Universidade Estadual de Campinas
Instituto de Computação

Fabíola Martins Campos de Oliveira

Partitioning Convolutional Neural Networks for Inference on Constrained Internet-of-Things Devices

Particionamento de Redes Neurais de Convolução para Inferência em Dispositivos Restritos da Internet das Coisas

CAMPINAS
2020

Fabíola Martins Campos de Oliveira

Partitioning Convolutional Neural Networks for Inference on Constrained Internet-of-Things Devices

Particionamento de Redes Neurais de Convolução para Inferência em Dispositivos Restritos da Internet das Coisas

Tese apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Doutora em Ciência da Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Supervisor/Orientador: Prof. Dr. Edson Borin

Este exemplar corresponde à versão final da Tese defendida por Fabíola Martins Campos de Oliveira e orientada pelo Prof. Dr. Edson Borin.

CAMPINAS
2020

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

OL4p

Oliveira, Fabíola Martins Campos de, 1988-
Partitioning convolutional neural networks for inference on constrained Internet-of-things devices / Fabíola Martins Campos de Oliveira. – Campinas, SP : [s.n.], 2020.

Orientador: Edson Borin.
Tese (doutorado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Internet das coisas. 2. Redes neurais convolucionais. 3. Particionamento. 4. Computação distribuída. 5. Computação em névoa. 6. Inferência. I. Borin, Edson, 1979-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Particionamento de redes neurais de convolução para inferência em dispositivos restritos da Internet das coisas
Palavras-chave em inglês:
Internet of things
Convolutional neural networks
Partitioning
Distributed computing
Fog computing
Inference
Área de concentração: Ciência da Computação
Titulação: Doutora em Ciência da Computação
Banca examinadora:
Edson Borin [Orientador]
Kalinka Regina Lucas Jaquie Castelo Branco
Jó Ueyama
Edmundo Roberto Mauro Madeira
Sandra Eliza Fontes de Avila
Data de defesa: 17-07-2020
Programa de Pós-Graduação: Ciência da Computação

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0001-8531-4559
- Currículo Lattes do autor: http://lattes.cnpq.br/4049323565071061

Universidade Estadual de Campinas
Instituto de Computação

Fabíola Martins Campos de Oliveira

Partitioning Convolutional Neural Networks for Inference on Constrained Internet-of-Things Devices

Particionamento de Redes Neurais de Convolução para Inferência em Dispositivos Restritos da Internet das Coisas

Campinas, 17 de julho de 2020

Dedicated to my grandparents, Celso, Miriam, Francisco, and Laura, my parents, Eliel and Raquel, my brother Lucas, and my boyfriend, Helói.

"Computer Science is no more about computers than astronomy is about telescopes."
(Edsger W. Dijkstra)

Acknowledgments

First, I would like to express my gratitude to Prof. Dr. Edson Borin for his supervision, friendly guidance, support, and encouragement throughout this work.

I am forever indebted to my family and my boyfriend for their endless love, care, and encouragement in this journey and life.

I wish to acknowledge the High-Performance Computing Multidisciplinary Laboratory (LMCAD) and the Institute of Computing at the University of Campinas (IC - Unicamp) for providing the necessary infrastructure for my research.

I would also like to thank Prof. Dr. Maurício Breternitz Jr., Prof. Dr. João Paulo Papa, Prof. Dr. Luiz Fernando Bittencourt, Prof. Dr. Roberto Lotufo, Prof. Dr. Sandro Rigo, and Prof. Dr. Edmundo Madeira for their help and suggestions for my research.

I wish to acknowledge Prof. Dr. Kalinka Regina Lucas Jaquie Castelo Branco, Prof. Dr. Jó Ueyama, Prof. Dr. Edmundo Roberto Mauro Madeira, and Prof. Dr. Sandra Eliza Fontes de Avila to participate in the jury.

I also wish to acknowledge the professors and staff of the Institute of Computing for their help.

I would also like to thank all my friends at Unicamp for their understanding, help, and motivations throughout this work.

I wish to acknowledge the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001 and PROCAD (2966/2014), the Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq - 142235/2017-2 and 313012/2017-2, the Fundação de Amparo à Pesquisa do Estado de São Paulo - FAPESP - Centro de Pesquisa, Inovação e Difusão - CEPID - Center for Computing in Engineering & Sciences - CCES - grant 2013/08293-7, Microsoft, and Petrobras for their financial support.

Resumo

Bilhões de dispositivos comporão a Internet das Coisas (do inglês, *Internet of Things (IoT)*) nos próximos anos, gerando uma vasta quantidade de dados que necessitarão ser processados e interpretados eficientemente. A maioria dos dados é atualmente processada na nuvem, contudo, esse paradigma não pode ser adotado para processar a vasta quantidade de dados gerados pela IoT, principalmente devido a limites de largura de banda e requisitos de latência de muitas aplicações. Assim, pode-se usar a computação na borda para processar esses dados usando os próprios dispositivos. Neste contexto, técnicas de aprendizado profundo são adequadas para extrair informações desses dados, mas os requisitos de memória de redes neurais profundas podem impedir que até mesmo a inferência seja executada em um único dispositivo restrito em recursos. Além disso, os requisitos computacionais de redes neurais profundas podem produzir um tempo de execução inviável. Para habilitar a execução de modelos de redes neurais em dispositivos de IoT restritos em recursos, o código pode ser particionado e distribuído entre múltiplos dispositivos. Abordagens diferentes de particionamento são possíveis, no entanto, algumas delas reduzem a taxa de inferência à qual o sistema pode executar ou aumentam a quantidade de comunicação entre múltiplos dispositivos. Nesta tese, o objetivo é distribuir a execução da inferência de Redes Neurais de Convolução entre diversos dispositivos da IoT restritos. Três algoritmos de particionamento automático que modelam a rede neural profunda como um grafo de fluxo de dados e focam nas características de IoT foram propostos, usando funções-objetivo como maximização da taxa de inferências ou minimização de comunicação e considerando limites de memória como restrição. O primeiro algoritmo é o Particionamento baseado em Kernighan e Lin, cuja função-objetivo é minimizar a comunicação, respeitando as restrições de memória de cada dispositivo. O segundo algoritmo é o Particionamento de Redes Neurais Profundas para Dispositivos Restritos da IoT, que, adicionalmente ao primeiro algoritmo, pode maximizar a taxa de inferências da rede neural e também pode contabilizar apropriadamente a quantidade de memória requerida pelos parâmetros compartilhados e *biases* de Redes Neurais de Convolução. Finalmente, o terceiro algoritmo é o Particionamento Multinível de Redes Neurais Profundas para Dispositivos Restritos da IoT, um algoritmo que emprega a abordagem multinível para reduzir o tamanho do grafo e aproveitar as capacidades do algoritmo anterior. A principal contribuição desta tese é mostrar que se deve considerar o particionamento por neurônios ao particionar redes neurais profundas em dispositivos restritos da IoT. Comparada aos algoritmos na literatura, a redução de comunicação é geralmente a única função-objetivo oferecida e não há consideração de restrições de memória, permitindo que esses algoritmos produzam particionamentos inválidos. Além disso, os algoritmos propostos nesta tese, na maioria das vezes, produzem resultados melhores do que as abordagens na literatura. Finalmente, outra contribuição é que se podem utilizar os algoritmos propostos para particionar entre quaisquer dispositivos qualquer computação que possa ser expressa como um grafo de fluxo de dados.

Abstract

Billions of devices will compose the Internet of Things (IoT) in the next few years, generating a vast amount of data that will have to be processed and interpreted efficiently. Most data are currently processed on the cloud, however, this paradigm cannot be adopted to process the vast amount of data generated by the IoT, mainly due to bandwidth limits and latency requirements of many applications. Thus, we can use edge computing to process these data using the devices themselves. In this context, deep learning techniques are generally suitable to extract information from these data, but the memory requirements of deep neural networks may prevent even the inference from being executed on a single resource-constrained device. Furthermore, the computational requirements of deep neural networks may yield an unfeasible execution time. To enable the execution of neural network models on resource-constrained IoT devices, the code may be partitioned and distributed among multiple devices. Different partitioning approaches are possible, nonetheless, some of them reduce the inference rate at which the system can execute or increase the amount of communication that needs to be performed between the IoT devices. In this thesis, the objective is to distribute the inference execution of Convolutional Neural Networks to several constrained IoT devices. We proposed three automatic partitioning algorithms, which model the deep neural network as a dataflow graph and focus on the IoT features, using objective functions such as inference rate maximization or communication minimization and considering memory limitations as restrictions. The first algorithm is the Kernighan-and-Lin-based Partitioning, whose objective function is to minimize communication, respecting the memory restrictions of each device. The second algorithm is the Deep Neural Networks Partitioning for Constrained IoT Devices, which, additionally to the first algorithm, can maximize the neural network inference rate and can also account appropriately for the amount of memory required by the shared parameters and biases of Convolutional Neural Networks. Finally, the third algorithm is the Multilevel Deep Neural Networks Partitioning for Constrained IoT Devices, an algorithm that employs the multilevel approach to reduce the graph size and take advantage of the previous algorithm capabilities. The major contribution of this thesis is to show that we should consider the partitioning per neurons when partitioning deep neural networks into constrained IoT devices. When compared to the literature algorithms, communication reduction is usually the only offered objective function and there is no consideration of memory restrictions, allowing these algorithms to produce invalid partitionings. Additionally, our algorithms mostly produce better results than the approaches in the literature. Finally, another contribution is that we can use the proposed algorithms to partition into any kind of device any computation that can be expressed as a dataflow graph.

List of Figures

List of Tables

Abbreviations and Acronyms

2D Two-Dimensional 11, 42, 44, 45, 80, 82, 92, 93

3D Three-Dimensional 41

A Attribute 62

ACM Association for Computing Machinery 55

API Application Programming Interface 129

ARM Advanced Reduced Instruction Set Computer (RISC) Machine 36, 73, 78

B bytes 11, 13, 47–50, 52, 53, 73, 77, 88, 113, 114, 116, 118

CEML Complex Event Machine Learning 60, 62

CNN Convolutional Neural Network 9, 11, 12, 21, 22, 24, 26–29, 33, 39–43, 46, 47, 50, 51, 53, 55, 57, 60, 64, 65, 67, 68, 70, 73, 76, 85–87, 89, 98, 100, 101, 107, 109, 110, 113–115, 120–126, 128, 129

CoAP Constrained Application Protocol 34

CPU Central Processing Unit 55, 60, 61

D0 Class 0 33–35

D1 Class 1 33–35, 90

D2 Class 2 34, 35, 74, 90, 115

DAG Directed Acyclic Graph 39, 40, 47, 114, 126, 129

DIANNE Distributed Artificial Neural Networks for the Internet of Things 11, 25, 28, 60, 67, 75, 76, 78, 79, 83, 85, 87, 91, 97, 98, 100, 107, 124, 125

DN^{2}PCIoT Deep Neural Networks Partitioning for Constrained IoT Devices 9, 14, 22, 28, 83, 86–88, 95–100, 102, 103, 106, 107, 109, 110, 112–114, 116, 117, 119, 122, 125–129, 137

DNN Deep Neural Network 10, 11, 24–27, 39, 46, 49–51, 55–63, 70, 73, 83, 85–87, 89, 98, 107, 124–126, 128, 129

DSP Digital Signal Processor 24, 60

FLOP FLoating-point OPerations 34, 35, 47–50, 52, 87, 89, 114–116

FNN Feedforward Neural Network 29, 39, 40, 47

FPU Floating-Point Unit 36, 78

GPS Global Positioning System 29

GPU Graphics Processing Unit 24, 55, 59, 61

HTTP Hyper Transfer Protocol 34

IBM International Business Machines 30

IEEE Institute of Electrical and Electronics Engineers 55

ILSVRC ImageNet Large Scale Visual Recognition Challenge 40, 135

IoT Internet of Things 8–10, 12, 14, 21, 23–30, 32, 36, 37, 47, 51, 53–56, 58–62, 64, 65, 67, 68, 70, 73, 79, 80, 83, 85–87, 100, 107, 120, 121, 124–126, 128, 129

iRgreedy Inference-rate greedy approach 22, 93, 96

KiB 1 kibibyte = 1024 bytes 10, 34, 35, 64, 73, 75, 79, 90, 92, 118, 120, 121

KLP Kernighan-and-Lin-based Partitioning 9, 11, 12, 21, 28, 67–70, 74–76, 78–87, 125, 126, 137

LoRa Long-Range spread spectrum modulation technique 31, 129

LTE Long-Term Evolution 31, 129

MCU Microcontroller Unit 25, 36, 78

MDN^{2}PCIoT Multilevel Deep Neural Networks Partitioning for Constrained IoT Devices 9, 14, 22, 28, 109–120, 122, 123, 126–129, 137

MFLOP 1 mega floating-point operation = 10^6 floating-point operations 90

MiB 1 mebibyte = 1024 kibibytes 10, 35, 115, 116, 118

MLP Multilayer Perceptron 10, 38–42

O Operator 62

OAP Operator-Attribute-Parameter 57–59, 62

RAM Random Access Memory 33–35, 64, 73, 75, 90

RBM Restricted Boltzmann Machine 55

ResNet Residual Network 41, 107

RFID Radio-Frequency Identification 10, 29, 30, 32

RGB Red, Green, and Blue 43

RISC Reduced Instruction Set Computer 15, 36, 78

RNN Recurrent Neural Network 39, 55, 60

S Sample 58, 59, 62

SA Sample-Attribute 58, 59

SO Sample-Operator 58, 59

SOA Sample-Operator-Attribute 58, 59

SOAP Sample-Operator-Attribute-Parameter 57–59

SOP Sample-Operator-Parameter 58, 59

SVM Support Vector Machine 39

TLS Transport Layer Security 34

TPU Tensor Processing Unit 59

UDP User Datagram Protocol 34

VGGNet Visual Geometry Group Network 40, 107

WLAN Wireless Local Area Network 31

WSN Wireless Sensor Network 10, 29, 30

XML Extensible-Markup-Language 34

ZFNet Zeiler and Fergus Network 40

Symbols

$*$ Convolution operation 43, 44

∞ Infinity 43, 49

$\forall$ Universal quantification 49, 51, 53

$\in$ Set membership 49, 51, 53

$\sum$ Summation 51

a Neuron activation value 38, 42, 48, 52

b Bias associated with a neuron 35, 38

CC Cross-Correlation function 44

$\mathbf{C}$ Device C 48–50, 52

Cn Convolution layer n 73

c Kernel function to the convolution operation 43

D_i Depth of the input layer of a convolution or pooling layer 45, 46

D_c Depth of a convolution layer 45

D_p Depth of a pooling layer 46

$\mathbf{D}$ Device D 48–50, 52

d Derivative function 43

diff Function that returns 1 if two elements are assigned to different partitions and 0 otherwise 51, 53

dev Device 48, 49, 53

$\mathbf{E}$ Number of edges 83, 106

e Element e 51

F Number of filters for one convolution layer 44–46

FCn Fully connected layer n 73

f Neuron activation function 38, 52

G_n Graph in the level n of the multilevel approach coarsening phase 12, 111

G 1 giga $= 10^9$ 35, 116

g An arbitrary function 42

H_i Height of the input layer of a convolution or pooling layer 45, 46

H_c Height of the output layer of a convolution layer 45

H_p Height of the output layer of a pooling layer 46

h Element h 51

I Input image to the cross-correlation operation 44

i Neuron at layer l 38

J Number of neurons at layer $l - 1$ 38

j Neuron of layer $l - 1$ 38

K Input kernel to the cross-correlation operation 44

k 1 kilo $= 10^3$ 115

kw Kernel width 44

L Number of layers in a neural network 38, 51, 92

l Neural network layer 38, 51

M Number of blocks that consist in a pooling layer after a convolution layer 46

M 1 mega $= 10^6$ 34, 35, 90, 116

m An arbitrary function 42

min Minimum function 48, 49, 53

m_n Memory required by neuron n 51

m_p Memory that partition p can provide 51

m_{sbp_l} Memory required by the shared parameters and biases of layer l 51

N Number of neurons in the neural network 51, 53

n Neuron 51, 53

O Asymptotic upper bound 83, 92, 106, 115

o Shift related to the convolution domain variable 43

P Number of partitions 51, 92

Pn Pooling layer n 73

partition Function that returns 1 if an element e is assigned to partition p and 0 otherwise 51, 53

p Number of devices in the partitioning 49, 51, 53

Q Number of fully connected layers before the last fully connected layer 46

q Column index of matrix CC 44

R_h Receptive field height 45, 46

R_w Receptive field width 45, 46

$\mathbf{R}$ Receptive field size 44–46

r Row index of matrix CC 44

s Convolution operation output 43

T Stride 45, 46

t Convolution operation domain 43

U Number of convolution layers 46

u Kernel height 44

$\mathbf{V}$ Number of vertices 83, 106, 115

W_i Width of the input layer of a convolution or pooling layer 45, 46

W_c Width of the output layer of a convolution layer 45

W_p Width of the output layer of a pooling layer 46

w Neuron weight 52

w_{ij} Weight of the connection between neuron i at layer l and neuron j at layer $l-1$ 38

x Input layer data 38, 42, 48, 52

y Input function to the convolution operation 43

Z Zero padding 45, 46

z Neuron 53

Contents

Chapter 1

Introduction

The Internet of Things (IoT) is a research and industrial paradigm in which physical devices are connected to the Internet [32], wired or wireless, and should automatically detect and react to the environment [121]. Smart cities, health care, transportation, tracking of environmental conditions, user activity monitoring and user tracking such as speech, movements, physical activities, fall detection, heart conditions, cognitive support, and life logging are some of the application domains which IoT may revolutionize by improving environmental knowledge and user experiences [32, 70, 121].

The number of works about the IoT is growing, causing the IoT to become popular in the research field [25, 26, 69, 131, 132]. At the same time, an increasing demand for more applications motivates the deployment of IoT solutions in industries [11, 110, 121, 132]. In the next few years, a burst in the number of IoT devices is expected [19, 77, 111]. To achieve the estimated billions of devices in the IoT, many of them will have to be constrained, for instance, in size and cost [88]. A constrained device presents limited hardware in comparison to the current devices connected to the Internet. Recently, a classification of constrained devices has been proposed, showing the increasing importance of them in the IoT [15]. These devices are constrained due to their embedded nature, size, cost, weight, power, and energy. Considering that these constraints impact on the amount of memory, computational and communication performance, and battery life, these resources must be properly employed to satisfy applications' requirements. The proposed classification not only differentiates more powerful IoT devices such as smartphones and single-board computers such as Raspberry Pi from constrained devices but also delimits the IoT scope, which neither includes servers nor desktop or notebook computers. We consider devices that belong to this classification of constrained devices in this work.

In the IoT scenario, an environment may have several IoT devices, some of which may spend most of the time idle or executing little work [121]. We can use these devices to help with the task processing of other devices. On the other hand, other devices may need to execute heavy applications to process their sensor data, for instance, cameras that may need to continuously record and analyze images in real time [25, 60]. The IoT devices may also be heterogeneous, raising the challenge of how to use the IoT resources intelligently.

IoT devices may contain many sensors and can generate a large amount of data per second. This amount may prevent the data from being sent to the cloud for processing due to the high and variable latency and limited bandwidth of current networks [80, 111]. Some devices do not even allow Internet connection all the time, for example, battery-operated devices with a low amount of energy, which prevents the continuous transmission of IoT-generated data [69]. Furthermore, some applications require real-time or near real-time responses, which preclude this remote dispatch and open the challenge of how to process these data to achieve (near) real-time actions. Finally, another question related to remote dispatching is privacy because a vast amount of sensitive data can be sent to the Internet.

These challenges entail the data to be completely or at least partially processed on the Internet edge such as routing switches, multiplexers, and gateways (routers and firewalls), in a paradigm called fog computing [17], or even on the devices themselves, in a paradigm called edge computing [73, 78]. These scenarios can be seen as a cloud that runs closer to the end devices or on the devices themselves [13]. By running the virtualization infrastructure on any of those devices, the connected devices of an environment may communicate in a peer-to-peer fashion in order to potentially cooperate among themselves to support services and applications independently [111]. Therefore, fog computing and edge computing may be a solution to the vast amount of IoT sensor data that must be processed but may not be sent to the cloud and to efficiently use the IoT resources.

As the data generated by IoT sensors are usually multimedia data such as images, videos, and audio data, to obtain valuable information from this vast amount of data, an application class that has been gaining the attention of both academia and industry is machine learning, specifically the branch of deep learning [10, 54, 59]. Deep learning techniques have already been successfully applied to analyze data generated by the sensors of IoT devices such as smartphones, Graphics Processing Units (GPUs), and smartphone Digital Signal Processors (DSPs) [58, 59, 61]. Some of the deep learning advantages related to IoT are that it can automatically extract features from the data and strongly benefits from large amounts of data [81]. Deep Neural Networks (DNNs) are one of the most popular techniques in deep learning and their application occurs in two phases: training and inference. In the training phase, data are used to find the most suitable parameters for the neural network while, in the inference phase, the trained parameters are used to analyze new input data. We focus on the inference phase execution in this work.

Deep learning techniques often present a high computational cost, especially the Convolutional Neural Network (CNN), which is one of the most successful neural networks [34, 41, 54, 63, 97, 100, 106]. Since the DNN training is more computationally intensive than the inference, previous works have employed more effort to optimize the training [85, 117, 124]. Even though the inference is less computationally intensive, it may still require a lot of memory and computations to be deployed on resource-constrained IoT devices. With a limited configuration and network infrastructure restrictions, the inference performance becomes important as well, due to real-time responses or inference-rate requirements that an application may have. Additionally, the size of the DNNs may not fit into constrained IoT devices, causing the inference execution on a single IoT device to be impossible.

