresult_lin][poolingresult_col]
        = c1_value_from_activation;
      poolingresult_col++;
    }
    poolingresult_lin++;
  }
  return 0;
}

int c3_maxpooling(int identifier_convlayer_filter) {
  int value_from_activation = 0;
  int poolingresult_lin = 0;
  int poolingresult_col = 0;

  for (int lin = C3_DIM_INPUT_IMAGE_PADDING; lin <= C3_DIM_SQUARE_PADDED_IMAGE; lin += C3_POOLING_STRIDE) {
    poolingresult_col = 0;
    for (int col = C3_DIM_INPUT_IMAGE_PADDING; col <= C3_DIM_SQUARE_PADDED_IMAGE; col += C3_POOLING_STRIDE) {
      value_from_activation = 0;
      for (int j = 0; j < C3_POOLING_STRIDE; j++) {
        for (int k = 0; k < C3_POOLING_STRIDE; k++) {
          if (
              c3_conv_result[identifier_convlayer_filter]
              [C3_DIM_SQUARE_PADDED_IMAGE * (lin + j) + col + k] > value_from_activation)
            value_from_activation = c3_conv_result[identifier_convlayer_filter]
            [C3_DIM_SQUARE_PADDED_IMAGE * (lin + j) + col + k];
        }
      }
      c3_pooling_result[identifier_convlayer_filter][poolingresult_lin][poolingresult_col] = value_from_activation;
      poolingresult_col++;
    }
    poolingresult_lin++;
  }
  return 0;
}

int f6_perceptron(int perceptron_number) {
  int result = 0;
  for (int i = 0; i < C5_NUMBER_OF_FILTERS; i++) {
    result += aSimpleIntegerDivision(c5_conv_result[i] * F6_MUL_NUMERATOR,
                                     1 * F6_MUL_DENOMINATOR, weights_F6[perceptron_number][i][0],
                                     weights_F6[perceptron_number][i][1]);
  }
  result = aSimpleReLu(result + aSimpleIntegerDivision(1 * F6_MUL_NUMERATOR,
                                                       1 * F6_MUL_DENOMINATOR,
                                                       bias_F6[perceptron_number][0],
                                                       bias_F6[perceptron_number][1]));
  f6_output[perceptron_number] = result;
  return 0;
}

int output_perceptron(int perceptron_number) {
  int result = 0;
  for (int i = 0; i < F6_NUMBER_OF_PERCEPTRONS; i++) {
    result += aSimpleIntegerDivision(f6_output[i] * OUTPUT_MUL_NUMERATOR, 1 * OUTPUT_MUL_DENOMINATOR,
      weights_OUTPUT[perceptron_number][i][0], weights_OUTPUT[perceptron_number][i][1]);
  }
  result = aSimpleReLu(result + aSimpleIntegerDivision(1 * OUTPUT_MUL_NUMERATOR, 1 * OUTPUT_MUL_DENOMINATOR,
    bias_OUTPUT[perceptron_number][0], bias_OUTPUT[perceptron_number][1]));
  output_output[perceptron_number] = result;
  return result;
}

int c1s2() {
  Echo("C1S2.");
  Echo("SETTING-SAIL");

  for (int identifier_convlayer_filter = 0;
       identifier_convlayer_filter < C1_NUMBER_OF_FILTERS;
       identifier_convlayer_filter++) {
    Echo("C1");
    Echo(itoa(GetTick()));
    c1_kernel(identifier_convlayer_filter);
    Echo(itoa(GetTick()));

    Echo("S2");
    Echo(itoa(GetTick()));
    c1_maxpooling(identifier_convlayer_filter);
    Echo(itoa(GetTick()));
  }

  Echo("FINISHING-HERE");
  Echo(itoa(GetTick()));
```

```
  return 0;
}

int c3s4() {
  Echo("C3S4.");
  Echo("SETTING-SAIL");
  Echo(itoa(GetTick()));

  for (int identifier_convlayer_filter = 0;
       identifier_convlayer_filter < C3_NUMBER_OF_FILTERS;
       identifier_convlayer_filter++) {
    Echo("C3");
    Echo(itoa(GetTick()));
    c3_kernel(identifier_convlayer_filter);
    Echo(itoa(GetTick()));