Two approaches are commonly adopted to enable the execution of DNNs on resource-constrained devices. The first approach prunes the neural network model so that it requires fewer resources. The second approach partitions the neural network model and executes its calculations in a distributed way on multiple devices. In some works that employ the first approach, pruning a neural network results in accuracy loss [26, 35, 66]. On the other hand, several works can apply the first approach to reduce DNN requirements and enable its execution on limited devices without any accuracy loss [38, 39, 125]. It is important to notice that, even after pruning a DNN, its size and computational requirements may still prevent the DNN from being executed on a single constrained device. Therefore, our focus is on the second approach. In this scenario, the challenge of how to partition the neural network for distributed execution aiming to satisfy one or more requirements arises since this is an NP-complete problem [49].

The rationale behind the DNN partitioning is to assign as many calculations as one device can handle and, then, transfer the partially computed data to other devices, which perform the remaining calculations. We can partition a DNN according to the network capabilities, computational performance, and/or the amount of energy of the devices. Thus, the result is a distributed system in which each device performs part of the inference computation of a DNN [2, 25, 131]. The understanding of how a neural network can be partitioned is essential to this thesis. Figure 1.1 shows how a DNN can be distributed to execute its inference using multiple devices in the context of smart cities, which can be one application to our work. For instance, in smart cities, there may be lots of cameras collecting images that need to be processed. The cameras may have some amount of extra resources such as memory and computational performance and, thus, can calculate a part of the inference of DNNs. Nearby devices can calculate the rest of the inference of this DNN while they also perform their primary tasks, which may be the inference calculations of other neural networks. Thus, the devices communicate the necessary data to the other devices so that the inference can be performed.

We are concerned with scenarios in which constrained devices produce data and only constrained devices such as the ones containing Microcontroller Units (MCUs) and sensors, e.g., microphones and cameras, are available to process these data. Although devices equipped with cameras might not be constrained in some of their resources, we have to consider that only part of these resources is available for extra processing. After all, the devices have to execute their primary task in the first place. Additionally, in this work, we considered Wi-Fi wireless connections, i.e., the medium is shared and can benefit from reduced data transmission, however, we did not consider intermittence either mobility.

Some techniques have been employed to bring more efficiency to deep learning execution on IoT, optimizing DNNs for distributed execution [25, 26, 59]. However, these works are still limited as they do not partition the DNN appropriately. Additionally, these works do not offer subsidies to help programmers to distribute computation so that they respect conditions such as the amount of device memory and optimize objectives such as the inference rate or energy consumption. There are some machine learning and IoT frameworks such as TensorFlow [1], Distributed Artificial Neural Networks for the Internet of Things (DIANNE) [25], and DeepX [59] that offer the infrastructure to distribute the neural network execution to multiple devices. However, they require the user to manually partition the neural network and they limit the partitioning into a per-layer approach. The per-layer partitioning may prevent neural networks from being executed on devices

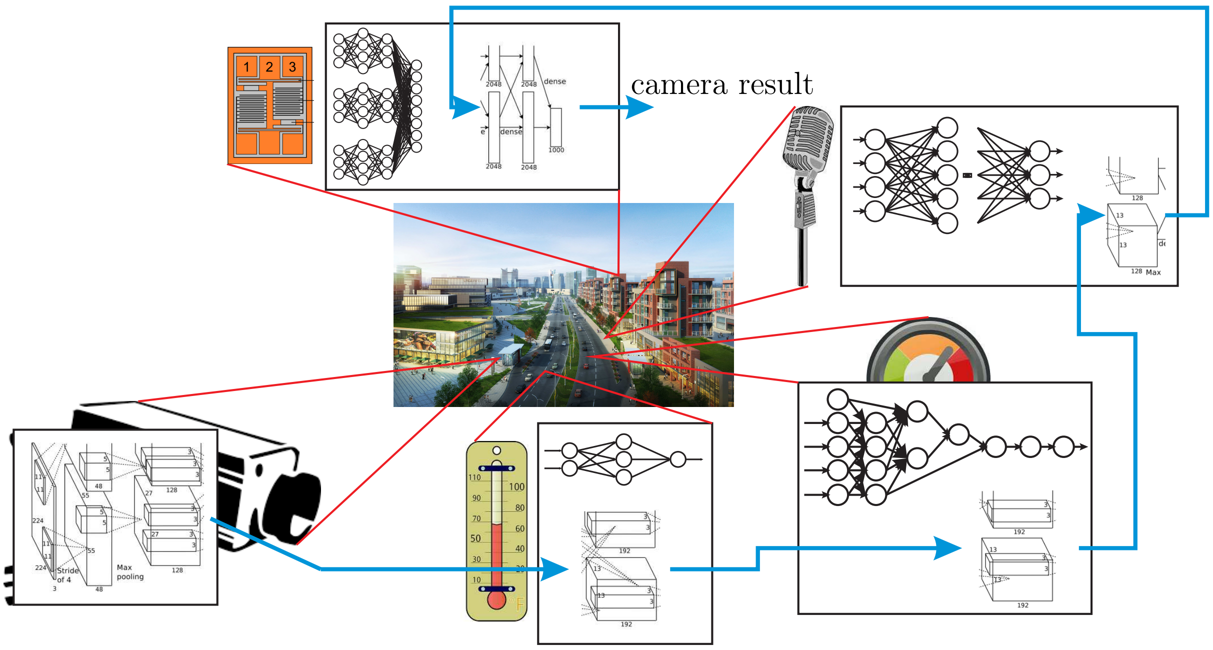


Figure 1.1: Example of how a DNN can be distributed to execute its inference in multiple devices in the context of smart cities. A camera may be collecting images that need to be processed. Cameras may have some amount of extra resources such as memory and computational performance and, thus, can calculate a part of the inference of DNNs. Nearby devices can calculate the rest of the inference of this DNN while they also perform their primary tasks, which may be the inference calculations of other neural networks. Thus, the devices communicate the necessary data to the other devices so that the inference can be performed. [98] (modified).

with more severe constraint conditions, for instance, some devices from the STM32 32-bit microcontroller family [102]. This may happen because there may be a single DNN layer whose memory requirements do not fit into the available memory of these constrained devices. On the other hand, although general-purpose frameworks such as SCOTCH [21] and METIS [49] present an automatic partitioning, they do not take into account the characteristics of neural networks and constrained devices. For this reason, they provide a suboptimal result or, in some cases, they cannot provide any valid partitioning, which are partitionings that have all the partitions respecting the amount of memory of each device.

1.1 Challenges, Objectives, and Contributions

The main challenges identified throughout the literature review are a proper consideration of the heterogeneous IoT devices' limited resources such as memory and computational performance; a proper consideration of the IoT networks' characteristics such as bandwidth, latency, and congestion; and an automatic partitioning of the CNNs execution into constrained devices.

The objective of this work is to treat the vast amount of data generated by the IoT devices by executing CNNs on the devices themselves, removing completely the dependency on the cloud. We consider these data as any kind of data that can be processed with CNNs such as images or audio data. For this purpose, we propose algorithms to automatically partition CNNs for efficient distributed execution of the inference on resource-constrained IoT devices, considering the network bandwidth and the devices' computational performance and memory. As we do not change the neural network architecture nor inference calculations, we maintain the quality of the result, i.e., we do not affect the neural network result. Thus, we propose approaches to obtain efficiency in communication and in the inference rate. Different from the partitioning strategies proposed on current machine learning frameworks, our algorithms can assign neural network neurons that belong to the same layer to different devices.

First, to address some of the challenges in the network infrastructure, we propose a new algorithm to partition CNNs among constrained IoT devices with the objective function of communication minimization. This algorithm always produces partitionings that respect the amount of memory of the devices. Furthermore, we consider the amount of memory of each device independently, allowing the algorithm to use devices with different amounts of memory.

To address the vast amount of data that can be generated by sensors, we are also concerned with the CNN inference rate. Based on the first algorithm and its results, we propose another algorithm that maximizes the inference rate or minimizes communication, according to the application. Additionally, for both objective functions, this new algorithm accounts more precisely for the amount of memory required by the shared parameters and biases of CNNs in each partition. This feature allows the algorithm to provide valid partitionings even when more constrained setups are employed in the applications. Other contributions resulting from the work with the second algorithm are case studies and the proposal of a simple greedy algorithm for communication reduction.

Finally, we propose an algorithm to decrease the second algorithm execution time. This algorithm employs techniques such as the multilevel approach, which automatically groups the CNN neurons, resulting in a smaller graph [49]. Other approaches include the smaller number of combinations that the algorithm tests and the smaller number of epochs that the algorithm executes for each subgraph of the multilevel approach. This algorithm contributes with 1) a different coarsening phase, 2) an always-valid initial partitioning for the smallest, most coarsen graph, which satisfies memory constraints since the algorithm beginning, and 3) a more flexible uncoarsening phase that executes either one epoch or all the epochs of the partitioning algorithm depending on the number of vertices of the subgraphs and the number of devices in the partitioning.

The main contribution of this thesis is to show that, when partitioning CNNs into constrained IoT devices to execute the inference, we should be able to assign neurons of the same layer to different partitions, instead of partitioning the CNN into its layers, which is a common approach for training. Other contributions of this thesis include the generalization of our algorithms, which can partition into any kind of devices any computation that can be expressed as a dataflow graph and the enabling of DNN developers to easily choose how to assign the CNNs neurons to IoT devices, expanding and, thus, facilitating the spread of intelligent sensors in the world. It is worth noting that we published the source code of all the algorithm implementations, the graphs for the CNNs and the setups, and the initial partitionings used in this research [74, 75].

1.2 Thesis Organization

This thesis is organized as follows.

- In Chapter 2, we present the main concepts related to this work: Internet of Things, resource-constrained devices, fog and edge computing, machine learning, deep learning, neural network models represented as dataflow graphs, partitioned neural networks, synchronization, and our problem definition.

- In Chapter 3, we discuss the related work in general frameworks for machine learning, frameworks for machine learning on IoT, and partitioning algorithms.

- In Chapter 4, we present the Kernighan-and-Lin-based Partitioning (KLP), our first algorithm to automatically partition neural networks for distributed execution among hardware-constrained IoT devices. We employ KLP to reduce communication among constrained IoT devices in the partitioning of the LeNet neural network model. We show that the partitionings provided by KLP require up to 4.5 times less communication than the partitionings offered by popular machine learning frameworks such as TensorFlow, DIANNE, and DeepX.

- In Chapter 5, we extend KLP and propose the Deep Neural Networks Partitioning for Constrained IoT Devices (DN^{2}PCIoT), an algorithm that not only reduces communication but also maximizes the inference rate of neural networks partitioned among constrained IoT devices. Additionally, this new algorithm accounts more precisely for the amount of memory required by the shared parameters and biases of CNNs in each partition. We compare DN^{2}PCIoT to several approaches and show that our algorithm can always produce valid partitionings and results that require less communication and/or more inferences per second than the other approaches.

- In Chapter 6, we propose the Multilevel Deep Neural Networks Partitioning for Constrained IoT Devices (MDN^{2}PCIoT), an algorithm that employs the multilevel approach to reduce the graph size and take advantage of the DN^{2}PCIoT capabilities. We also apply some techniques so that DN^{2}PCIoT executes faster in the multilevel approach. We validated this algorithm with the LeNet model for the communication reduction objective function and performed experiments with the AlexNet [54] model for the inference rate maximization and communication reduction objective functions.

- Finally, Chapter 7 summarizes the thesis work, states the main contributions of this research, discusses some limitations and difficulties, and presents recommendations for further research.

Chapter 2

Background

This chapter introduces the main concepts used in this research. First, we introduce the Internet of Things (IoT), constrained devices, and fog and edge computing. Next, we present some important concepts in machine learning and deep learning, including Feedforward Neural Networks (FNNs) and Convolutional Neural Networks (CNNs), indicating the state-of-the-art results and their suitability for processing sensor data. Then, we explain how a neural network can be represented as a dataflow graph and how this graph can be partitioned through two examples. Finally, we discuss synchronization and state the definition of our partitioning problem when considering communication reduction and inference rate maximization objective functions.

2.1 Internet of Things

The Internet of Things is a paradigm to connect different devices over networks and involves technologies such as Radio-Frequency Identification (RFID) to uniquely identify the devices in the network [9, 70]. The devices can be sensors, actuators, Global Positioning Systems (GPSs), smartphones, and any device that can directly or indirectly connect to the Internet, desktop computers and servers not included. These technologies, together with communication protocols, should be integrated into the IoT paradigm, causing devices from different vendors and with different communication protocols to interact with each other seamlessly. The network that connects them must be dynamic in order to allow, recognize, and process the connection and disconnection of devices [121].

Since the 1980s, the RFID technology has been used to provide device identification through a wireless medium, and, after 1990, Wireless Sensor Networks (WSNs) started being used to connect intelligent sensors, which acquire and process data [121]. Figure 2.1 shows this evolution of technologies which are the basis for the IoT. WSN applications include some types of monitoring such as animal monitoring [23] and traffic monitoring [121]. Different from the IoT, WSNs usually have homogeneous sensors with short-range communication, low bandwidth, low energy, low processing power, and small memory. These sensors should communicate their data to other sensors until reaching a gateway or router to transmit data forward. Both RFID and WSN technologies help to improve the IoT, combined with barcodes, smartphones, near-field communication, and cloud computing [121].

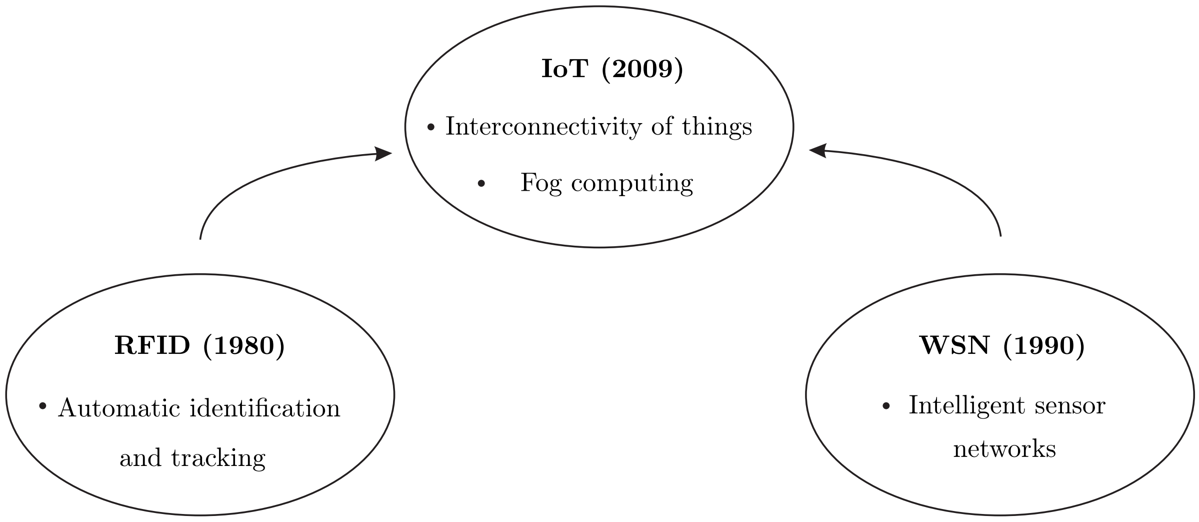

IoT (2009)
• Interconnectivity of things
• Fog computing
RFID (1980)
• Automatic identification and tracking
WSN (1990)
• Intelligent sensor networks

Figure 2.1: Evolution of technologies that form the IoT basis. The year represents when the respective technology started being used. Different from the IoT, WSNs usually have homogeneous sensors with short-range communication, low bandwidth, low energy, low processing power, and small memory. Both RFID and WSN technologies help to improve the IoT, combined with barcodes, smartphones, near-field communication, and cloud computing.

The IoT devices have physical attributes, which may form a distributed system connected to the Internet that performs cooperative computation to achieve the same objective [14]. Examples of areas that are increasingly incorporating IoT in their solutions are security surveillance, agriculture, environmental monitoring, smart cities, health care, and transportation [32, 121]. Figure 2.2 shows how smart structures can work in the context of smart cities. In this figure, smart buildings and smart structures contain sensors connected in a network. These sensors communicate the building state at relevant intervals and provide information about the building's movement in response to strong winds or earthquakes and about the structural health, temperature, vibration, and displacement. If any problem is encountered by the processing of these data, actuators in the building can absorb shocks or reduce its movements and a warning can be sent to other buildings or structures so that the people responsible for them take the necessary actions.

As more and more devices enter the Internet, their technology advancements - in wireless communication, smartphones, and sensors - are incorporated into the IoT, too. The reports from the International Business Machines (IBM) and the Information Technology and Innovation Foundation in 2009 advocate for IoT increasing productivity and innovation and improving information technology infrastructure [121]. With the ubiquity and the vast amount of IoT sensors, some design characteristics should be taken into account when considering cooperation among IoT devices to solve a problem:

- **latency** - time needed to send a message and process it;
- **throughput** - the maximum amount of data that can be sent through the network during some period;

Figure 2.2: Sensor communication in smart structures in the context of smart cities. Smart buildings and smart structures contain sensors connected in a network, which communicate the building state at relevant intervals and may provide information about the building's movement in response to strong winds or earthquakes and about the structural health, temperature, vibration, and displacement. If any problem is encountered by the processing of these data, actuators in the building can absorb shocks or reduce its movements and a warning can be sent to other buildings or structures so that the people responsible for them take the necessary actions [98] (modified).

- **scalability** - the number of supported devices in the system;
- **topology** - a subset of the available devices that each device are allowed to communicate;
- **energy** - battery life of battery-powered devices; and
- **safety** - possible points of failure.

Latency arises for applications that require (near) real-time responses. While we usually need low latency and high **throughput**, we need to determine the **scalability** of a system to estimate the number of supported devices in the system with reliability, as well as the runtime complexity and the necessity of buffering. The network type, which, in the case of Wireless Local Area Networks (WLANs), may have a transmission limit that is shared, also can be taken into account. Different communication technologies can be used such as 5G, Long-Range spread spectrum modulation technique (LoRa), ZigBee, Long-Term Evolution (LTE), or others. The **network topology** defines to which devices each device can communicate, considering that the devices may send and receive information to/from any other device or a subset of devices. **Energy consumption** may be a restriction for IoT devices that operate with batteries or that harvest limited amounts of energy from the environment such as solar, thermal, or kinetic energy or from passive RFID systems. For these devices, an execution that consumes low power is required so that it does not saturate the device's power supply. Finally, **safety** includes connection and disconnection of the devices and any other failures such as power or hardware failures.

Several IoT resources can be considered when designing an IoT solution to improve the quality of service. The main IoT issues include the challenges in the network infrastructure and the vast amount of data generated by the IoT devices, but other requirements such as security, dependability, and energy consumption are equally important [112]. Additionally, minimizing communication is important to reduce interference in the wireless medium and to reduce the power consumed by radio operations [114]. These issues and requirements usually demand a trade-off among the amount of memory, computational and communication performance, and battery life of the IoT devices. For instance, by raising the levels of security and dependability, offloading processing to the cloud, and/or processing data on the IoT devices, energy consumption is raised as well, impacting the device battery life.

In this thesis, we focus on three main characteristics of the IoT: communication network, device memory, and device processing power. These characteristics can be considered solely or combined, since a problem may have design requirements that involve one or more of these aspects. The communication network limits the data transfer rate and becomes a restriction when it limits the inference rate of a neural network, for example, when there is a data stream that needs to be processed. This problem may be aggravated when the transmission medium is shared, as is the case with wireless media, and may be a problem to latency as well. Device memory is a limitation to the problem size while the processing power may harm the execution time or result throughput requirement. Additionally, the processing power of each device may be heterogeneous and limited. We list the discussed IoT characteristics in Figure 2.3.

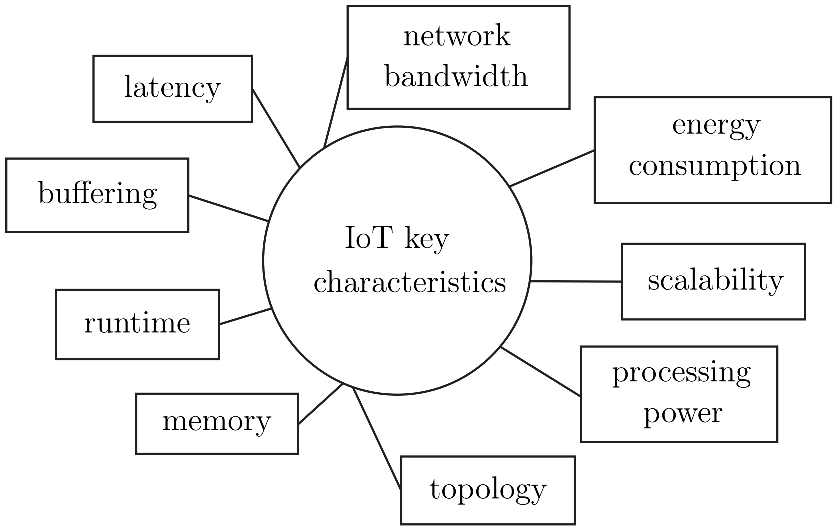


Figure 2.3: IoT key characteristics. These characteristics can be considered solely or combined, since a problem may have design requirements that involve one or more of these aspects.

2.2 Resource-Constrained Devices

Resource-constrained devices present one or more characteristics that are limited. As there is no consensus on the value that each characteristic should have so that the device is considered constrained, in this thesis, we employ the definition presented in the "Terminology for Constrained-Node Networks" report [15]. In this report, the authors define a constrained device as a device that does not present some of the characteristics that are common to other Internet nodes. These characteristics may be limited hardware, cost, electrical power, energy, and physical constraints such as size and weight. These limited characteristics impose a tight bound on the state, code space, and processing cycles, making energy optimization and network bandwidth usage fundamental in any design requirements. This definition is not rigorous, however, it separates resource-constrained devices from more powerful nodes such as server systems, desktop and laptop computers, smartphones, and single-board computers such as Raspberry Pi.

In this thesis, we focus on the size of available memory and the computational performance of the constrained devices. Thus, we employ the memory classification to experiment with devices from different classes. Figure 2.4 shows this classification for constrained devices, which may indicate the device capabilities. Even though the thresholds for each class reflect current technology and can change over time, Moore's law tends to be weaker for embedded devices when compared to personal computing devices. This is due to the fact that the increase in the transistor count and density are more likely to be invested in cost and power reduction than in computational performance increase. Additionally, we can combine devices from different classes and perhaps also nonconstrained devices to achieve the requirements of an application.

In Figure 2.4, we show the approximate magnitude order for data size, which may be the amount of Random Access Memory (RAM), for instance, and for code size, which may be the flash memory size, that each constrained device class can support. These constraints not only limit application and data sizes but also the communication protocols that each device can use. First, we detail each class of constrained devices and, after that, we show some examples of real-world devices that are categorized in each class, including the devices whose characteristics we used in this work.

Class 0 (D0) represents very constrained sensor devices, which include constraints in memory and computational performance. Due to these constraints, devices in D0 usually cannot securely connect to the Internet directly, depending on larger devices that act as proxies, gateways, or servers. These devices are usually preconfigured and rarely reconfigured, using a very small data set. Additionally, they can usually answer keepalive signals and send on/off or health indicators. It is worth noting that this class may contain constrained devices with different capabilities, i.e., the supported set of functions for each device may be different, according to the application types, protocols, and intended operation that the device can execute. We did not employ any characteristic of devices that belong to D0 because the smallest CNN that we used (LeNet) presents memory requirements that are not satisfied by the devices in this class. We show the required memory limits for partitioning CNNs in Section 2.6.

Class 1 (D1) devices are still quite a lot constrained in memory and computational per-

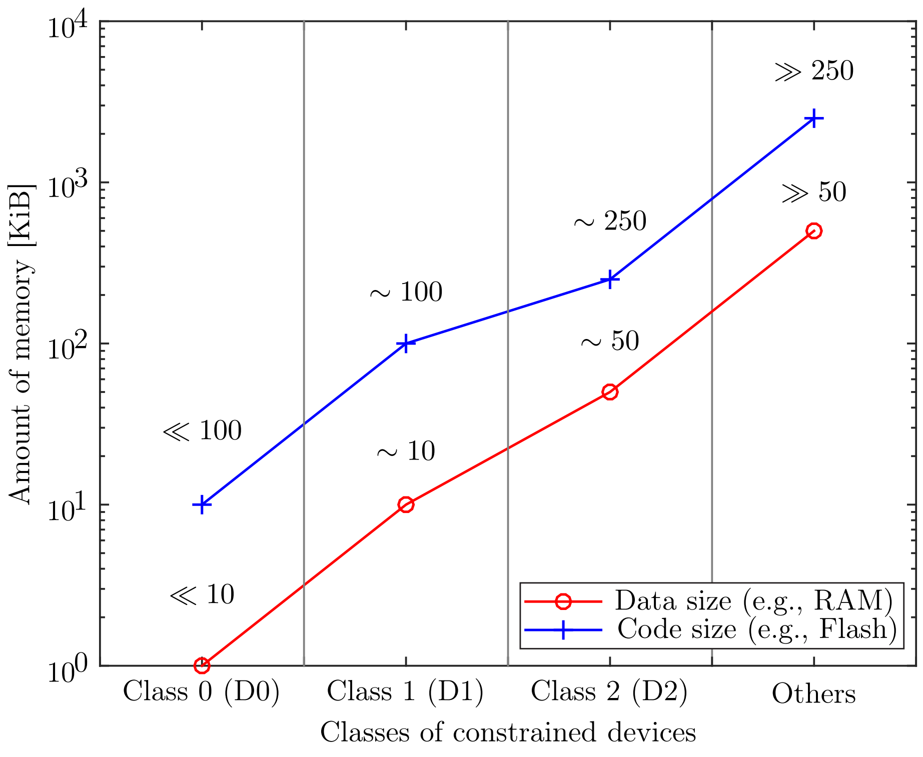


Figure 2.4: Classes of constrained devices concerning the amount of available memory (lower = more constrained). 1 kibibyte = 1024 bytes (KiB) [88] (modified).

formance, causing them to communicate to other Internet nodes with difficulty if they use a full protocol stack such as Hyper Transfer Protocol (HTTP), Transport Layer Security (TLS), and Extensible-Markup-Language (XML)-based data representations. Nevertheless, D1 devices can securely communicate without using gateways if they use protocol stacks for constrained devices such as the Constrained Application Protocol (CoAP) over User Datagram Protocol (UDP). Thus, although D1 devices can communicate with other Internet nodes, they need to save resources such as memory and energy. In this thesis, we used the characteristics of the Nest Learning Thermostat, which contains an STM32L151VB microprocessor that belongs to D1. The characteristics of this device were the most constrained characteristics that we used in this work, considering both the amount of memory that the device provides (16 KiB) and its estimated computational performance (1.6×10^6 FLoating-point OPerations (FLOP)/s).

Devices in Class 2 (D2) represents less constrained devices that can support most of the protocol stacks used by notebooks and servers. Despite that, these devices can employ fewer resources for communication if they use lightweight and/or energy-efficient protocols for more constrained devices. Additionally, D2 devices can optimize communication and use less bandwidth, saving more resources that can be spent on applications. If devices from D2 adopt the same protocols for D1 devices, interoperability may be increased and development costs may be reduced as well. In this class, we used the characteristics of the FitBit smartwatch, which contains an STM32L133VB microprocessor that provides 64 KiB of RAM and presents an estimated computational performance of 80 MFLOP/s.

Devices with fewer constraints than D2 are categorized as *others* because almost all of them can use existing protocols for communication without any changes. Devices in this category may still be constrained in memory, computational performance, and/or energy, limiting the applications that they can execute. Thus, in this thesis, we consider devices categorized as *others* also another class of constrained devices since its constrained resources are far from more powerful devices such as smartphones. In this class, memory use, computation, communication, and/or energy consumption should be optimized so that application requirements are satisfied. We used the characteristics of the Amazon Dash Button in this class, which contains a SAM G55J Atmel microprocessor that provides 176 KiB of RAM and presents an estimated computational performance of 120 MFLOP/s. We also used the characteristics of other devices that would be categorized as *others*, with the maximum amount of memory being 183 MiB and the maximum estimated computational performance, 312 GFLOP/s.

Figure 2.5 shows some examples of real-world devices for each category described in Figure 2.4. Table 2.1 shows each constrained device model's name according to the respective numbers in Figure 2.5. In each class of constrained devices, there is at least one device that is directly sold to end users, for instance, the development board Arduino Uno Rev3 in D0, Nest Learning Thermostat in D1, the FitBit smart bracelet in D2, and the TP-Link nano router in *Others*. The letters that are bold and red both in Figure 2.5 and in Table 2.1 correspond to some devices whose configuration was used in the experiments of this thesis.

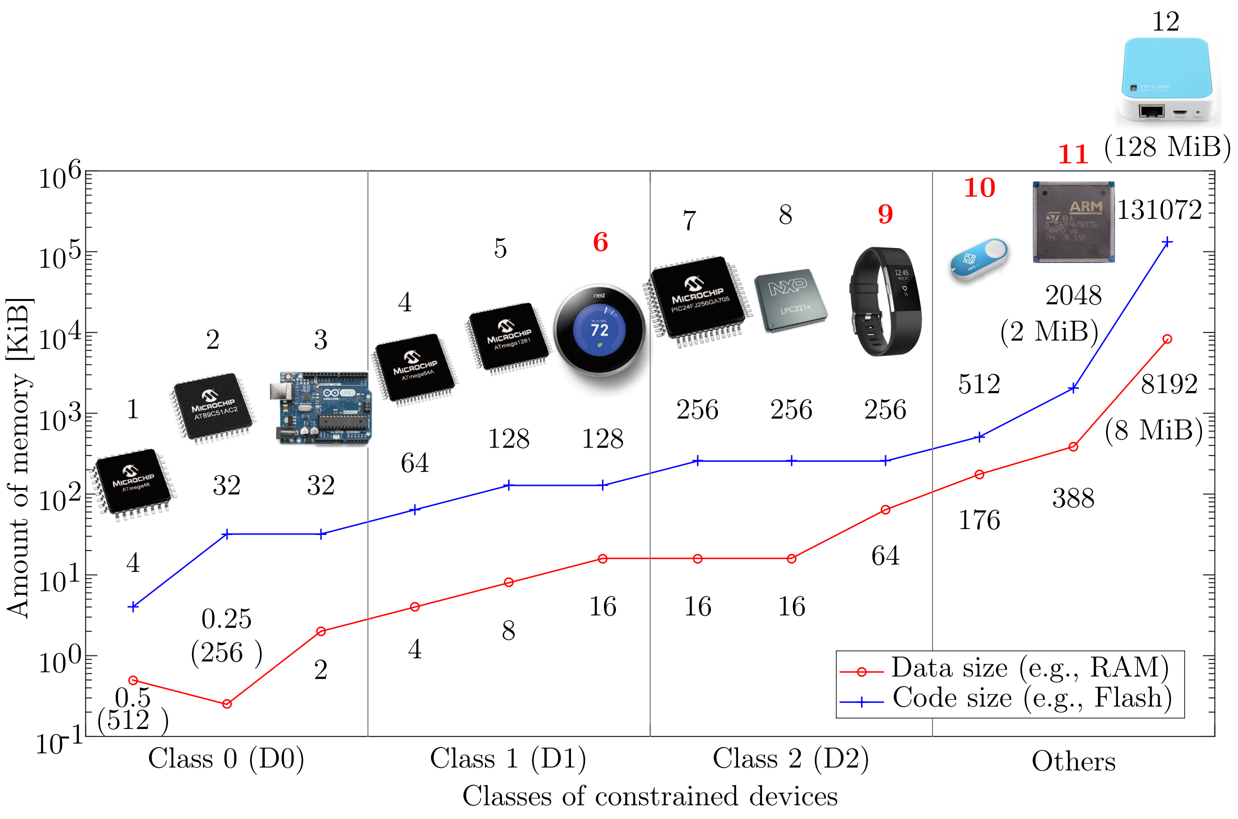


Figure 2.5: Examples of real-world devices for each class of constrained devices concerning the amount of available memory. The bold and red numbers indicate devices that were used in this work. 1 mebibyte = 1024 kibibytes (MiB) [88] (modified).

Table 2.1: Constrained device models used as examples of real-world devices in Figure 2.5. The bold names and numbers and the red numbers indicate devices that were used in this work [88] (modified).

	Constrained device model
1	ATmega48 8-bit AVR RISC-based microcontroller
2	AT89C51AC2 Atmel Enhanced 8-bit microcontroller
3	Atmega328P 8-bit microcontroller (Arduino Uno Rev3)
4	ATmega64A Microchip 8-bit AVR RISC-based microcontroller
5	ATmega1281 8-bit microcontroller (Waspmote)
6	**ST Microelectronics STM32L151VB 32-MHz ARM Cortex-M3 MCU (2nd-generation Nest Learning Thermostat)**
7	Microchip PIC24FJ256GA705
8	NXP 60-MHz ARM7TDMI-S Processors LPC221x
9	**STM32L433 ARM Cortex-M4 32-bit MCU+FPU (FitBit)**
10	**STM G55G / SAM G55J Atmel | SMART ARM-based Flash MCU (Amazon dash button)**
11	**STM32F469xx ARM Cortex-M4 32-bit MCU+FPU**
12	Atheros 560-MHz AR9344 (TP-Link nano router)

2.3 Fog and Edge Computing

The new IoT applications require a platform with characteristics that are different from the cloud. Fog computing can be defined as a cloud that operates closer to the end devices, extending the cloud computing paradigm to the network edge [13]. In fog computing, the virtualization infrastructure executes on the Internet edge, in devices such as routing switches, multiplexers, and gateways (routers and firewalls) [111]. Another paradigm related to fog computing is edge computing [73, 89, 127]. In edge computing, the virtualization infrastructure executes on the end devices, which usually generate data.

Applications executing on the fog and the edge are expected to present low latency, fulfilling requirements such as the ability to process streaming and send real-time responses [17, 73]. Fog and edge nodes are aware of their location, are widely and geographically distributed, and have support to communicate directly with mobile devices through mobility techniques. In the fog and the edge, there is a huge number of heterogeneous nodes, which mainly present wireless connections to the Internet or to other devices. With these characteristics, the fog and the edge become suitable for IoT applications such as smart vehicles, health care, and smart cities.

The fog and edge computing paradigms do not intend to substitute cloud computing but complement it instead, proposing intermediate computing levels between the cloud and the end user [17, 73]. The fog and the edge also enable new applications and services, contributing to the fields of data management and analytics. Although the fog, the edge, and the cloud employ the same resources such as network, computation, and storage, the same mechanisms such as virtualization, and are both multi-tenant approaches, the fog and the edge computing paradigms focus on applications and services in which the cloud presents difficulties in satisfying their requirements [14].

Applications that can most take advantage of the fog and edge computing paradigms require very low and predictable latency such as gaming and video conferences. Additionally, applications that employ sensor networks to monitor the environment such as smart vehicles can take advantage of the geo-distributed fog and edge characteristic, fast mobile applications, and mobility support. Large-scale distributed control systems such as smart traffic light systems are another application type that can benefit from the fog and edge paradigms due to its location awareness and geo-distributed nodes.

As sending data to the cloud for the neural network processing may increase latency and/or decrease the inference rate, in this work, we do not consider cloud processing. Instead, we employ end devices at the edge because we want to take advantage of the remaining resources present in IoT devices within an environment [73, 96]. Thus, we only consider IoT devices for the neural network partitioning.

2.4 Machine Learning

Machine learning applies statistics to estimate functions using computers [34, 82]. A machine learning algorithm is an algorithm that learns from data. Machine learning can be divided into supervised learning, when data are labeled, or unsupervised learning, when there is not such a label for each sample [34, 82]. A machine learning system is composed of a dataset, a model that tries to generalize some function based on the dataset, a cost function that the model tries to minimize, and an optimization algorithm that tries to reduce the cost function value, i.e., the learning procedure. The model has parameters whose values are learned by the optimization algorithm and that correspond to features, which are relevant information that can be obtained from the dataset.

The main challenge in machine learning is how to adequately predict the result of previously unobserved inputs, an ability called generalization [34]. During the learning procedure, the objective is to reduce the training error, which is the error measure for a machine learning algorithm when trying to predict the result of inputs from the training set. The learning corresponds to an optimization problem. In machine learning, it is also necessary to reduce the generalization error, or test error, which can be estimated by measuring the algorithm performance on a test set of inputs with different samples from the training set.

Neural networks are one of the most powerful techniques in machine learning and consist of layers, which, in turn, are formed by many simple, connected neurons. These neurons calculate real-valued activations, starting with input neurons that may represent data captured by sensors in the environment, passing through the layers of trainable neurons, which have weighted connections to neurons from their previous layer, until the output neurons, which calculate the final result. Figure 2.6 illustrates the basic structure of a neural network with connections to all neurons from the previous layer. This is a fully connected neural network, which is known as Multilayer Perceptron (MLP) [72, 104]. The final result may be a classification probability or even actions in the environment, performed by actuators.

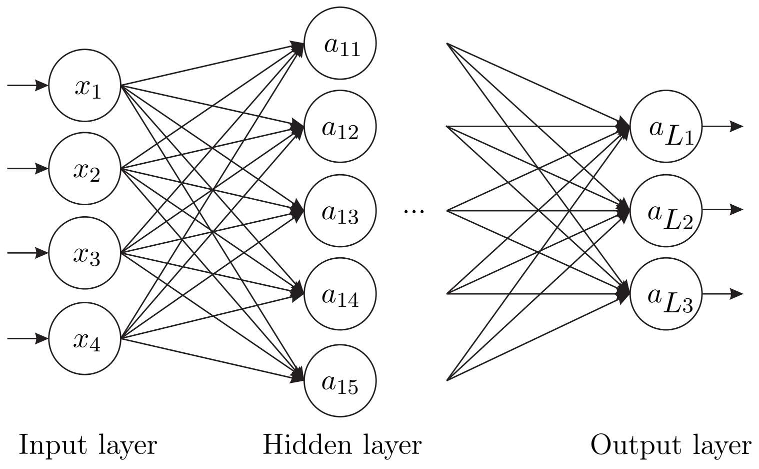


Figure 2.6: Basic structure of a neural network: an input layer, one or more hidden layers, and an output layer. This fully connected neural network is the MLP [100] (modified).

2.4.1 Multilayer Perceptron

Several deep learning methods use Multilayer Perceptron (MLP) layers, which contain neurons that have connections to every neuron from the previous layer. Figure 2.6 exemplifies this type of connection between layers. The first layer comprises the input, which receives data that may have been captured by a sensor, for example, then the hidden layers transform these data to achieve a classification output with the inference or class of the input values. Each neuron of the output layer corresponds to a class.

After the training procedure, the optimal values for the parameters of such a neural network are set and the inference can be performed to predict values for new inputs. In the case of MLPs, the inference is simply the step of feedforward evaluation from the training step. This step comprises a matrix multiplication followed by an addition and, after that, a function is applied to the final result. Furthermore, this operation is performed between each neural network layer and the previous layer to calculate the activation of neuron i a_i, starting with the input layer:

$$a_i = f\left(b_i + \sum_{j=1}^{J} w_{ij} \times a_j\right), \tag{2.1}$$

in which w_{ij} is the weight of the connection between neuron i at layer l and neuron j at layer $l-1$, b_i is the bias term associated with neuron i, f is the usually nonlinear activation function that is applied to every neuron, often sigmoid or rectified linear functions, and J is the number of neurons at layer $l-1$. The parameters found in the learning step are the weights and bias. The inference process continues until the output layer calculates the prediction value, which may be a classification or a real value.

2.5 Deep Learning

In recent years, deep learning techniques have achieved the state of the art in many pattern-recognition contests, in machine learning, finance forecast, driving assistance, among other areas [33, 52, 54, 93]. Deep learning started with unsupervised learning in 2006 but achieved notoriety when it outperformed important machine learning algorithms such as Support Vector Machines (SVMs) in many applications and pattern recognition competitions [93]. Another field in which deep learning outperformed was Reinforcement Learning.

Three main aspects helped deep learning to become notorious: the increase in the available training data made them more useful; improvements in hardware and software infrastructure enabled bigger and more complex models; and these more complex models were executed with increasing accuracy over time, besides allowing the solution of more complicated applications [34]. Interpreting sensor data is one of these complex problems, and deep learning is an adequate approach to handle it [60]. Examples of applications in which deep learning has shown promising results are speech recognition [4, 5], emotion recognition in videos [30], and face recognition [108].

In deep learning, we have the Deep Neural Network (DNN), which is a neural network with lots of layers that learn their features. A system is said to have learned from data when its weights or parameters are balanced so that the neural network produces in the output the desired behavior [34]. Each neural network layer learns increasing abstraction levels of the input data [116], usually transforming the aggregate activations of the previous layer by applying nonlinear functions. It is precisely this chained process along the layers that makes the neural network learn increasingly complex functions of the input. To train a supervised DNN, the most common algorithm is called Backpropagation, which is an efficient gradient descent method developed in the 1960s and applied to neural networks in 1981 [93].

There are three main DNN types: Feedforward Neural Networks (FNNs), Recurrent Neural Networks (RNNs) [116], and Undirected Neural Networks. FNNs calculate the composition of many functions and its model can be associated with a Directed Acyclic Graph (DAG) describing data flow [34]. In an FNN, data flow from the input and pass through the intermediate connections that define the mapping function until the output, i.e., there are no feedback connections. When there are many layers in an MLP, it becomes a DNN. RNNs have this type of feedback connection, so they are the deepest neural network, being more general than FNNs and processing any arbitrary input sequences. RNNs can efficiently learn using both sequential (temporal) and parallel (data features) information, are capable of fully making use of parallel processors, and are adequate for processing temporal data with dependence on each other [34] such as natural language modeling [79]. In Undirected neural networks, there are undirected connections between the layers, which means that data flow in both directions [34, 116]. In this work, we focused on Convolutional Neural Networks (CNNs), an FNN that has been the state of the art in many applications [54, 84, 106, 115]. Nonetheless, the proposed algorithms work for any neural network that can be represented as a DAG when considering the inference rate maximization objective function and for any neural network when considering the communication reduction objective function. In general, our algorithms work for any DAGs for the inference rate maximization and for any graphs for the communication reduction.

2.5.1 Convolutional Neural Networks

The Convolutional Neural Networks (CNNs) were loosely inspired by the visual cortex of animal brains [34]. In a convolution layer, every entry in its output volume may be viewed as the output of a neuron that is connected to only a small region of the input [100]. Like the MLP, CNNs have layers with learnable neurons and biases. These layers take data as input, perform a dot product between them and the filter weights, and, in most cases, apply a nonlinear function to the result [100]. This kind of neural network uses a convolution mathematical operation, which is a linear operation, in place of the general matrix multiplication of FNNs, in at least one layer. Fundamentally, CNNs learn simple representations from the input data at a high resolution, then convert these simple representations into increasingly complex representations at coarser resolutions along the layers. CNNs are composed of a sequence of one or more convolution and rectifier layers, pooling or subsampling layers, and fully connected layers, being this last type the same as the MLP explained in Subsection 2.4.1 [60]. The convolution layers apply convolution filters with a small kernel size to capture local data properties, then maximum or minimum pooling layers make the representations invariant to translations and also reduce their dimensionality. At last, fully connected layers indeed classify the sample.