    Echo("S4");
    Echo(itoa(GetTick()));
    c3_maxpooling(identifier_convlayer_filter);
    Echo(itoa(GetTick()));
  }

  Echo("FINISHING-HERE");
  Echo(itoa(GetTick()));
  return 0;
}

int c5() {
  Echo("C5.");
  Echo("SETTING-SAIL");
  Echo(itoa(GetTick()));

  for (int identifier_filter = 0; identifier_filter < C5_NUMBER_OF_FILTERS; identifier_filter++) {
    Echo("C5");
    Echo(itoa(GetTick()));
    c5_kernel(identifier_filter);
    Echo(itoa(GetTick()));
  }

  Echo("FINISHING-HERE");
  Echo(itoa(GetTick()));
  return 0;
}

int f6() {
  Echo("F6.");
  Echo("SETTING-SAIL");
  Echo(itoa(GetTick()));

  for (int identifier_perceptron = 0; identifier_perceptron < F6_NUMBER_OF_PERCEPTRONS; identifier_perceptron++) {
    Echo("F6");
    Echo(itoa(GetTick()));
    f6_perceptron(identifier_perceptron);
    Echo(itoa(GetTick()));
  }

  Echo("FINISHING-HERE");
  Echo(itoa(GetTick()));
  return 0;
}

int output() {
  int greatestResult;
  int result;
  int perceptronThatHasTheBestResult = 0;

  Echo("OUTPUT.");
  Echo("SETTING-SAIL");
  Echo(itoa(GetTick()));

  // initialize 'result' with the value from the first Perceptron.
  result = output_perceptron(0);
  greatestResult = result;

  for (int identifier_perceptron = 0;
       identifier_perceptron < OUTPUT_NUMBER_OF_PERCEPTRONS;
       identifier_perceptron++)
  {
    result = output_perceptron(identifier_perceptron);
    Echo(itoa(result));
    if (result > greatestResult) {
      greatestResult = result;
      perceptronThatHasTheBestResult = identifier_perceptron;
    }
  }

  Echo("FINISHING-HERE");
  Echo(itoa(GetTick()));
  return perceptronThatHasTheBestResult;
}

int main() {
  int classe = 0;

  Echo("STARTING-EVERYTHING");
  Echo(itoa(GetTick()));

  Echo("C1S2-START");
  Echo(itoa(GetTick()));
```

```
  c1s2();
  Echo("C1S2-END");
  Echo(itoa(GetTick()));

  Echo("C3S4-START");
  Echo(itoa(GetTick()));
  c3s4();
  Echo("C3S4-END");
  Echo(itoa(GetTick()));

  Echo("C5-START");
  Echo(itoa(GetTick()));
  c5();
  Echo("C5-END");
  Echo(itoa(GetTick()));

  Echo("F6-START");
  Echo(itoa(GetTick()));
  f6();
  Echo("F6-END");
  Echo(itoa(GetTick()));

  Echo("OUTPUT-START");
  Echo(itoa(GetTick()));
  classe = output();
  Echo("OUTPUT-END");
  Echo(itoa(GetTick()));

  Echo("Classe:");
  Echo(itoa(classe));

  Echo("FINISHING-EVERYTHING");
  Echo(itoa(GetTick()));
  exit();
}
```

A.2 Uma função de emissão de dados

Esta é uma função das tarefas C1S2 da versão paralela da experimento que prepara e emite dados para as instâncias de C3S4. Destaque para a função `Send()`, da API da MEMPHIS, que recebe como primeiro parâmetro um ponteiro para a estrutura de dados que contém o *payload* e alguns parâmetros da mensagem a enviar e, em seguida, o nome da tarefa que consumirá o dado.