CNNs started with the successful work of Lecun, Bottou, Bengio, and Haffner [63] in recognizing handwritten digits in images, known as the LeNet architecture [100]. In the field of computer vision, AlexNet was the first work that became famous: it did not only win the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [55] in 2012 but also outperformed the second place with a difference of 10% in the top five error, with an error of 16% [54]. Its architecture is deeper and bigger than LeNet's one with more than one convolution layer before a pooling layer. In 2013, the winner was the Zeiler and Fergus Network (ZFNet), which is a hyperparameter improvement of AlexNet: it increased the size of the middle convolution layers and decreased the filter and the stride sizes in the first layer [129]. Hyperparameters, in contrast to the model parameters or weights learned during the learning procedure, are the parameters that define the model architecture: the kernel size for the convolution, number of features learned by the layer, and the size of the output volume. The hyperparameters are detailed later in this section. GoogLeNet won the ILSVRC contest in 2014, presenting the Inception Module, which significantly reduced the number of required parameters from 60 millions in the AlexNet architecture to 4 millions and substituted the fully connected layers for average pooling layers [106]. Still in 2014, another relevant neural network was the Visual Geometry Group Network (VGGNet), which came in second place in the ILSVRC [97]. It is deep: there are 16 layers, considering convolution and fully connected ones, and it uses the same convolution (3×3) and pooling (2×2) filter sizes in every layer [100]. The main disadvantage of this model is that it is expensive to train and to classify and uses lots of parameters (140 millions) and memory, which are mostly in the first fully connected layer. The pretrain of this model is available to use with the Caffe framework [36]. The Residual Network (ResNet) was the winner in 2015 [41]. It uses skip connections and batch normalization, but, in the neural network end, there are not any fully connected layers [100]. This model is also available, for use with the Torch framework [40]. In 2016, the Trimps-Soushen team won the challenge, reaching a classification error of 2.991% [56, 94]. They employed an ensemble of models such as Inception, Inception-ResNet, ResNet [107], and a Wide Residual Network [128] to predict image class labels, merging the results from each model by using each model accuracy as weights. WMW won the contest in 2017 with a classification error of only 2.251% [43, 57]. They designed a new architecture building block called Squeeze-and-Excitation, which embeds information from global receptive fields with the squeeze operation and selectively improves the result with the excitation operation. Receptive fields are a property of convolution and pooling layers and we explain them ahead in this subsection. As in 2017 76% of the competitors had an accuracy that was over 95%, surpassing the human performance in image detection, the ImageNet group decided to propose another challenge in visual recognition, the classification of Three-Dimensional (3D) objects with a description using natural language [91].

One of the problems that act as a motivation or advantage of CNNs is that MLPs do not scale to images [100]. For example, for input images with a size of only $32 \times 32 \times 3$ (32 wide pixels, 32 high pixels, and 3 color channels), a single layer with a single fully connected neuron requires $32 \times 32 \times 3 = 3072$ weights. For images with a size of $200 \times 200 \times 3$, the nuumber of required weights for only a single neuron grows to 120,000 weights, being this image size far from the video size captured by high-definition cameras of simple smartphones ($1920 \times 1080 = 2,073,600$ pixels). With this vast number of parameters, the neural network is prone to overfitting or may require a vast amount of data. Overfitting may happen when the model is too sophisticated to represent the input data, i.e., the model presents too many parameters, or when the number of samples in the training set is too small.

Another advantage of CNNs that improves machine learning is the sparse interactions, or sparse connectivity or weights. In the convolution layers, a kernel matrix is convolved with some neurons from the previous layer. As the kernel size is usually smaller than the input size, only a small region of the input is involved in each calculation, leading to a more efficient process because of the fewer number of operations. The kernel is used to detect small but important features, such as edges, using fewer parameters. The use of a small kernel also reduces the required amount of memory and improves the model's statistical efficiency. These efficiency improvements bring high performance to many practical applications [34]. A consequence of these sparse interactions is that neurons in deeper layers may indirectly learn from a larger portion of the original input. Thus, CNNs can efficiently learn complex interactions between many variables by employing simple building blocks that only describe sparse interactions, as shown in Figure 2.7. In this figure, neurons from more distant layers, for instance, a_{23}, despite having a small receptive field related to its previous layer, are indirectly connected to a larger portion of the input through the receptive fields of the neurons of previous layers to which they are connected. In this example, each neuron has a receptive field of size three. Receptive fields correspond to the portion of the input volume that is connected to a neuron in the output volume.

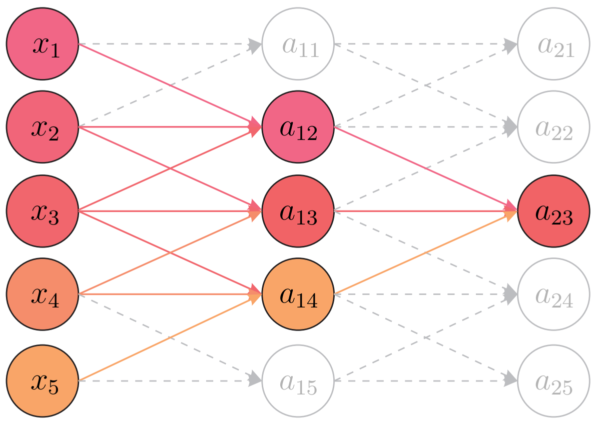


Figure 2.7: Sparse interactions between neurons from different layers. These interactions indirectly connect neurons from distant layers to a larger portion of previous layers and the input [34] (modified).

There is another characteristic that further reduces memory and computation requirements for CNNs: parameter sharing [34]. In MLPs, there is one parameter for each element of the matrix multiplication, which is used only once in the computation. In the convolution layers, instead, the parameters are only defined by the kernel size used in the convolution operation and, as the kernel size is usually much smaller than the input size, memory and computation requirements are greatly reduced. In the convolution operation, each parameter of the kernel is convolved with every input elements, except for the boundary elements in some cases.

The last advantage of CNNs that acts as motivation is the equivariance property related to translation [34]. It occurs in convolution layers due to the convolution operation and the way the parameters are shared. Some function $m(x)$ is equivariant to another function $g(x)$ if $m(g(x)) = g(m(x))$, e.g., the application of a translation operation followed by a convolution results in the same output as if the convolution were applied before the translation. This property can be used to detect small patterns in images by using the neighboring pixels of a pixel in multiple locations. For images, it is useful to be able to find edges everywhere in the image during the CNN first layer. However, for cropped images that are centered in some specific part of the images, e.g., faces, this feature may not be so useful as, in this case, it is usually interesting to detect different patterns depending on the image location. It is worth noting that not all transformations, such as rotation and scaling, are equivariant to convolution; in these cases, other approaches are necessary.

CNNs assume that the input data have a known grid-like topology [34], so their architecture can use some specific properties to implement a more efficient feedforward function and to allow an aggressive reduction in the number of the neural network parameters [100]. Usually applied to image data, which is a Two-Dimensional (2D) grid of pixels, CNNs can also be applied to time-series data, in which one axis is time and the other is the feature being sampled.

The architecture of a CNN is based on specific characteristics of images: the CNN layers are organized in three dimensions, which are width, height, and depth. The layer depth should not be confused with the neural network depth, which is the neural network number of layers. Figure 2.8 shows an example of a CNN architecture with four layers. The first layer is the input, which usually has depth three for color images, the second and third layers are hidden layers, and the last layer is the output, which is a 1 wide x 1 high layer, and the depth corresponds to the possible output classes.

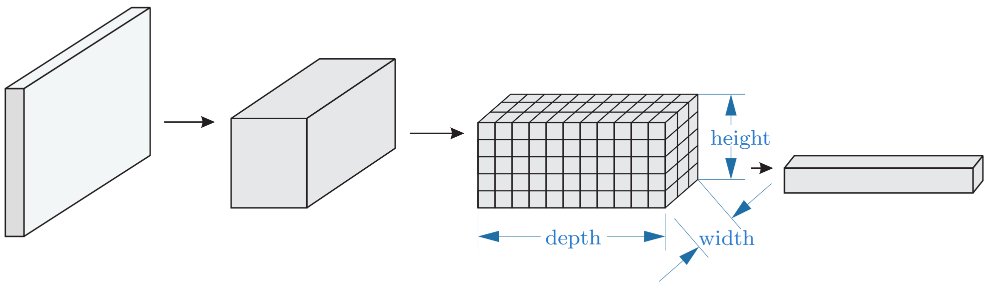


Figure 2.8: An example of a CNN architecture with four layers: an input layer with depth three, two hidden layers, and the output layer with size 1 wide x 1 high and the depth corresponding to the number of output classes [100] (modified).

To explain the CNN, it is necessary to detail each layer type and the CNN possible architectures. A simple CNN architecture may have an input layer, a convolution layer, a rectifier layer, a pooling layer, and a fully connected layer [100]. The convolution and fully connected layers contain trainable parameters, while the rectifier and the pooling layers usually do not have any trainable parameters.

The input layer contains one image, composed by a volume of pixels: the height and width of the image, and, in the case of color images, three color channels, for instance, Red, Green, and Blue (RGB), form the image depth [100].

The convolution layer applies the convolution, which is an operation on two functions y and c of a real-valued input t [34]:

$$s(t) = \int_{-\infty}^{\infty} y(o)c(t-o)do,$$

or

$$s(t) = (y*c)(t),$$

in which, in a CNN, y is the input, c is the kernel function, s is the output volume, t is the domain, which may be time, and o is a shift related to t. In the case of the time domain, since the input data are processed by a computer, time is discretized, meaning that the sensors provide data at regular time intervals. Using a linear transformation, the time index t can be represented only by integer values, so it is possible to define a discrete convolution operation defined for integer values of t in both functions y and c:

$$s(t) = \sum_{o=-\infty}^{\infty} y(o)c(t-o).$$

For machine learning, the input is often a multidimensional array of data as well as the kernel is often a multidimensional array of parameters, or tensors. As an array is defined only for its indexes, i.e., an array is a finite set of elements, it is possible to calculate the discrete convolution as a summation over only this finite number of elements. Furthermore, convolution is usually applied to 2D data:

$$CC(r,q) = (I*K)(r,q) = \sum_{kw}\sum_{u} I(r+kw, q+u)K(kw,u), \tag{2.2}$$

in which r is the row index of matrix CC, q is the column index of matrix CC, I is a 2D image (the input), K is a 2D kernel, kw is the kernel width, and u is the kernel height. Equation (2.2) is called cross-correlation, a related function that is the same as the convolution but without flipping the kernel [34]. Figure 2.9 exemplifies a convolution operation on a 2D tensor. Because the kernel is usually much smaller than the input, convolution may be considered a matrix multiplication in which the kernel has many entries equal to zero so that its size is equivalent to the input size.

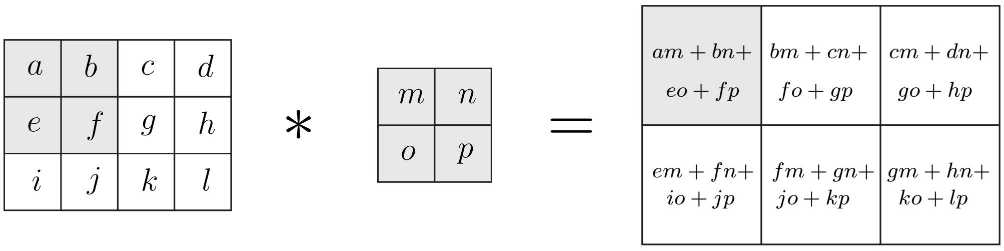


Figure 2.9: An example of a 2D convolution without kernel-flipping for an input of size 3×4 and a kernel of size 2×2, resulting in an output with size 2×3. The painted boxes represent the regions involved in the convolution calculation of the first output position [34] (modified).

The convolution layer parameters are a set of learnable filters, which are small along width and height but extends through the entire depth of the previous layer volume data [34]. During the inference, each filter is convolved with the input, across its width and height, generating a 2D activation map. Then, these maps are stacked along the depth to produce the output volume of the layer. Figure 2.10 exemplifies these connections and the feature maps.

The convolution layer has four hyperparameters, which define the output volume size [100]. The first hyperparameter is the receptive field size ($\mathbf{R}$), which is equivalent to the filter size. The receptive field corresponds to the portion of the input volume that is connected to a neuron in the output volume. For the depth axis, an output neuron is connected to every neuron in the input volume depth. Thus, the connections are local along width and height and global along the input volume depth. The receptive field size is given by a vector of dimension 1×2 because its depth is equal to the input depth. Figure 2.10 shows the receptive field connections.

The second hyperparameter is the output volume depth, which corresponds to the number of filters (F) for one convolution layer [100]. Each filter can learn a different feature from its input volume. The set of neurons that is connected to the same region

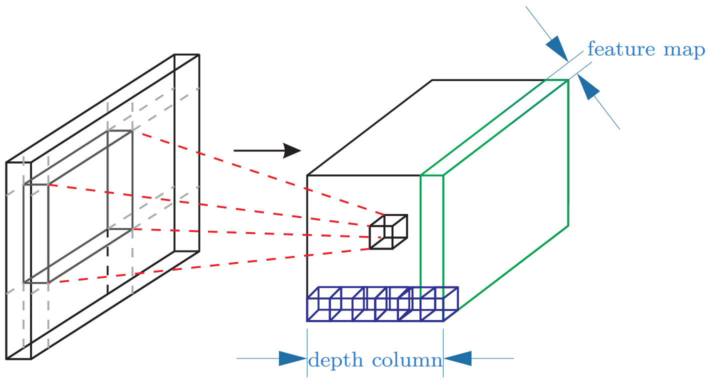


Figure 2.10: The receptive field of each neuron, which comprises only a portion of the full width and height of the input layer but its complete depth. Furthermore, each depth slice (a 2D matrix) consists of a feature map and the output neurons in the same position in width and height form a depth column, also known as depth fiber [100] (modified).

of the input forms a depth column or a fiber, as shown in Figure 2.10.

The third hyperparameter is the stride (T), which is the distance in pixels between each convolution. For instance, when the stride is one, the filter is moved one pixel after each convolution operation. For strides of two or three, which are uncommon, or more, which are rare, the convolution is applied and then two, three, or more pixels are skipped until the next convolution operation. Clearly, in these cases, the output volume gets increasingly smaller.

Finally, the last hyperparameter is the zero padding (Z), which allows for better control of the output volume. Zero padding is usually applied to maintain the input width and height and consists in filling the input volume borders with zeros before applying the convolution.

Considering the four hyperparameters explained above and that the convolution layer receives an input volume of size $W_i \times H_i \times D_i$, it is possible to calculate the output volume of a convolution layer:

$$\begin{aligned} W_c &= \frac{W_i - R_w + 2Z}{T} + 1, \\ H_c &= \frac{H_i - R_h + 2Z}{T} + 1, \\ D_c &= F, \end{aligned} \tag{2.3}$$

with $R_w \times R_h \times D_i$ weights per filter for a total of $(R_w \times R_h \times D_i) \times F$ weights and F biases in the layer. With Equation (2.3), some hyperparameter settings may be invalid and the choice should be careful to meet a valid configuration. Common recommendations include small filters (3×3 mainly, or 5×5), $T = 1$ (leaving the subsampling only to the pooling layers and causing the convolutional layers to be responsible for changing only the depth), and a zero padding that maintains the input size ($Z = \frac{\mathbf{R}-1}{2}$), which helps to keep the border information during more layers than without any padding [100].

Before the pooling layer, the rectifier layer applies an element-wise activation function, such as $\max(0, x)$, which does not change the input size. The pooling layer subsamples along the height and width of its input data volume, resulting in a smaller volume. This layer usually stays between convolution layers to increasingly reduce the representation spatial size [100]. Therefore, the pooling layer reduces the number of parameters, computation, and, thus, the probability of overfitting and the number of samples needed for training.

The pooling operation is applied to each depth slice independently, resizing the width and height of the input volume according to the filter and stride sizes. The most common pooling operation is the maximum function, but others such as arithmetic mean or $\mathcal{L}2$ norm are also used. The most used configuration, $\mathbf{R} = 2 \times 2$ and $T = 2$, takes the maximum value over four numbers and discards 75% of the activations, without any changes to the depth dimension. Another configuration, with $\mathbf{R} = 3 \times 3$ and $T = 2$, called overlapping pooling, is also used in practice, but larger filters are rarely seen because there is too much information loss in this process. A general pooling layer receives as input a volume $W_i \times H_i \times D_i$ and, with the parameters $\mathbf{R}$ and T, produces an output volume $W_p \times H_p \times D_p$:

$$\begin{aligned} W_p &= \frac{W_i - R_w}{T} + 1, \\ H_p &= \frac{H_i - R_h}{T} + 1, \\ D_p &= D_i. \end{aligned}$$

Finally, the fully connected layer at the CNN end calculates the class probabilities, usually resulting in a volume of size $1 \times 1 \times$ the number of classes. It is possible to convert a fully connected layer into a convolution layer, gaining efficiency for larger input sizes [100]. In this conversion, the convolution layer would have as hyperparameters $R_w = W_i$, $R_h = H_i$, $Z = 0$, $T = 1$, the hyperparameter F would be the number of neurons in the fully connected layer depth, and the output volume would be $1 \times 1 \times F$, with results identical to the fully connected layer.

Typically, a CNN architecture is composed of some convolution and rectifier layers, optionally followed by pooling layers, then more convolution and rectifier layers may be present, optionally followed by pooling layers again, until the image has an appropriately small size. In the end, the architecture has fully connected layers followed by rectifier layers, with the last layer being the output of a fully connected layer, which may be the probabilities of each class. For a general architecture type:

$$\begin{aligned} &\text{input} \rightarrow [(\text{convolution} \rightarrow \text{rectifier}) \times U \rightarrow \text{pooling?}] \times M \rightarrow \\ &\qquad\qquad (\text{fully connected} \rightarrow \text{rectifier}) \times Q \rightarrow \text{fully connected}, \end{aligned}$$

with $0 \leqslant U \leqslant 3$, the question mark indicating that the pooling layer is optional, $M \geqslant 0$, and $0 \leqslant Q < 3$, usually. With $U = M = Q = 0$, the classifier is linear. For $U > 1$, it means that there are two or more convolution layers before a pooling layer. This configuration is useful for DNNs because it allows the development of more complex features before the destructive pooling operation [100].

The last concern is about computation and memory. The first convolution layers are the ones that require most of the computation and memory since the data are large in the first layer and get shrunk during the subsequent layers. However, most of the parameters come from the last fully connected layers, since each neuron in this layer is connected to every neuron in the previous layer. Memory is the main concern in CNNs, nevertheless, for the inference process, it is necessary to store the activations only for the current layer. Therefore, the activations of the previous layers may be discarded to save memory.

2.6 Neural Network Models as Dataflow Graphs and Partitioned Neural Networks

One of the challenges that are addressed in this work is the automatic partitioning of CNNs. Some important concepts need to be defined before proceeding with the related work in machine learning, IoT, and partitioning frameworks. The understanding of how a neural network can be represented as a dataflow graph and how this graph can be partitioned is essential to this thesis.

Neural networks can be modeled as a dataflow graph. Dataflow graphs are composed of a DAG that models the computation of a program through its data flow [120]. In a dataflow graph, vertices represent computations and may send/receive data to/from other vertices in the graph. In our approach, a vertex represents one or more neural network neurons and may also require an amount of memory to store the intermediate (layer) results and the neural network parameters and biases required by the respective neurons that this vertex represents. Dataflow graph edges may contain weights to represent different amounts of data that are sent to other vertices.

In this section, two simple theoretical examples are outlined. The first example is depicted in Figure 2.11a, in which there is a simple fully connected neural network represented as a dataflow graph. In this graph, each dataflow vertex represents one neural network neuron. The first layer is the input layer with two vertices; each vertex requires 4 bytes (B) to store the neuron input value, if we use data represented with 4 B. In this layer, the vertices do not perform any computation. The second layer is the hidden fully connected layer; each vertex requires 12 B, being 4 B to store the neuron intermediate result and the other 8 B to store the neuron parameters, which are the edge weights that are multiplied by each input value. It is worth noting that, in this example, no bias is used, so the bias weight is not needed. Furthermore, in the case of CNNs, there is only one set of parameters per layer in the case of convolution layers and not parameters per neurons as in FNNs. Each vertex in this layer performs 4 FLOP per inference, which correspond to the multiplication of the input values by the parameters, to the sum of both multiplied values, and the application of a function to this result. We presented all these calculations in Equation (2.1). The last layer is a fully connected output layer that contains one vertex; this vertex requires 16 B, being 4 B to store the final result and the other 12 B to store the neuron parameters. This vertex performs 6 FLOP, which correspond to the three multiplications of the parameters by the layer input values, to the two sums of the multiplied values, and the application of a function to this result. Again, Equation (2.1) presents all these calculations.

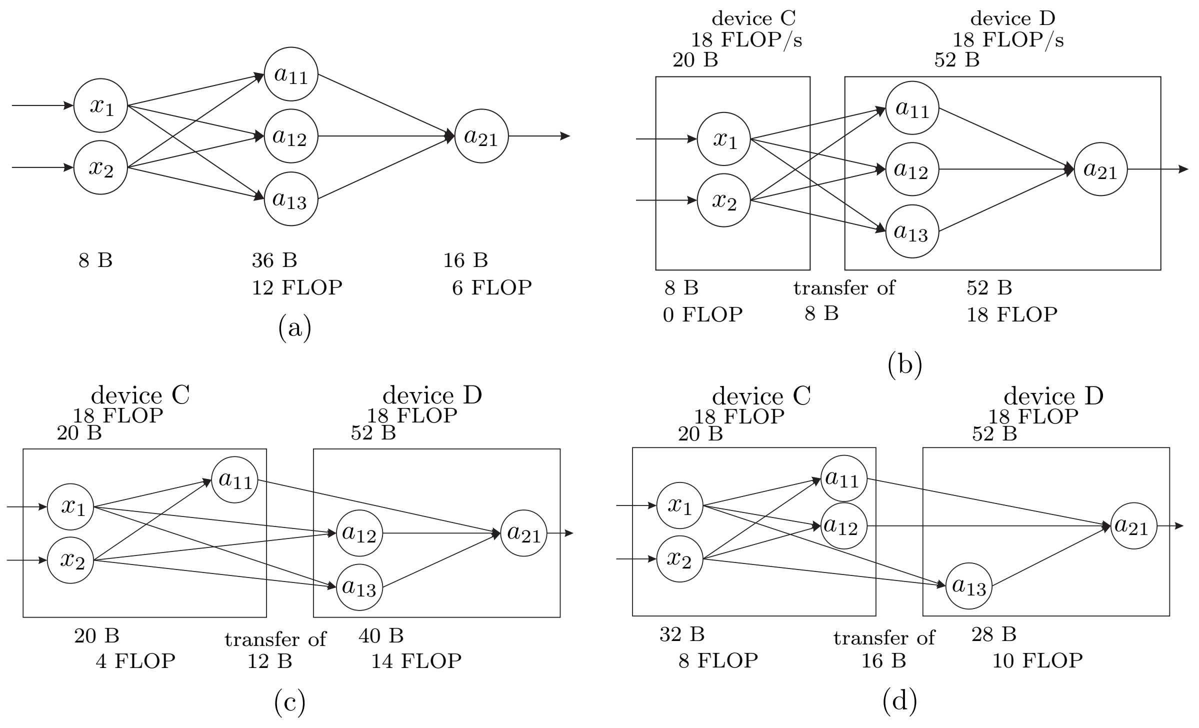


Figure 2.11: (a) Example of how a fully connected neural network may be represented as a dataflow graph. (b) Example of how the dataflow graph represented in item (a) can be partitioned to produce an optimal partitioning; (c) a suboptimal partitioning; and (d) an invalid partitioning.

Figure 2.11b shows the same dataflow graph partitioned for distributed execution on two fictional devices: device C, which can perform 18 FLOP/second (FLOP/s) and provide 20 B of memory, and device D, which can also perform 18 FLOP/s and provide 52 B of memory. Additionally, the communication link between these devices can transfer up to 4 B per second. The amount of transferred data per inference in this partitioning is 8 B because, although six edges cross the partitions, they represent the data transfer of only 8 B.

We define the cost of a partitioning as the calculation of the objective (or cost) function for that partitioning. If we want to optimize the neural network for the inference rate, then this cost is the inference rate calculation for the partitioning that we have at hand. Since all devices and communication links can work in parallel, we can calculate the inference rate of a partitioned neural network as the minimum value between the inference rate of the devices and the inference rate of the communication links between each pair of devices, according to

$$\text{inference rate} = \min(\text{inference rate}_{devices}, \text{inference rate}_{links}). \tag{2.4}$$

The inference rate of the devices, expressed in inferences per second, is calculated as the minimum value between the computational performance of each device *dev*, expressed in FLOP/s, divided by the total computational requirement of the vertices that compose the partition assigned to that device, expressed in FLOP/inference:

$$\text{inference rate}_{devices} = \min\left[\left(\frac{\text{computational performance}}{\text{computational load}}\right)_{dev}\right], \forall\ dev \in 1, ..., p, \quad (2.5)$$

in which p is the number of devices in the system. The inference rate of the communication links between each pair of devices, also expressed in inferences per second, is calculated as the minimum value between the transfer performance of each link, expressed in bytes per second, divided by the total communication requirement of the two partitions involved in this link, expressed in bytes per inference:

$$\text{inference rate}_{links} = \min\left[\left(\frac{\text{link performance}}{\text{communication load}}\right)_{dev_1 dev_2}\right], \forall\ dev_1, dev_2 \in 1, ..., p, \quad (2.6)$$

in which dev_1dev_2 represents the communication link between devices dev_1 and dev_2. Thus, the device or connection between devices that most limit the result is the maximum inference rate that some partitioning can provide.

When there is a data stream that must be processed as fast as possible, for instance, the inference rate maximization can be the objective function for the partitionings. Thus, taking into account Equation (2.4), Equation (2.5), Equation (2.6), and using the scenarios of Figure 2.11, in the partitioning of Figure 2.11b, device C can perform $18/0 = \infty$ inferences/s, which means that device C does not limit the inference rate. The communication link between device C and device D can perform $4/8 = 0.5$ inference/s. Device D can perform $18/18 = 1$ inference/s. Therefore, the inference rate of this partitioning is

$$\infty\ \frac{\text{inferences}}{s}\ \text{(C)}\ \frac{1}{2}\ \frac{\text{inference}}{s}\ \text{(transmission)}\ \frac{18}{18}\ \frac{\text{inference}}{s}\ \text{(D)} = 0.5\ \frac{\text{inference}}{s},$$

which is the minimum value among the inference rate of the devices and the communication links. The partitioning of Figure 2.11b is valid because both partitions respect the memory limit of the devices and is optimal since, in the scheme of Figure 2.11c, the system can only perform

$$\frac{18}{4}\ \frac{\text{inferences}}{s}\ \text{(C)}\ \frac{1}{3}\ \frac{\text{inference}}{s}\ \text{(transmission)}\ \frac{18}{10}\ \frac{\text{inference}}{s}\ \text{(D)} = 0.33\ \frac{\text{inference}}{s}$$

and, thus, is suboptimal. The partitioning of Figure 2.11c is also valid because both partitions respect the memory limit of the devices. The last partition scheme in Figure 2.11d is invalid because it needs more memory than device C can provide.

As the inference rate was limited by the communication link in this case, the optimal partitioning was the one that transferred a smaller amount of memory per second. Thus, if we wanted to change the objective function so that we chose a partitioning with the minimum amount of communication, the same partitioning of Figure 2.11b would be the optimal partitioning. However, for complex systems with more devices and more communication links, we may have different partitionings when we optimize a DNN for inference rate maximization or communication reduction. As this thesis shows in the next chapters, it is important to consider the trade-off between computation and communication in a distributed DNN, even though, intuitively, communication reduction may improve the inference rate. Furthermore, the example indicates that not all the neural network connections result in data transfer among the devices because some of them represent the transfer of the same data.

The second example consists in a fully connected neural network with five input neurons that need 20 B, being 4 B for each neuron to store its input value, one neuron in the hidden layer that needs 24 B, being 4 B so that the neuron stores its result and 20 B to store the parameters, and one neuron in the output layer requiring 4 B, as shown in Figure 2.12a. Furthermore, the nine operations in the single neuron of the hidden layer were modeled as nine vertices of weight one in the graph, i.e., each vertex calculates 1 FLOP per inference. This modeling rendered the graph bigger, however, with finer granularity and allowed the partitioning of the operations inside a neuron. This model can be seen in Figure 2.12b. It is worth noting that every vertex needs at least 4 B to store its result and vertices that represent a multiplication require 4 B more to store their parameters.

As this example is simple, it is possible to visualize an acceptable partitioning. Considering that device C can now provide 48 B, device D, 32 B, and the objective function is to maximize the inference rate, with a random partition scheme, it is possible to achieve a suboptimal result like the one in Figure 2.12c for simple neural networks such as the one in this example:

$$\frac{18}{4}\ \frac{\text{inferences}}{s}\ (\text{C})\ \frac{1}{4}\ \frac{\text{inference}}{s}\ (\text{transmission})\ \frac{18}{5}\ \frac{\text{inference}}{s}\ (\text{D}) = 0.25\ \frac{\text{inference}}{s}.$$

By applying an automatic partitioning algorithm, it is possible to reach an optimal or near-optimal partitioning, as shown in Figure 2.12d:

$$\frac{18}{6}\ \frac{\text{inferences}}{s}\ (\text{C})\ \frac{1}{2}\ \frac{\text{inference}}{s}\ (\text{transmission})\ \frac{18}{3}\ \frac{\text{inference}}{s}\ (\text{D}) = 0.5\ \frac{\text{inference}}{s}.$$

Nevertheless, if the restrictions are not properly considered, it is possible to remain in a suboptimal partitioning or even reach an invalid partitioning, as in Figure 2.12e, in which the partition in device D needs more 16 B.

The second example shows the importance of employing an automatic partitioning algorithm to reach partitionings that are better optimized for certain objective functions. This algorithm should also properly take into account all the restrictions. The example indicates how increasing size graphs add further complications to the problem, rendering it more difficult to visualize an optimal partitioning or even an acceptable result without using any partitioning algorithm.

Finally, when partitioning CNNs, we have to consider the minimum amount of memory required by each device, besides the total amount of memory provided by all the devices in the partitioning. Each device in the partitioning should contain an amount of memory that is at least equal to the memory required by the vertex and its corresponding set of parameters and bias that, summed, equals to the largest amount of memory among all neural network vertices.

2.7 Synchronization

When transferring information from one partition to another, synchronization may be necessary to ensure correctness. The synchronization guarantees that all the data that a vertex needs to calculate its computation arrive in its inputs. However, it may require extra time and reduce the inference performance. It is worth pointing out that techniques such as retiming [64] can be employed to the partitionings provided by our algorithms to enforce synchronization. Such a technique would increase the amount of memory required to execute the CNN in a distributed form. If we do not employ devices with a larger amount of memory, we may need to use more devices and, then, the amount of transferred data may be increased. This increased amount of transferred data may impact the inference rate in the case that it becomes the execution bottleneck. Although this is an important issue for the deployment of distributed CNNs on constrained IoT devices, in this thesis, we are not concerned by it because one of our aims is to show how better our algorithms can be in partitioning CNNs for constrained IoT devices when comparing to state-of-the-art partitioning algorithms, which do not include the synchronization overhead as well.

2.8 Problem Definition

In this section, we formally define the partitioning problem as an objective-function optimization problem subject to constraints [76]. First, we define a function that returns 1 if an element e is assigned to partition p and 0 otherwise:

$$partition(p, e) = \begin{cases} 1, & \text{if } e \text{ is assigned to } p; \\ 0, & \text{otherwise.} \end{cases}$$

We can define the partitioning problem as an optimization problem with an objective function subject to memory constraints:

$$\begin{array}{ll} \text{optimize} & cost \\ \text{subject to} & \sum_{n=1}^{N} m_n \times partition(p, n) + \sum_{l=1}^{L} m_{sbp_l} \times partition(p, l) \leq m_p, \forall\ p \in [1...P], \end{array}$$

in which $cost$ is the objective function (detailed below), N represents the number of neurons in the DNN, m_n equals the memory required by neuron n, L is the number of layers of the DNN, m_{sbp_l} represents the memory required by the shared parameters and biases of layer l, m_p equals the memory that partition p can provide, and P is the number of partitions in the system. It is worth noting that $partition(p, l)$ returns 1 if any neuron of layer l is assigned to partition p.

If we want to reduce communication, we can define a function that returns 1 if two elements are assigned to different partitions and 0 otherwise:

$$\text{diff}(e, h) = \begin{cases} 1, & \text{if } e \text{ and } h \text{ are assigned to different partitions;} \\ 0, & \text{otherwise.} \end{cases}$$

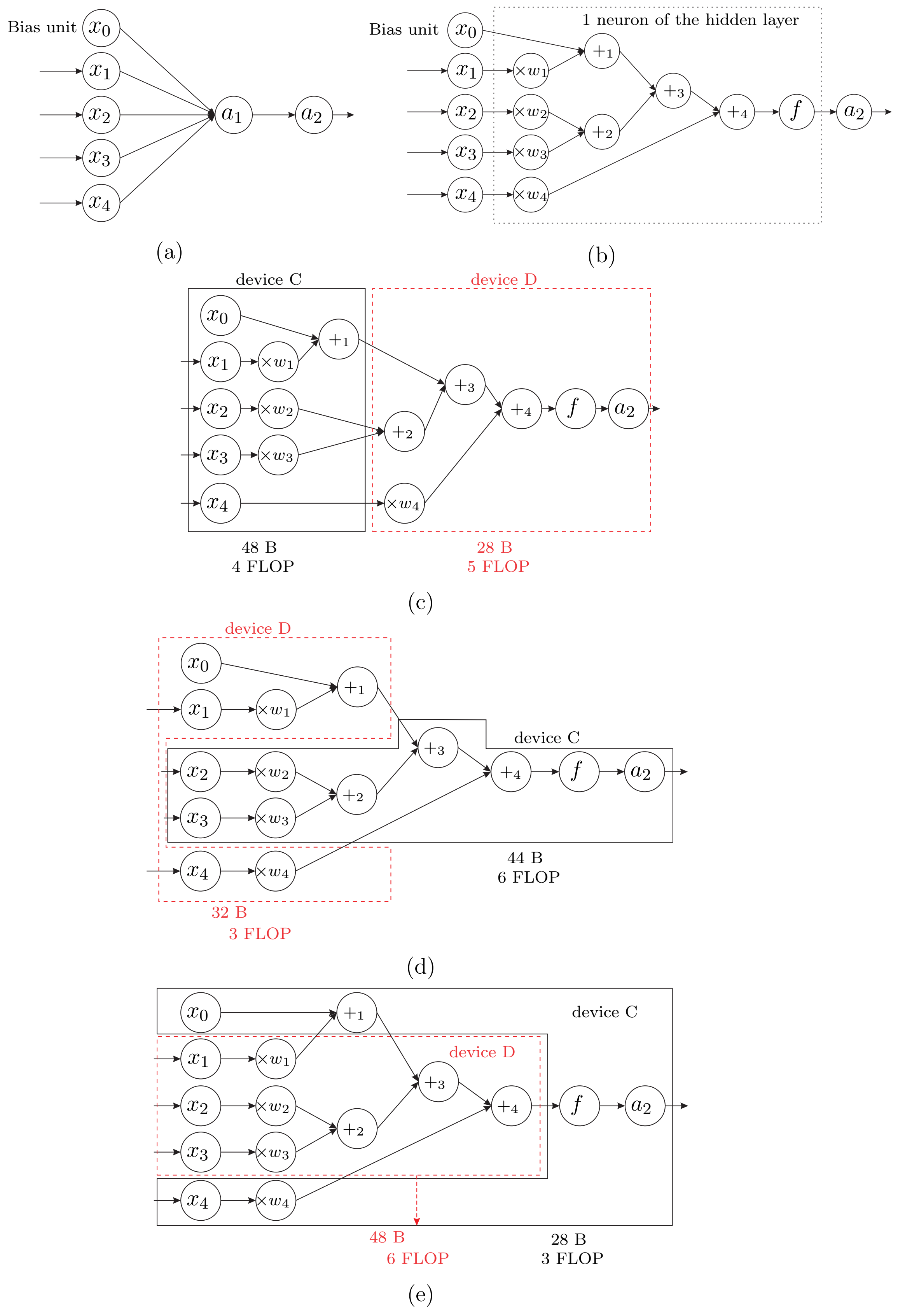


Figure 2.12: (a) The fully connected neural network for the second example. (b) The hidden-layer neuron modeled as nine vertices, one for each neuron operation. (c) A random partitioning may be suboptimal. (d) The application of a partitioning algorithm may lead to an optimal or near-optimal partitioning. (e) An invalid partitioning.

Then, we can define the communication cost, expressed in bytes per inference, as

$$\text{communication cost} = \sum_{n=1}^{N} \sum_{z=1}^{adj(n)} \text{edge weight}_{nz} \times \text{diff}(n, z),$$

in which $adj(n)$ are the adjacent neurons of neuron n and edge weight$_{nz}$ is the weight of the edge between neurons n and z, which is also expressed in bytes per inference.

If we want to maximize the inference rate, then Equation (2.4) represents the cost function and we rewrite it here:

$$\text{inference rate} = \min(\text{inference rate}_{devices}, \text{inference rate}_{links}). \tag{2.4}$$

To formally define the optimization problem, we can rewrite the computational load of device dev of Equation (2.5):

$$\text{inference rate}_{devices} = \min\left[\left(\frac{\text{computational performance}}{\text{computational load}}\right)_{dev}\right], \forall\ dev \in 1, ..., p, \tag{2.5}$$

as

$$\text{computational load}_{dev} = \sum_{n=1}^{N} \text{computational load}_{n} \times partition(dev, n),$$

and the communication load between devices dev_1 and dev_2 of Equation (2.6):

$$\text{inference rate}_{links} = \min\left[\left(\frac{\text{link performance}}{\text{communication load}}\right)_{dev_1 dev_2}\right], \forall\ dev_1, dev_2 \in 1, ..., p, \tag{2.6}$$

as

$$\text{communication load}_{dev_1 dev_2} = \sum_{n=1}^{N} \sum_{z=1}^{adj(n)} \text{edge weight}_{nz} \times \text{diff}(n, z) \times partition(dev_1, n) \times partition(dev_2, z).$$

2.9 Final Remarks

With the concepts presented in this chapter, we bounded the scenarios considered in this thesis: we have several IoT devices within an environment, some of them may be constrained in memory and computational performance, thus, we can use them to execute the inference of a CNN in a distributed way. For this purpose, we can model the CNN as a dataflow graph and we can partition it for the distributed execution, considering the amount of memory and the computational performance provided by the devices and the communication performance of the links between each device. The partitioning can be performed such that the communication between each device is minimized or the partitioning inference rate is maximized. As this problem belongs to the NP-complete class of problems [49], we can execute heuristic algorithms to find partitionings with reduced communication or increased inference rate. Thus, the next chapter presents the related work in general frameworks for machine learning, approaches and frameworks for machine learning on IoT, and partitioning algorithms.

Chapter 3

Related Work

This chapter discusses the state of the art in three topics. First, we present general frameworks for machine learning. Then, we introduce the related work in specific frameworks and approaches for IoT and machine learning. Finally, we present general-purpose partitioning algorithms. We performed a systematic review by the application of the snowballing and reverse snowballing techniques to find the most relevant papers for each topic and we used the Association for Computing Machinery (ACM), Institute of Electrical and Electronics Engineers (IEEE), and Web Of Science databases.

3.1 General Frameworks for Machine Learning

Many frameworks have been designed for training neural networks on computing nodes with multiple GPUs [12, 22, 45, 126] and for training DNNs on systems with multiple computing nodes [2, 18, 29, 68, 116, 118]. Traditionally, the works on the distribution of neural networks focused much more on the training phase than on the inference phase, considering that the training is more expensive to compute than the inference. Furthermore, most of the works also focus on the distribution to homogeneous systems.

For instance, Yan et al. [124] developed performance models to estimate the training time and to build a scalability optimizer to choose the best partitioning scheme to minimize the training time. Although they proposed an efficient algorithm, the focus is on the training time on homogeneous hardware.

Also for training, Ooi et al. [85] proposed a distributed deep learning system called SINGA to train large models with large datasets, which supports CNNs, Restricted Boltzmann Machines (RBMs), and RNNs. Though SINGA is a platform for training, it partitions the neural network at the neuron level.

TensorFlow is the current Google's machine learning framework that distributes neural networks for both training and inference on heterogeneous systems, ranging from mobile devices to large servers [1]. The user can partition the neural network at the level of neuron operations, presenting a static partitioning and scheduling. If not specifically assigned, TensorFlow performs the training phase of convolution layers in GPUs and the other layers in Central Processing Units (CPUs). The partitioning must be defined by the user, which is limited to a per-layer fashion to enable the use of TensorFlow's implemented functions, i.e., partitioning gets limited to operations in a whole layer, for instance, the whole convolution layer would be assigned to one device. The per-layer partitioning not only produces suboptimal results [28] but also cannot be deployed on very constrained devices [76]. Although this tool fills gaps in several features, it focuses on speeding up the training of neural networks on large clusters and puts aside challenges in deploying such a system for constrained IoT devices: its runtime may be too heavy to run on small, embedded devices and it does not consider the challenges of constrained IoT devices, for instance, memory and energy requirements [131].

PyTorch is a framework for deep learning that provides usability and speed [87]. It gives the user full control of all its aspects while its programming style makes debugging easy and is consistent with other popular scientific computing libraries. PyTorch focuses on distributing the training using data parallelism.

The strategies used by the deep learning frameworks cited until now focus on parallelizing the training phase. The most common partitioning strategy to parallelize the training phase is data parallelism. In this strategy, the whole neural network is deployed on every device and the neural network of each device is called a replica of the model. Each device trains its parameters locally for a subset of the training set, which can be viewed as a batch dimension parallelization. Data parallelism is not the most suitable approach to train very large models due to memory constraints. Additionally, it shows high latency and is inefficient for small batch sizes and layers with a large number of parameters, for instance, densely-connected layers like the fully-connected layer, becoming the bottleneck in large scale distributed training [47].

The works that employ data parallelism must use some technique to keep the parameters updated. One of these techniques is the use of parameter servers, which are processes that maintain the updated version of the model parameters, and workers, which are processes that are responsible for performing the computation needed to train the model. A basic architecture of a distributed DNN for training using data parallelism and parameter servers is shown in Figure 3.1. TensorFlow allows more flexibility making the use of parameter servers not mandatory and Watcharapichat et al. [117] proposed a decentralized distribution that does not need any parameter server. In their paper, the workers themselves communicate their updates to the others in a partitioned way to use the network constantly and without affecting convergency.

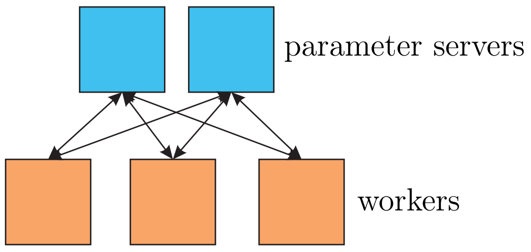


Figure 3.1: A distributed DNN basic architecture for training using data parallelism and parameter servers.

To overcome the limitations of data parallelism, another partitioning strategy is model parallelism [29], which partitions the neural network model to be executed in a distributed fashion on the devices. In this approach, there is no need for parameter synchronization, but there may be a lot of communication between devices. Krizhevsky [53] employed a hybrid approach to train DNNs [53]. In the convolutional and pooling layers, he applied data parallelism while, in the fully-connected layers, he applied model parallelism [47]. This approach improved performance on previous works and indicated that different layer types may benefit from different partitioning approaches. We also reached the same conclusion in one of our works [28].