```
int dispatchResultsToC3S4() {
  int poolingresult_lin = 0;
  int poolingresult_col = 0;
  msgOutAtivPool.msg[0] = IDENTIFIER_APP; // App Identifier
  msgOutAtivPool.msg[1] = IDENTIFIER_CONVLAYER; // Convolutional Layer Identifier
  msgOutAtivPool.msg[2] = IDENTIFIER_CONVLAYER_FILTER; // Convolutional Layer Filter Identifier
  msgOutAtivPool.msg[3] = MSGTYPE_ISDATA; // because the first message will contain data
  for (lin = 0; lin < (DIM_SQUARE_PADDED_IMAGE - DIM_SQUARE_PADDED_IMAGE_PADDING * 2) / POOLING_STRIDE; lin++) {
    msgOutAtivPool.msg[4] = poolingresult_lin; // the fifth byte will bring the result line number
    poolingresult_col = 0;
    for (col = 0; col < (DIM_SQUARE_PADDED_IMAGE - DIM_SQUARE_PADDED_IMAGE_PADDING * 2) / POOLING_STRIDE; col++) {
      msgOutAtivPool.msg[8 + poolingresult_col] = pooling_result[poolingresult_lin][poolingresult_col];
      poolingresult_col++;
    }
    Send( & msgOutAtivPool, fint_1_0 ); // send a line to the next stage
    Send( & msgOutAtivPool, fint_1_1 ); // send a line to the next stage
    Send( & msgOutAtivPool, fint_1_2 ); // send a line to the next stage
    Send( & msgOutAtivPool, fint_1_3 ); // send a line to the next stage
    Send( & msgOutAtivPool, fint_1_4 ); // send a line to the next stage
    Send( & msgOutAtivPool, fint_1_5 ); // send a line to the next stage
    Send( & msgOutAtivPool, fint_1_6 ); // send a line to the next stage
    Send( & msgOutAtivPool, fint_1_7 ); // send a line to the next stage
    Send( & msgOutAtivPool, fint_1_8 ); // send a line to the next stage
    Send( & msgOutAtivPool, fint_1_9 ); // send a line to the next stage
    Send( & msgOutAtivPool, fint_1_10 ); // send a line to the next stage
    Send( & msgOutAtivPool, fint_1_11 ); // send a line to the next stage
    Send( & msgOutAtivPool, fint_1_12 ); // send a line to the next stage
    Send( & msgOutAtivPool, fint_1_13 ); // send a line to the next stage
    Send( & msgOutAtivPool, fint_1_14 ); // send a line to the next stage
    Send( & msgOutAtivPool, fint_1_15 ); // send a line to the next stage
    poolingresult_lin++;
  }
  return 0;
}
```

A.3 Uma função de aquisição de dados

Estas são funções usadas nas instâncias das tarefas C3S4 da versão paralela do experimento para a aquisição de dados das instâncias de C2S2. Destaca-se aqui o uso da função `Receive()`, parte da API da MEMPHIS. O primeiro parâmetro é um ponteiro para a estrutura de dados onde os campos da mensagem que chega – incluindo seu *payload* – serão dispostos. O segundo parâmetro é o nome da tarefa produtora, da mensagem. A função `insertLine()` popula a estrutura de dados que servirá de entrada à computação a ser feita na instância de C3S4.

```
int insertLine(int c1s2_channel, Message * content) {
  for (int i = 0; i < C0_RESULT_SIDE_SIZE; i++) {
    // copies the payload to input_from_c0. The line reference comes from msg[4].
    input_from_c0[c1s2_channel].data[content -> msg[4]][i] = content -> msg[8 + i];
  }
  return 0;
}

int acquireDataFromC1S2() {
  for (int lin = 0; lin < C0_RESULT_SIDE_SIZE; lin++) {
    Receive( & msgInConv0, fint_0_0);
    insertLine(0, & msgInConv0);

    Receive( & msgInConv0, fint_0_1);
    insertLine(1, & msgInConv0);

    Receive( & msgInConv0, fint_0_2);
    insertLine(2, & msgInConv0);

    Receive( & msgInConv0, fint_0_3);
    insertLine(3, & msgInConv0);

    Receive( & msgInConv0, fint_0_4);
    insertLine(4, & msgInConv0);

    Receive( & msgInConv0, fint_0_5);
    insertLine(5, & msgInConv0);
  }
  return 0;
}
```