Sze et al. [105] presented a comprehensive review of the efficient execution of DNNs. They focus on the inference phase, showing hardware platforms and architecture that support DNNs. Additionally, they discuss the trends towards the computational cost reduction of DNNs by hardware design changes and/or algorithm changes. They also provide metrics and design considerations to analyze new DNN hardware and algorithmic designs. Finally, they summarize key design considerations for DNNs, the trade-offs between several hardware architectures and platforms, and the techniques for DNN efficient processing.

The Google Brain team has recently published a paper with a study on deep learning with the data parallelism and the model parallelism approaches [95]. The authors also proposed a language to specify a general class of distributed computations called Mesh-TensorFlow. With this language, a user can define both types of parallelism.

Huang et al. [44] proposed a pipeline parallelism library to provide efficient, task-independent model parallelism in the neural network training. This approach can scale any neural network that is composed of layers. To avoid device under-utilization, they divided the training set into smaller sets and train the neural network using them, which enables different devices to work on different subsets of the training set simultaneously. After some interval, they update the parameters on each device. Thus, they employed a sort of data parallelism by creating a pipeline execution and also partitioned the neural network model into layers.

Jia et al. [47] proposed Sample-Operator-Attribute-Parameter (SOAP), a search space of parallelization strategies for DNNs that tries to generalize and overcome previous approaches. They argued that previous approaches only consider a subset of the SOAP search space. They also proposed FlexFlow, a deep learning engine that uses SOAP to automatically find an optimized parallelization strategy for a specific parallel setup. Additionally, FlexFlow is composed of an incremental execution simulator that evaluates different partitionings and a Markov Chain Monte Carlo search algorithm that uses the information from the execution simulator to rapidly explore the search space. Before this work, Jia et al. [46] proposed OptCNN, an algorithm that parallelizes linear CNNs in a per-layer approach using dynamic programming in the operator dimension. In the execution simulator, they used the task graph proposed in OptCNN: a graph that models both the DNN architecture and the cluster network topology. Then, the performance of some partitioning is estimated through the task graph execution simulation. The incremental execution uses a delta simulation algorithm that updates previous simulations and further improves performance.

Our work proposes model parallelism by partitioning DNNs in the Operator-Attribute–Parameter (OAP) dimensions. We do not use data parallelism because our scenarios present only constrained devices that do not provide the necessary memory for a complete DNN model. Namely, in our work, the operator dimension is equivalent to the per-layer approach, the attribute dimension is partitioning in the height and width of each layer, and the parameter dimension is partitioning the depth of each layer in the DNN. For convolution and pooling layers, we make replicas of the trained parameters when necessary and, for fully connected layers, we partition the parameter set together with the corresponding neurons. It is worth noting that our partitioning strategy keeps the inference result by design. Compared to our approach, FlexFlow does not consider memory restrictions, optimizes only for one objective function, which is the execution time, and considers only homogeneous hardware in the experiments.

We list the main characteristics of the works referenced here in Table 3.1 together with our approach. The partitioning approaches may be in the Sample (S), Sample-Operator (SO), Sample-Attribute (SA), Sample-Operator-Attribute (SOA), Sample-Operator-Parameter (SOP), OAP, and SOAP. Summing up, most distributed systems for DNNs focus on the computationally expensive training phase, consider a homogeneous distributed system, and target the neural network execution distribution at servers. In the IoT, the system should be able to adequately distribute the work to heterogeneous devices. Furthermore, the inference execution of neural networks also becomes challenging when we consider constrained devices. DNN models may need to be partitioned for distributed execution due to resource constraints, however, most of the frameworks presented in this subsection employ data parallelism, when the neural network model is replicated to the several devices, or partition the neural network at coarser grains, i.e., at the layer level. While data parallelism requires that the memory needed by a neural network fits into a single device memory, when we partition the neural network at finer grains, we may produce partitionings with a reduced amount of communication or a large inference rate, as we show in this thesis. The frameworks may offer an automatic partitioning or the user must choose where each partition will be executed. Our algorithms offer an automatic partitioning so that the user does not need to have knowledge about computing systems to choose where to assign each part of the neural network inference execution. The target platform listed in Table 3.1 shows to which platform the frameworks were deployed. While servers may be limited by the network bandwidth, embedded or IoT devices may be limited by their memory, for instance, and the frameworks reflect the platform characteristics in their design. As most of these frameworks were designed for large servers, they do not take into account characteristics that should be considered for constrained IoT devices such as memory, communication, computation, and energy.

3.2 Approaches and Frameworks for Machine Learning on IoT

When dealing with the problem of deploying deep learning models on IoT devices, both for training and inference, two approaches are commonly used. Either the number of neurons and/or parameters of the neural network is reduced so that it fits into the memory of constrained devices and lowers resource requirements, or the neural network execution is distributed among more than one device, which is an approach that may present performance issues.

Table 3.1: Main features of general frameworks for machine learning and our approach.

Work	**Training or Inference?**	**Distributed system**	**Partitioning approaches**	**Automatic partitioning?**	**Target platform**
DistBelief [29]	Focus on training	Homogeneous	SOA	Yes	Servers
Adam [18]	Training	Homogeneous	SA	Yes	Servers
Parameter Server [68]	Both	Homogeneous	S	Yes	Servers
Bösen [118]	Training	Homogeneous	S	Yes	Servers
TensorFlow [2]	Focus on training	Heterogeneous	SO	Yes	From mobile devices to servers
SINGA [85]	Training	Homogeneous	SOP	Yes	Servers
Ako [117]	Training	Homogeneous	S	Yes	Servers
GPipe [44]	Training	Homogeneous	SO	Yes	Accelerators (GPU, TPU)
Mesh-TensorFlow [95]	Training	Homogeneous	SO	No	Servers
SOAP and FlexFlow [47]	Training	Heterogeneous	SOAP	Yes	Servers
PyTorch [87]	Training	Homogeneous	S	Yes	GPU
Our approach	Inference	Heterogeneous	OAP	Yes	IoT devices

One approach to reducing the neural network size to enable its execution on IoT devices is the Big-Little approach [26]. In this approach, a small, critical neural network is obtained from the original DNN to classify the most important classes that should be identified in real time such as the occurrence of fire in a room. For other noncritical classes, data are sent to the cloud for inference in the complete neural network. This approach depends on the cloud for the complete inference and presents some accuracy loss.

Some accuracy loss also happens in the work proposed by Leroux et al. [66], which build several neural networks with an increasing number of parameters. Their approach is called Multi-fidelity DNNs. The neurons of these neural networks are designed to match different IoT devices according to their computational resources. This design aims to satisfy the heterogeneity in the IoT. However, there is some accuracy loss for each version of the original neural network that they used. This loss may not be acceptable under some circumstances such as fault detection in critical structures of Industry 4.0 [7].

Leroux et al. [65] used a cascade of neural networks and also divided the inference into two types: the simpler neural network executes on embedded devices and the complete neural network runs on the cloud. The simpler neural network comes from an extension of the original neural network structure to get intermediate outputs. Then, if the output quality is high enough, the remaining layers do not need to be calculated, otherwise, the complete neural network executes on the cloud. The authors trade off accuracy for speed, thus, the accuracy loss is controlled in this work.

DeepIoT proposed a unified approach to compress DNNs that works for CNNs, fully connected neural networks, and RNNs [125]. The compression makes smaller dense matrices by extracting redundant neurons from the DNN. It can greatly reduce the DNN size, which also greatly reduces the execution time and energy consumption without loss of accuracy. However, as discussed in Chapter 1, even after pruning a DNN, its requirements may still prevent it from being executed on a single constrained device. Thus, this approach may not be sufficient and we focus on distributing the execution of DNNs to multiple constrained devices.

Regarding the distributed execution of neural networks to IoT devices, de Coninck et al. [25] proposed Distributed Artificial Neural Networks for the Internet of Things (DIANNE), which is an IoT-specific framework to model, train, and evaluate neural networks distributed among multiple devices. Although this platform is IoT-specific and even optimizes the inference execution of a single sample at a time for inference streaming on IoT devices, it does not provide any means for automatic partitioning of the neural network. The platform leaves this burden to the user, who needs to decide how to partition the neural network. Furthermore, the neural network architecture may be too complex for manual partitionings. The partitioning also only divides the neural network at the level of layers, in which we can assign only entire layers to a specified device. Again, this partitioning type may limit the performance and may not work for very constrained scenarios [28, 76].

Lane and Georgiev [58] proposed a mobile DNN classification engine for several sensor inference tasks. This engine executes most inference operations on the low-power DSP of smartphones, which increases resource efficiency when compared to a CPU-only approach. The low-power DSP allows for a high level of energy saving, saving up to 14 times more energy than a CPU approach. They showed that it is possible to execute DNNs in real time on DSPs, presenting a low energy and runtime overhead. Additionally, they found out that DNNs are significantly more scalable as the number of classes increases, i.e., as the number of neurons in the last layer increases.

Soto et al. [99] proposed the Complex Event Machine Learning (CEML) framework to distribute machine learning algorithms automatically for (near-)real-time, automatic, continuous evaluation at the network edge. This framework combines complex-event processing and machine learning applied to IoT. The authors acquire data from different sensors, from the same devices that process them or from local networks, pre-process data for attribute extraction, perform the training phase on IoT devices as well, analyze the learning result, and, when the training finishes, deploy the model. This process can repeat whenever the environment changes, i.e., whenever there are new data. They also implemented this framework and called it IoT Learning Agent. This implementation executes on the cloud or on embedded devices at the network edge.

When it is not possible to run an application on a single IoT device, another approach is to offload some parts of the code onto the cloud. DeepX is a hybrid approach that not only reduces the neural network size but also offloads the execution of some neural network layers onto the cloud, dynamically deciding between its local CPU, GPU, or the cloud itself [59]. However, its runtime algorithms may be too computationally heavy to be executed on more constrained devices than those used in their work, which were smartphones and GPUs, every time that the system requests an inference. Furthermore, DeepX partitions the neural network at the layer level only again and may not be able to distribute the neural network to other local devices.

Benedetto et al. [11] also used the code offloading approach in a framework that decides if some general computation should be executed locally or should be offloaded onto the cloud. The authors did not design this approach for machine learning specifically. Although the approach is interesting, as well as the fact that constrained IoT devices may prevent their runtime program to execute on such a small device, in this work, we consider a scenario in which it is not possible to send data to the cloud all the time and we have only constrained devices to perform the inference of DNNs.

Li et al. [67] proposed the opposite situation: a tool to offload deep learning executing on cloud computing onto edge computing, i.e., deep learning processing that would be first executed on the cloud can also be offloaded onto IoT gateways and other edge devices. This offload aims to improve learning performance while reducing network traffic, however, it also employs a per-layer approach.

Zhao et al. [131] proposed DeepThings, a framework for the inference distribution to resource-constrained IoT edge devices with a partitioning along the neural network data flow. They used a few devices and a high amount of memory, avoiding the use of more constrained devices such as the ones used in this thesis.

Finally, Xu et al. [122] proposed a heuristic offloading method to minimize the total transmission delay when offloading deep learning tasks to the cloud or the edge. Additionally, the authors did not want to overload edge computing nodes but want to reduce the renting cost of cloud services. They also proposed a framework to offload deep learning tasks in 5G networks. However, in their work, each task is a complete inference processing, thus, they do not perform any deep learning partitioning. As an evolution of this work, Xu et al. [123] proposed a computation offloading method for IoT on cloud-edge computing. They employed a genetic algorithm to solve the multi-objective optimization problem considering execution time and energy consumption for mobile devices.

Table 3.2 summarizes the related work in specific frameworks and approaches for IoT and machine learning. We identified the main characteristics of these frameworks related to our work and show them in the table. First, the works that prune the DNN have the disadvantage that the user needs to have experience in deep learning to obtain a simple neural network that is effective in classifying data with acceptable quality. Furthermore, even after pruning a neural network, the computational requirements of this neural network may prevent it from being executed on constrained devices. This pruning may or may not impact the DNN result accuracy, which we show in the third column of Table 3.2. Our algorithms do not prune the DNN and do not impact the result accuracy. Some works offload the execution to the cloud (fourth column of Table 3.2), however, our scenario does not allow an Internet connection all the time and, thus, we need to execute the DNN only on the IoT devices. When the framework offers a distributed execution, it is important to know how the DNN is distributed, which is shown in the fifth column of Table 3.2. The partitioning approaches may be in the Sample, Operator (O), Attribute (A), and OAP. Our algorithms can partition DNNs using the OAP strategy, which allows both per-layer partitionings and partitionings into the height, width, and depth of each layer, i.e., per neurons. Finally, the frameworks may offer an automatic partitioning or the user must choose where each partition will be executed. Our algorithms offer an automatic partitioning so that the user does not need to have knowledge about computing systems to choose where to assign each part of the neural network inference execution.

Table 3.2: Summary of machine learning and IoT frameworks discussed in the related work and our approach.

Approach	**Prune neural network?**	**Loss of accuracy?**	**Offload to the cloud?**	**Partitioning type**	**Automatic approach?**
Big-Little [26]	Yes	Yes	Yes	O	No
Cascade [65]	Yes	Trade-off	Yes	O	No
DIANNE [25]	No	No	No	O	No
Lane and Georgiev [58]	No	No	Yes	along the layers (A)	No
CEML [99]	No	No	No	S	Yes
DeepX [59]	Yes	Yes	Yes	O	Yes
DeepIoT [125]	Yes	No	No	N/a*	Yes
Li, Ota, and Dong [67]	No	No	Yes	O	Yes
DeepThings [131]	No	No	No	along the layers (A)	Yes
Multifidelity [66]	Yes	Yes	No	N/a	No
Benedetto et al. [11]	No	No	Yes	OAP	Yes
Heuristic Offloading Method [122, 123]	No	No	Yes, and the edge	N/a	Yes
Our approach	No	No	No	OAP	Yes

* Not applicable.

3.3 Partitioning Algorithms

The computation distribution may affect the inference performance. One solution to avoid this issue is to use automatic, general-purpose partitioning algorithms to define a profitable partitioning for the DNN inference. One of the frameworks to do that is SCOTCH, which is a project and software package designed mainly for graph partitioning and static mapping [21]. It implements the Dual Recursive Bipartitioning mapping algorithm and other graph bipartitioning heuristics such as Fiduccia-Mattheyses [31]. The goal of this framework is to balance the computational load while minimizing communication costs. SCOTCH uses two graphs to perform mapping: one is the source graph, which represents the computation that SCOTCH will map, and a target graph, i.e., a graph that contains information about the devices' computational performance (vertices) and communication configuration (edges). In the source graph, vertices represent computation while edges are data transfers; it works like a dataflow graph. However, as SCOTCH was not designed for constrained devices, there is no memory constraint treatment and it may produce invalid partitionings. Additionally, this framework cannot factor redundant edges out, which are edges that represent the same data transfer to the same partition, a situation that often happens in partitioned neural networks.

Kernighan and Lin originally proposed an algorithm [51] to partition graphs that has a large application in distributed systems [6, 16, 119]. First, their heuristic randomly partitions a graph that may represent the computation of some application. Then, the algorithm calculates the communication cost for this random initial partitioning and tries to improve it by swapping vertices from different partitions and calculating the gain or loss in performing this swap. The best swap operation in each iteration is chosen and its respective vertices are locked for the next iterations and cannot be selected anymore until every pair is selected. When every pair is selected, the whole process may be repeated while improvements are made so that it is possible to achieve a near-optimal partitioning, according to the authors. This algorithm also accounts for partition balance in the hope of achieving an adequate performance while minimizing communication.

Another framework is METIS, an open-source library and software from the University of Minnesota that partitions large graphs and meshes and also computes orderings of sparse matrices [49]. This framework employs an algorithm that partitions graphs in a multilevel way, i.e., first, the algorithm gradually groups the graph vertices based on their adjacency until the graph presents only hundreds of vertices. Then, the algorithm applies a partitioning algorithm such as Kernighan and Lin [51] to the small graph and, finally, returns to the original graph also in a multilevel way, performing refinements with the vertices that present communication to other vertices in different partitions during this return. METIS also reduces communication while balances all the other constraints, which may be memory and computational load, for instance. However, METIS does not present an appropriate treatment of memory constraints either and, thus, may produce invalid partitionings. Additionally, METIS cannot factor redundant edges out either.

A multilevel Kernighan and Lin approach was developed to partition software components in mobile cloud computing aiming to achieve the near-optimal solutions of Kernighan and Lin and the fast execution time of METIS [113]. This solution does not require the partitions to be balanced and takes into account the system heterogeneity and local devices. However, it does not consider memory constraints or redundant edges. Furthermore, the aim is to minimize communication, which may not yield the best result for other objective functions such as inference rate.

All the general-purpose approaches discussed so far in this subsection are edge-cut partitionings, i.e., the algorithms partition the graph vertices into disjoint subsets [37]. Another strategy to general-purpose graph partitioning is vertex-cut partitioning, which partitions the graph edges into disjoint subsets, while the vertices may be replicated among the partitions. Rahimian et al. [90] proposed JA-BE-JA-VC, an algorithm that performs vertex-cut partitioning. Their approach attempts to balance the partitioning aiming to satisfy memory constraints. The main disadvantage of this approach is that it needs vertex replicas, that is, computation replicas, and synchronization, which may involve more communication. When we consider constrained IoT devices and their computational performance, the computation replicas may decrease the inference rate of neural networks to a value that does not comply with the application requirements.

LeBeane et al. [62] considered the heterogeneity in the processing nodes of modern data centers to modify five online data ingress strategies. They aimed to optimize the execution in heterogeneous data centers, improving the runtime of popular machine learning and data mining applications. They also considered partitioning algorithms that perform edge cuts and vertex cuts. When an application needs to have barriers for synchronization between different nodes, the authors provide data proportionally to the computational performance of the nodes so that they have similar execution times when processing them. They also use proportions to define the amount of memory that each partition requires, which does not impose a strict limit on the amount of memory of the partitions and may lead to invalid partitionings. Additionally, they do not factor redundant edges out of the cost computation.

The general-purpose partitioning algorithms are present in Table 3.3. The memory treatment is important in constrained IoT devices because the memory that they usually provide is small. In this thesis, the RAM of the devices started at 16 KiB. Thus, our proposed algorithms treat memory restrictions, unlike most partitioning algorithms in this table. Therefore, these algorithms may produce invalid partitionings and we showed that in the results. Partition balance causes all the partitions to have similar sizes, usually resulting in good performance through load balancing. Our algorithms allow unbalanced partitionings and our results show that unbalanced partitionings may lead not only to reduced communication but also a larger inference rate. Additionally, our algorithms are the only ones that can factor redundant edges out of the cost computation during the whole algorithm execution. This feature allows for a precise amount of communication and also may reduce the final amount of communication because our algorithms try to reduce the real amount of communication without any redundant data transfer. While the algorithms in Table 3.3 reduce communication or balance partitions as their objective function, our algorithms can reduce communication or maximize the inference rate. The inference rate maximization objective function allows our algorithms to produce partitionings that optimize throughput performance directly, producing partitionings with a better performance when compared to the state-of-the-art algorithm, which we showed in the results. Finally, our algorithms are the only ones that account for the memory required by shared parameters and biases of CNNs adequately because we considered CNNs in their design. This feature allows the algorithms to indicate a precise amount of memory for each partition and, thus, allows us to use devices with a smaller amount of memory. Nonetheless, our proposed algorithms can partition any dataflow graph and not only CNNs.

3.4 Final Remarks

In this chapter, we presented the literature review in the three topics that are relevant to our research. Most general frameworks for machine learning focus on the training phase of neural networks, homogeneous distributed systems, data parallelism, and larger servers. However, when we consider constrained IoT devices, we need to focus on the inference phase of neural networks because these devices may not even be able to execute the training phase. Additionally, IoT devices are usually heterogeneous, thus, we have to consider heterogeneous devices when distributing the execution of a neural network. Data parallelism is used for training, however, constrained IoT devices may benefit from model parallelism that partitions the neural network at finer grains, as we show in this thesis. Finally, as these frameworks focus on larger servers, they do not consider memory constraints, however, constrained IoT devices must consider memory limitations so that we enable the distributed execution of neural networks on these devices.

When we considered approaches and frameworks for machine learning on IoT, there are several approaches. Some frameworks prune the neural network so that it requires fewer resources, losing some accuracy or not. Most frameworks for machine learning on IoT offload part of the neural network execution to the cloud and partition the neural network using the operator dimension, which corresponds to the per-layer partitioning. However, when we offload the execution to the cloud, we may increase the latency or reduce the inference rate of the neural network execution. Thus, this approach may not be the most adequate for the distributed execution of neural networks on IoT devices. Additionally, using only the per-layer partitioning may limit the memory constraint of the devices employed in the partitioning, i.e., we may not use devices that are more memory-constrained than the amount of memory of the layer that most requires memory. Furthermore, this approach leads to suboptimal partitionings when the objective function is to reduce communication or increase the inference rate. We show these two situations in this work. Finally, some frameworks do not offer an automatic approach, requiring the user to set the parameters for the execution. For instance, the user must choose how the neural network is pruned or which layer is executed on the cloud. However, the user may not be experienced enough to make these decisions, thus, we should offer automatic frameworks so that the user gets the most profitable execution that the framework can provide.

Finally, most general-purpose partitioning algorithms do not treat memory constraints, thus, they can provide invalid partitionings for constrained IoT devices. Additionally, most partitioning algorithms balance the partitions, however, in this work, we show that, if we do not enforce partition balance, the partitionings can achieve a larger inference rate or a reduced amount of communication when compared to approaches and algorithms that balance partitions. Most partitioning algorithms reduce communication as their objective function and balance partitionings aiming at better execution performance, however, this approach does not lead to the partitioning that achieves the smallest amount of communication neither the largest inference rate. In this thesis, we use communication minimization and inference rate maximization as objective functions, directly optimizing partitionings for better execution performance, and we show that our algorithms achieve better results than the literature algorithms. The last two characteristics that we considered when partitioning neural networks are the elimination of the redundant edges out of the cost computation and the adequate count of shared parameters and biases. Although redundant edges often occur in the neural network partitioning, they may be present in any dataflow graph partitioning. Finally, the adequate count of shared parameters and biases of neural networks allow for the use of more constrained devices because the memory required by them is counted only once per partition that requires them. The next chapters present our proposals to deal with the issues encountered in this literature review.

Table 3.3: Summary of the partitioning algorithms discussed in the related work and our approach.

Approach	**Memory constraints?**	**Partition balance?**	**Factor redundant edges out?**	**Objective function**	**Adequate account of shared parameters?**
SCOTCH [21]	No	With some load unbalancing	No	Minimize communication	No
KL [51]	Yes	With some unbalancing	No	Minimize communication	No
METIS [49]	No	With some unbalancing in the constraints	No	Minimize communication	No
Multilevel KL [113]	Yes	No	No	Minimize communication	No
JA-BE-JA-VC [90]	No	Yes	No	Balance partitions	No
Our approach	Yes	No	Yes	Maximize inference rate or minimize communication	Yes

Chapter 4

Partitioning the LeNet Convolutional Neural Network for Communication Minimization

In this chapter, we explore the Kernighan-and-Lin-based Partitioning (KLP), an algorithm that we recently proposed [28] to automatically partition neural networks for distributed execution on hardware-constrained IoT devices. When partitioning the neural network, reducing communication is important because it reduces the amount of power consumed on radio operations and the amount of interference on the wireless medium. Thus, in this chapter, we use KLP to minimize the amount of communication among IoT devices when partitioning the LeNet model and show that the partitioning found by KLP requires up to 4.5 times less communication than the partitionings offered by other approaches such as the one adopted by popular machine learning frameworks such as TensorFlow, DIANNE, and DeepX.

4.1 Proposed Kernighan-and-Lin-based Partitioning

This section is based on both one of our conference papers [28] and one of our journal papers [76] for a complete view of KLP. The KLP algorithm is inspired by the Kernighan and Lin's heuristic, which attempts to find a better solution than its initial partitioning by swapping vertices from different partitions. Kernighan and Lin's algorithm avoids some local minimum solutions by allowing swaps that produce a partitioning that is worse than the previous one. This situation can happen if such a swap is the best valid operation at some point in the algorithm [76].

Our initial goal was to use the original Kernighan and Lin algorithm to partition neural networks [28]. To this end, the neural network was modeled as a dataflow graph in which vertices represent input data, operations, or output data while edges represent data transfers between vertices. This same approach is used in SCOTCH and METIS [76]. KLP also receives a target graph, which contains information about the devices (the number of them in the system, computational and communication performance, and system topology) in a way similar to SCOTCH.

To work with more than two partitions, the original Kernighan and Lin's heuristic repeatedly applies its two-partition algorithm to pairs of subsets of the partitions to try to achieve a pairwise optimal result. This approach may fall into local minima, thus, we avoid some of these local minima by allowing the algorithm to work with multiple partitions by considering swaps between any partitions during the whole algorithm.

The swap operation in the original Kernighan and Lin's algorithm also led to other local minima because it was limited to produce partitions with the same number of vertices of the initial partitioning. This limitation caused the final solution to be heavily dependent on the initial partitioning configuration [28]. To solve this limitation, we introduced a "move" operation, in which the algorithm considers moving a single vertex from one partition to another, without requiring another vertex from the destination partition to move back to the source partition of the first vertex.

In the case of the communication reduction objective (or cost) function, this move operation allows all vertices to be moved to a single partition and, thus, the communication would be zero, which is the best result for this objective function [76]. However, the dataflow graph containing the neural network model may not fit into a single memory-constrained IoT device due to memory limitations. Hence, we added memory requirements for each vertex in the dataflow graph and modified the graph header to contain information about the shared parameters and biases for CNNs. Furthermore, we designed the KLP algorithm to consider the amount of memory of the devices as a restriction for the algorithm, i.e., the operations cannot be performed if there is not sufficient memory in the partitions. This feature allowed the initial partitioning and any partitioning in the middle and at the end of the algorithm to be unbalanced. At this point, unlike SCOTCH and METIS, our algorithm KLP could always produce valid partitionings. Additionally, KLP shows the amount of memory that each partition requires, which can help memory and partitioning analyses.

We also propose a feature to factor redundant edges out of the cost computation in the KLP algorithm. Redundant edges represent the transfer of the same data between partitions, which happens when there are multiple edges from one vertex to vertices that are assigned to the same partition. Neither SCOTCH nor METIS considers redundant edges, i.e., they show and partition the graph using information about communications that leads to a much larger value than the real value that must be indeed transferred.

The pseudocode of KLP is listed in Algorithm 1. The first step of the algorithm is to initialize *bestP*, which contains the best partitioning found so far, with the desired initial partitioning. After that, the algorithm runs in *epochs*, which are the iterations of the outer loop (Lines 3–26). This outer loop runs some epochs until the best partitioning found so far is no longer improved after the execution of a full epoch. For each epoch, the algorithm first unlocks all the vertices (Line 5) and initializes the current partitioning with the best partitioning found so far (Line 6). After that, the inner loop, which is a *step* of the epoch, searches for a better partitioning and updates the current and best partitionings (Lines 7–25). In each step, KLP performs a local search to identify which operation (swap or move) according to the objective function is better for the current partitioning (Line 8). This function is further detailed in Algorithm 2. Then, the best operation chosen in this function is applied to the current partitioning and the corresponding vertices are locked (Lines 10–15), i.e., they are not eligible to be chosen until the current epoch finishes. The best operation in each step may worsen the current partitioning because, if there are no operations that improve the partitioning, then the best operation is the one that increases the cost minimally. If there are no valid operations, the current step and the epoch finish (Lines 16–18). This happens when all vertices are locked or when there are unlocked vertices, but they cannot be moved or swapped due to memory constraints, i.e., if they are moved or swapped, then the partitioning becomes invalid. Whenever the current partitioning is updated, its cost is compared to the best partitioning cost (Line 21) and, if the current partitioning cost is better, then the best partitioning is updated and the *bestImproved* flag is set to *true* so that the algorithm runs another epoch to attempt a better partitioning. Figure 4.1 shows the flowchart related to Algorithm 1 and represents a general view of our proposal (KLP).

Algorithm 1 KLP algorithm [76].

```
1: function KLP($initialPartitioning, sourceGraph$)
2:     $bestP \leftarrow initialPartitioning$;
3:     repeat
4:         $bestImproved \leftarrow false$;
5:         $unlockAllNodes()$;
6:         $currentP \leftarrow bestP$;
7:         while there are unlocked nodes do
8:             $op \leftarrow findBestValidOperation(currentP)$;
9:             /* Perform the operation */
10:            if $op.type = SWAP$ then
11:                $currentP.swap(op.v1, op.v2)$;
12:                $lockVertex(op.v1)$; $lockVertex(op.v2)$;
13:            else if $op.type = MOVE$ then
14:                $currentP.move(op.v, op.targetPartition)$;
15:                $lockVertex(op.v)$;
16:            else if $op.type = INVALID$ then
17:                /* No valid operations */
18:                break;
19:            end if
20:            /* Update the best partitioning */
21:            if $currentP.cost() < bestP.cost()$ then
22:                $bestP \leftarrow currentP$;
23:                $bestImproved \leftarrow true$;
24:            end if
25:        end while
26:    until $bestImproved = false$
27:    return $bestP$;
28: end function
```

Algorithm 2 shows the pseudocode for the *findBestValidOperation()* function. First, the algorithm initializes the *op* type with “invalid” (Line 2). If this function returns this value, then there are no operations that maintain the partitioning valid. After that, a loop runs through all the unlocked vertices searching for the best valid operation for each vertex in this set (Lines 3–32). For each vertex, the algorithm searches for the best move for it (Lines 4–15) and the best swap using this vertex (Lines 16–31). In the best move search, a loop runs through all the partitions (Lines 5–15). In this loop, the algorithm changes the current partition of the vertex being analyzed (Line 6), checks if the partitioning remains valid (Line 7), calculates the new cost of this partitioning according to the objective function (Line 8), checks if this new partitioning has a better cost than the current one (larger inference rate or fewer communications) or if no valid operation was found so far (Line 9), and updates, if necessary, *bestCost* with the better value and *op* with the move operation and the corresponding vertex and partition (Lines 10–12). In the best swap search, another loop runs through all the unlocked vertices (Lines 16–31). In this loop, the algorithm changes the current partition of both vertices that are being analyzed (Lines 17–19), checks if the partitioning remains valid (Line 20), calculates the new cost of this partitioning according to the objective function (Line 21), checks if this new partitioning has a better cost than the current one (larger inference rate or fewer communications) or if no valid operation was found so far (Line 22), and updates, if necessary, *bestCost* with the better value and *op* with the swap operation and the corresponding vertices and partitions (Lines 23–25). At the end of the loop, the original partitions of the vertices being analyzed are restored to proceed with the swap search (Lines 28–29). After the outer loop finishes, the best operation found in this function is returned to KLP (or the “invalid” operation, if no valid operations were found).

4.2 Methods and Materials

In this section, we present the LeNet model, the device characteristics that we used in the experiments, and the experimental setup.

4.2.1 LeNet Neural Network Model

In this thesis, we used the original LeNet-5 DNN architecture [63] as a case study. This DNN was first applied to recognize handwritten digits in images, however, it can be applied to other kinds of recognition as well [27, 83, 109]. Although LeNet is the first successful CNN, its lightweight model is suitable for constrained IoT devices. In this and the next chapter, we show that even a lightweight model such as LeNet requires partitioning to execute on constrained IoT devices. Furthermore, several works have been recently published using LeNet [3, 27, 71, 83, 109], causing this CNN to be still relevant nowadays.

To validate KLP, we modeled the LeNet architecture with 765 vertices [28]. The LeNet neurons were grouped into vertices. The neurons in the same position of width and height but different positions in the depth of the LeNet convolution and pooling layers were grouped into one vertex because two neurons in these layers in the same position of width and height but different positions in depth present the same communication pattern. Thus, a partitioning algorithm would tend to assign these vertices to the same partition. For the input layer, every four neurons (width and height of size two) were grouped to form one vertex. The reason to group these adjacent vertices is that they present communication to adjacent neurons in the next layer and the algorithm tends to assign these vertices to the same partition anyway. This grouping of the first layer was used until the fourth layer (convolution layer), also grouping all neurons in depth to form only one vertex (depth of size one, as explained above). In the fifth layer, only the depth vertices were grouped and, after this layer, there was no grouping at all. This grouping

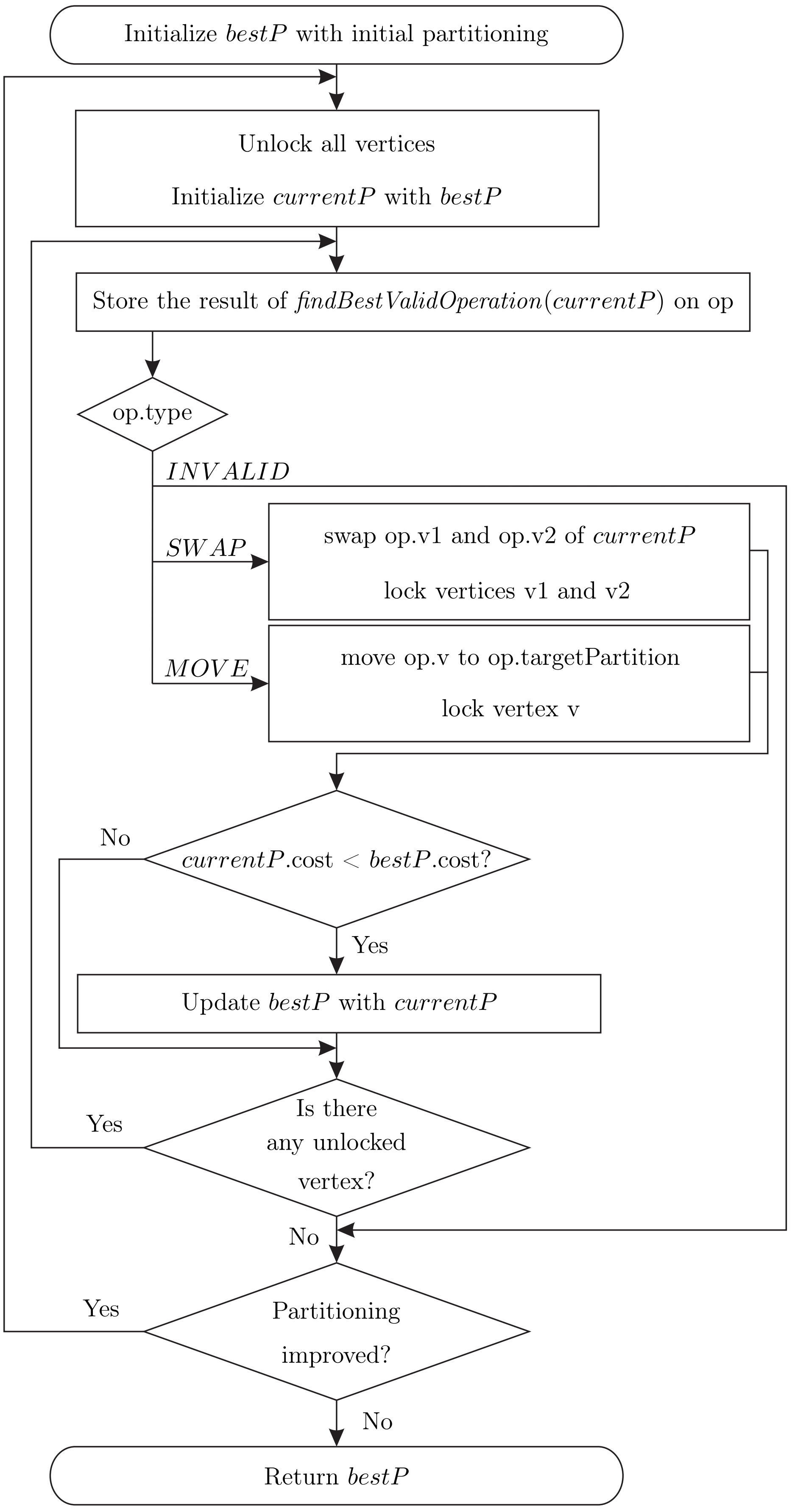

Initialize *bestP* with initial partitioning
Unlock all vertices
Initialize *currentP* with *bestP*
Store the result of *findBestValidOperation*(*currentP*) on op
op.type
INVALID
SWAP
swap op.v1 and op.v2 of *currentP*
lock vertices v1 and v2
MOVE
move op.v to op.targetPartition
lock vertex v
No
currentP.cost < *bestP*.cost?
Yes
Update *bestP* with *currentP*
Yes
Is there any unlocked vertex?
No
Yes
Partitioning improved?
No
Return *bestP*

Figure 4.1: Flowchart of Algorithm 1 [76].

Algorithm 2 *findBestValidOperation* function [76].

```
function findBestValidOperation($currentP$)
    $op.type \leftarrow INVALID$;
    for $i \leftarrow unlocked.first$ to $unlocked.last$ do
        $originalPi \leftarrow currentP[i]$;
        for $p \leftarrow 1$ to $numberOfPartitions$ do
            $currentP[i] \leftarrow p$;
            if $validPartitioning(currentP) = true$ then
                $cost \leftarrow computeCost(currentP)$;
                if $cost < currentP.cost$ or $op.type = INVALID$ then
                    $bestCost \leftarrow cost$;
                    $op \leftarrow moveOp(i, p)$;
                    $op.type \leftarrow MOVE$;
                end if
            end if
        end for
        for $j \leftarrow unlocked.first$ to $unlocked.last$ do
            $originalPj \leftarrow currentP[j]$;
            $currentP[i] \leftarrow originalPj$;
            $currentP[j] \leftarrow originalPi$;
            if $validPartitioning(currentP) = true$ then
                $cost \leftarrow computeCost(currentP)$;
                if $cost < currentP.cost$ or $op.type = INVALID$ then
                    $bestCost \leftarrow cost$;
                    $op \leftarrow swapOp(i, originalPi, j, originalPj)$;
                    $op.type \leftarrow SWAP$;
                end if
                /* Restore current partitioning */
                $currentP[i] \leftarrow originalPi$;
                $currentP[j] \leftarrow originalPj$;
            end if
        end for
    end for
    return $op$;
end function
```

led to 256 vertices in the input layer, 196 vertices in the second layer (convolution layer), 49 vertices in the third layer (pooling), 25 vertices in the fourth (convolution) and fifth (pooling) layers, and 120, 84, and 10 vertices in the fully connected layers.

Figure 4.2 shows how the LeNet neurons were grouped to form the dataflow graph with the following per-layer data: the number of vertices in height, width, and depth, the layer type, the amount of transferred data in bytes required by each edge in each layer, and the number of vertices in each layer. In this figure, the cubes represent the original LeNet neurons and the circles and ellipses represent the dataflow graph vertices.

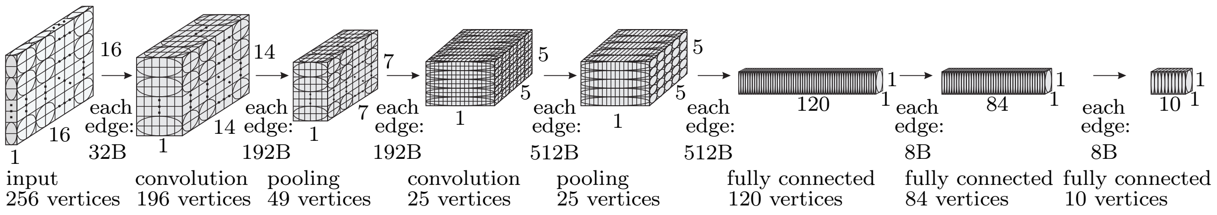


Figure 4.2: LeNet architecture and vertex granularity used in our algorithm. Each cube stands for a CNN neuron while each circle or ellipse is a vertex in the source graph. Edges represent data transfers and are labeled with the number of bytes per inference that each edge must transfer [28] (modified).

We grouped the LeNet neurons to reduce the dataflow graph size and, thus, to reduce the partitioning execution time and allow us to perform several experiments in a short time frame. This strategy constrains the partitioning algorithm since it cannot place some of the original neurons (the ones that were grouped into the same vertex) in different partitions. Nonetheless, our automatic partitioning algorithm could still find partitionings that were superior to the ones offered by state-of-the-art neural network partitioning frameworks.

In the convolution layers, there is a set of shared parameters and biases for each layer, which is shared among all the neurons of that layer. For the pooling layers, in the LeNet version that we used in this thesis, there is a set of biases and trainable coefficients for each layer, which is also shared among all the neurons of that layer [63]. In the fully connected layers, each neuron has its own set of parameters and bias. Table 4.1 shows the amount of memory required by each vertex in each layer, the amount of memory required by the shared parameters and biases for each layer, the filter size and stride at each convolution and pooling layers, and the depth size of each layer. While the first characteristic depends on how we grouped the neurons, the other characteristics are inherent to the original LeNet-5 DNN architecture [63] used in this and the other chapters. In this table and hereafter, the convolution layers are represented by Cn, the pooling layers are represented by Pn, and the fully connected layers, by FCn, in which n is the position of each layer in the neural network.

4.2.2 Setups and Experiments

Table 4.2 summarizes the IoT setups used in the experiments presented in this chapter. The manual partitionings in Subsection 4.3.1 balance the number of vertices and, thus, do not consider memory restrictions either employ any device setup for the algorithm. These approaches partitioned the LeNet model into four and six equally sized partitions. The homogeneous partitionings employed the ARM-based STM32F469xx processor, which can provide 388 KiB of RAM [102]. This device is in class *others* of constrained devices

Table 4.1: Characteristics of the LeNet model used in this chapter.

Characteristic	**input**	**C1**	**P2**	**C3**	**P4**	**FC5**	**FC6**	**FC7**
Memory per vertex (B)	32	192	192	512	128	12864	3904	3440
Memory of shared parameters and biases per layer (B)	0	1248	96	12128	256	0	0	0
Filter size	N/a*	5x5	2x2	5x5	2x2	N/a	N/a	N/a
Stride	N/a	1	2	1	2	N/a	N/a	N/a
Depth size	1	6	6	16	16	120	84	10

* Not applicable.

that we showed in Chapter 2. These experiments partitioned the LeNet model into four and six partitions. The heterogeneous experiments employed setups that were inspired by the configuration of the STM32F469xx processor. For instance, in the four-partition experiment, we set two devices with the same amount of memory of the STM32F469xx and two devices with half of this amount of memory. Both devices are in class D2 of constrained devices. In the three-partition experiment, we set one device with the same amount of memory of the STM32F469xx, one device with half of this amount of memory, and one device with 1.5 times this amount of memory. In this setup, the device with 1.5 times the amount of memory of the STM32F469xx is in class *others* of constrained devices, which contains the least constrained devices, and the other devices are in class D2. Our automatic partitioning tool (KLP) was executed 30 times for each experiment, each one starting with a different random-generated partitioning. It is worth noting that we assume a homogeneous communication performance between all the devices, a constant communication performance during the whole algorithm, and no limits for the amount of transferred data per second between each device.

4.2.3 Types of Input Layers in the Experiments

For each setup in Table 4.2, except for the manual partitionings, we performed two types of experiments with the input layers:

- the **free-input-layer** experiments, in which all the LeNet model vertices were free to be swapped or moved; and
- the **locked-input-layer** experiments, in which the LeNet input layer vertices were initially assigned to the same device and, then, they were locked, i.e., the input layer vertices could not be swapped or moved during the whole algorithm.

The free-input-layer experiments allow all the vertices to freely move from one partition to the others, including the input layer vertices. These experiments represent situations in which the device that produces the input data cannot process any part of the neural network and, thus, must send its data to nearby devices. In this case, we would have to add more communication to send the input data (the LeNet input layer) from the

Table 4.2: Setups and experiments.

Type of experiment	**Number of devices allowed to be used in the experiments**		**Device model**	**Device amount of RAM (KiB)**
Manual	4 6		N/a*	N/a
Homogeneous	4 6		STM32F469xx [102]	388
Heterogeneous	3	1	STM32F469xx	388
		1	0.5× STM32F469xx	194
		1	1.5× STM32F469xx	582
	4	2	STM32F469xx	388
		2	0.5× STM32F469xx	194

* Not applicable.

device that contains these data to the devices chosen by our algorithms. However, as the increased amount of transferred data involved in sending the input data to nearby devices is fixed, it does not need to be shown here. On the other hand, the locked-input-layer experiments represent situations in which the device that produces the input data can also perform some processing, therefore, no additional cost is involved in this case.

4.3 Experimental Results

This section presents the results for the manual partitionings and the automatic partitionings produced by the proposed KLP. First, the cost of four manual partitionings considering four and six partitions are discussed. Then, five automatic partitioning experiments using homogeneous hardware and KLP are shown. After that, two heterogeneous configurations were employed to perform the other four automatic partitioning experiments with KLP. We implemented KLP[1] using C++ and executed the experiments on Linux-based operating systems.

4.3.1 Manual Partitionings

We considered four manual partitionings:

- **per layers:** the model is sliced vertically and vertices belonging to the same layer are assigned to the same partition. This approach is equivalent to the approach of popular machine learning frameworks such as TensorFlow, DIANNE, and DeepX;
- **per lines:** the model is sliced horizontally and vertices belonging to the same slice in each layer are assigned to the same partition;

[1]`https://bitbucket.org/FabiolaOliveira/mdn2pciot`

- **per quadrants or niches:** the model layers are divided into quadrants or niches and vertices belonging to the same quadrant/niche in each layer are assigned to the same partition; and
- **hybrid partitioning mixing layers and lines or niches:** for four partitions, all the layers are divided into two lines; the first four layers of LeNet were assigned to two devices and the last four layers were assigned to the other two devices. For six partitions, the first four layers were divided into four niches and assigned to four partitions, the fifth layer was assigned to partition four, and the last layers were divided into lines and assigned to partitions four and five.

The manual partitionings balance the number of vertices and, thus, do not consider memory restrictions either employ any device setup. These approaches partitioned the LeNet model into four and six equally sized partitions. Figure 4.3 illustrates how the LeNet vertices were assigned to the partitions for each approach: per layers, per lines, per quadrants, and hybrid approach for four partitions and hybrid approach for six partitions, as it was slightly different from the hybrid approach with four partitions, and their respective communication costs. We can visualize the neural network partitioning in two dimensions because, in each layer, we modeled at least one dimension as size one. In the convolution and pooling layers, we grouped neurons in the same position of height and width but different positions in the depth into one vertex so that the depth had size one and, in the fully connected layers, both the width and height had size one. Thus, we modeled the convolution and pooling layers as squares and the fully connected layers as column vectors but broke them into a smaller size so that their height has the same dimension of the squares, when necessary.

Figure 4.4 presents the results for all the approaches and the different number of partitions. The worst result is when LeNet is partitioned across its layers, which is the same approach adopted by DIANNE, DeepX, and TensorFlow. To partition the first CNN layers into lines, quadrants, or niches reduces the communication cost as some of the communication in the first layers (convolution and pooling) is suppressed by the use of the same partition in the same layer portion. By analyzing these patterns, the last performed partitioning was a mix of the per-line partitioning with the per-layer partitioning. In the hybrid approach, the first half of the LeNet layers is partitioned with the per-line approach and the other half is partitioned using the per-layer approach. This partitioning leads to the best result without the use of any partitioning tool for both partitionings into four and six partitions: it achieved a result 2.10 times better for four partitions while it improved 2.03 times for six partitions, showing that a hybrid approach may be necessary to result in partitionings with less communication. With the manual approaches, we can see that even simple strategies such as the per-line and per-quadrant partitionings lead to better results than the approach of popular machine learning frameworks, which is the per-layer approach.

4.3.2 Automatic Partitionings with Homogeneous Setups

In this subsection, we show the results of five experiments that used our KLP algorithm. First, LeNet was partitioned into four partitions without considering any memory restrictions and, thus, performing only pair swaps so that all vertices would not end up in a single partition, as explained in Section 4.1. The second and third experiments partitioned LeNet with the free input layer among four and six homogeneous devices using KLP as

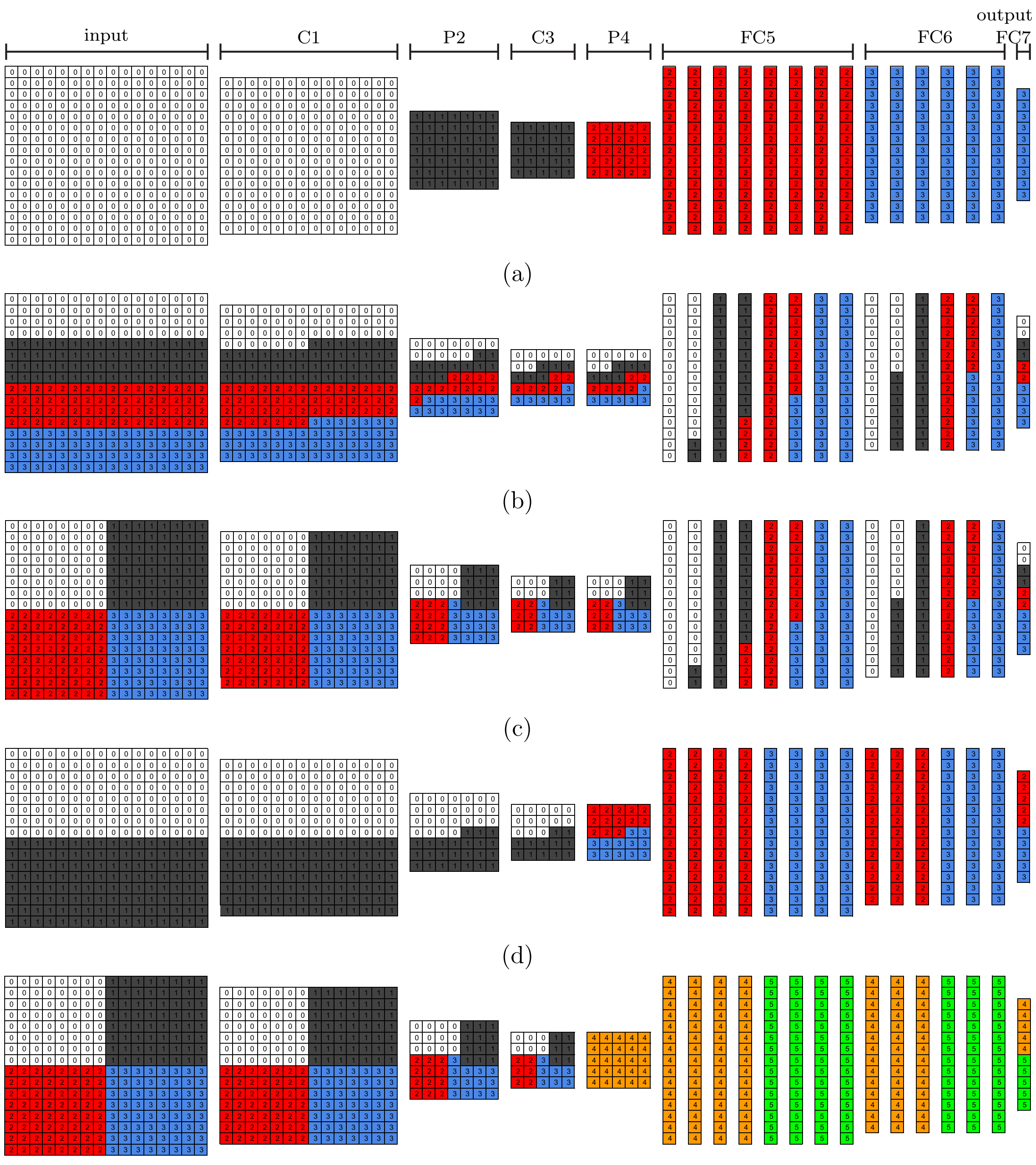


Figure 4.3: LeNet manual partitionings and their communication costs in bytes transferred between partitions per inference: (a) four partitions, per layers: 50.2 KiB; (b) four partitions, per lines: 31.5 KiB; (c) four partitions, per quadrants: 30.8 KiB; (d) four partitions, hybrid: 23.8 KiB; and (e) six partitions, hybrid: 30.8 KiB [28] (modified).

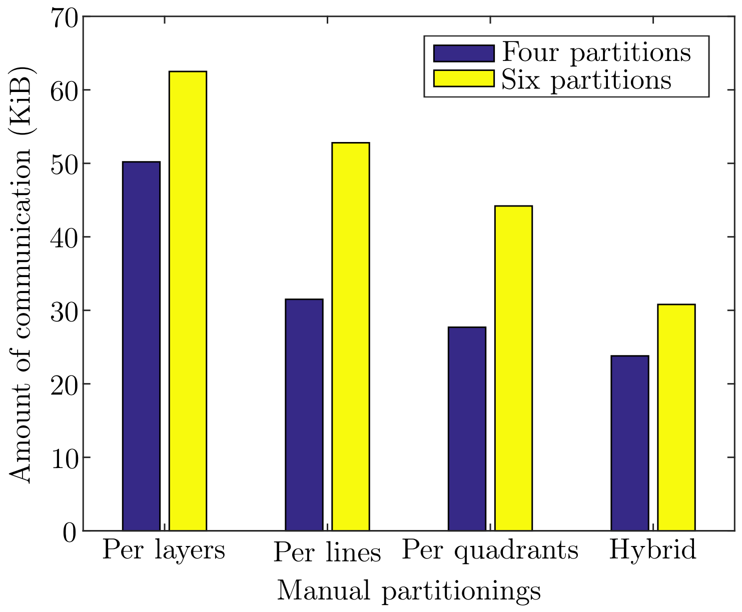


Figure 4.4: Amount of communication for the manual partitionings. The worst result is when LeNet is partitioned across its layers, which is the same approach adopted by popular machine learning frameworks such as DIANNE, DeepX, and TensorFlow. The hybrid partitioning leads to the best result without the use of any partitioning tool.

explained in Section 4.2. After that, the input layer of LeNet was forced to stay in the same partition during the whole algorithm execution to simulate the situation in which the input image is collected by a camera in a single device, for instance. Thus, the fourth and fifth experiments partitioned LeNet with the locked input layer among four and six homogeneous devices using our KLP algorithm. The configuration of the homogeneous devices was based on the STM32F469xx Advanced Reduced Instruction Set Computer (RISC) Machine (ARM) Cortex-M4 32-bit MCU+FPU.

Figure 4.5 shows the minimum, median, and maximum values achieved by KLP for each experiment and the respective best manual partitioning communication cost achieved in Subsection 4.3.1. In this figure, STM32 stands for the STM32F469xx device model. For the free-input-layer experiments, the best manual partitioning was the hybrid approach, which led to the lowest amount of communication in the previous subsection. For the locked-input-layer experiments, we have to compare the results of our KLP algorithm to some partitioning that also has its input layer entirely assigned to only one device. In the manual partitionings proposed in the previous subsection, the only approach that respects this condition is the per-layer partitioning, which assigns entire layers to devices. Thus, we can see that, for all the experiments, the median of all KLP executions starting from a random-generated partitioning was lower than the best manual partitioning method for each input-layer experiment type. Only in the first experiment, which did not consider any memory restrictions and, thus, the experiment balanced the partitions so that they contained the same number of vertices, the maximum amount of communication in 30 executions of KLP was 1.27 times higher than the hybrid manual partitioning. On the other hand, the minimum amount of communication in 30 executions for all the three experiments with four partitions was 1.60, 1.78, and 3.37 times lower than the hybrid manual partitioning communication cost, respectively for the experiment without memory restrictions, which balances the number of vertices among the partitions, the free-input-layer experiment, and the locked-input-layer experiment. These experiments show that balancing the number of vertices may not be the best approach to reduce communication between devices because the experiments that perform this approach, which are all the manual partitionings and the first experiment with KLP, did not achieve the lowest amount of communication.

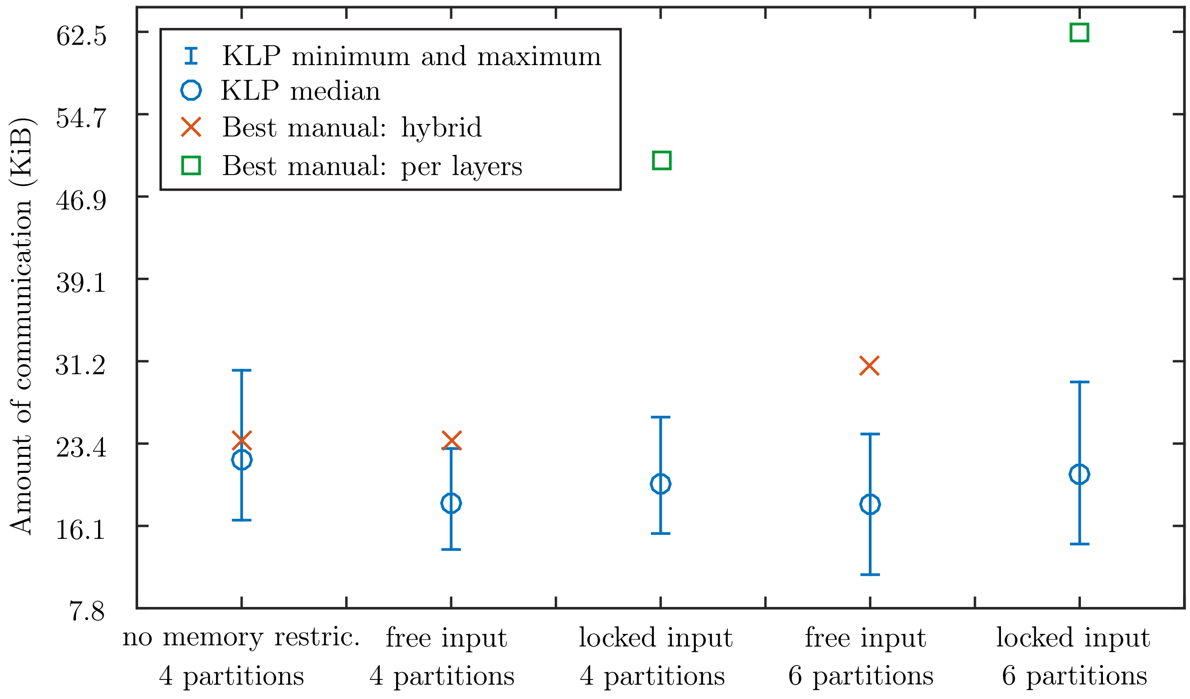


Figure 4.5: Amount of communication for the homogeneous partitionings using KLP compared to the best manual partitionings of Subsection 4.3.1 [28] (modified).

In the six-partition experiments, even the maximum amount of communication achieved by KLP in 30 executions was lower than the manual partitioning result. The minimum amount of communication achieved by KLP in 30 executions was 2.80 times and 4.50 times lower than the best manual partitioning for the free-input-layer and the locked-input-layer experiments, respectively. The same approach of the four-partition experiments was applied here: the best manual partitioning when the input was free is the hybrid partitioning while, when the input was locked, the per-layer partitioning is the best manual partitioning. The better results for the six-partition experiments are due to the larger number of partitions used: the algorithm had a larger search space, i.e., more pairs were considered to reduce communication at each epoch. The results for the six-partition experiments show that KLP can achieve a lower amount of communication, in fact, the lowest amount of communication among all the experiments in this subsection, even when more devices are provided. Providing more devices when the approach is to balance the size of the partitions increases the communication cost as we can see by the communication cost of the manual partitionings for the six-partition experiments.

The results in this subsection show that applying a partitioning algorithm to find lower costs than the ones found by manual partitionings is effective in achieving an execution with less communication in the neural network inference on constrained IoT devices. Furthermore, these manual partitionings include the per-layer partitioning, which corresponds to the approach of popular machine learning frameworks such as TensorFlow, DIANNE, and DeepX.

4.3.3 Visual Analysis for the Homogeneous Partitionings

Figure 4.6 shows the best partitioning achieved by KLP for the four-partition experiment without considering memory restrictions. In this and the next figures, partition zero is represented by the white color, one is represented by dark grey, two, by red, three, by blue, four, by orange, and five, by green. KLP used all the four partitions and did not divide the niches used in the first layers homogeneously because the last three layers used almost only partitions one and two and, when memory is not considered, KLP attempts to maintain partition balance. Between the convolution layer C1 and the pooling layer P2, communication is zero because the algorithm assigns the vertices perfectly, i.e., KLP assigns the vertices of C1 and P2 that present communication to each other to the same device. The same situation happens between the convolution layer C3 and the pooling layer P4. KLP employed only three partitions in the last (FC) layers. From the FC5 layer on, there are only fully connected layers and KLP tends to use the least number of partitions possible in layers of this type and to repeat these partitions in subsequent FC layers to reduce communication. Thus, KLP assigned most of the vertices belonging to the layers FC5 and FC6 to only two partitions while it assigned the vertices of the last layer totally to partition two. This experiment indicates that the best partitioning approach strongly depends on the layer type and its connections.

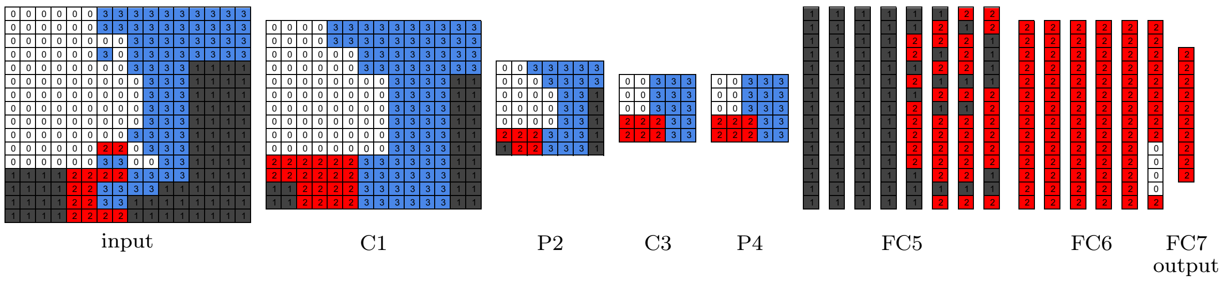


Figure 4.6: Best partitioning achieved by KLP using four partitions without considering IoT device memory restrictions. © 2018 IEEE.

More interesting results arise when we consider the memory limits of the devices. Figure 4.7 shows the best partitioning achieved by KLP for four devices considering memory restrictions and free input layer. When KLP considers memory restrictions, the input, convolution, and pooling layers present 2D niches, but they are less homogeneously than when KLP did not consider memory restrictions. This happens because the algorithm tends to fill all the available memory of the devices due to the possibility of single-vertex moves. Here again, from convolution to pooling layers, there is no communication cost among the partitions. In the first fully connected layer, KLP used only two partitions again, mostly partition two (red), while, in this experiment, the last two layers repeated partition two and only used it. This result confirms that the best partitioning approach strongly depends on the layer type and its connections. Additionally, it is worth noting that KLP could have assigned the vertices in partition three (blue) to partition zero (white) or partition one (dark grey), which would not change the communication cost. KLP does not move these vertices because its objective function is to reduce communication and nothing is constrained about the number of partitions used. However, and as KLP shows the amount of memory used by each partition at the end of the algorithm, the user can easily verify this possibility and apply it to the final partitioning.

For the six-partition, free-input-layer experiment, KLP virtually only employed two

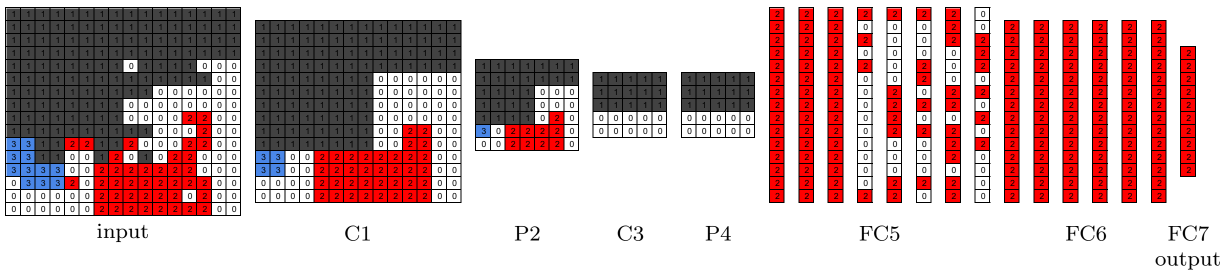


Figure 4.7: Best partitioning achieved by KLP with STM32F469xx memory restrictions when four devices were available and the input layer was free. © 2018 IEEE.

partitions in the first layers, as shown in Figure 4.8. The algorithm could discard one device and employed five devices altogether. KLP assigned the first fully connected layer to partition four (orange) only and the last two layers to only partition three (blue). In this experiment, KLP could have assigned the vertices in partition two (red) to partition one (dark grey) or partition three (blue), discarding one device and using four devices altogether without any changes in the communication cost. Comparing to the four-partition experiment, we can see that now the user would have to move only two vertices to discard one device. In other words, KLP itself could almost discard this device, missing only two vertices. This partitioning that almost used four devices is again due to the larger search space for KLP in the six-partition experiment: the algorithm could consider more operations to attempt communication reduction at each epoch. Another consequence of the larger search space for KLP in the six-partition experiment is the fact that KLP assigned each fully connected layer to one device only and the vertices in the convolution and pooling layers to fewer partitions than in the four-partition experiment. The larger search space allowed KLP to organize the partitioning better.

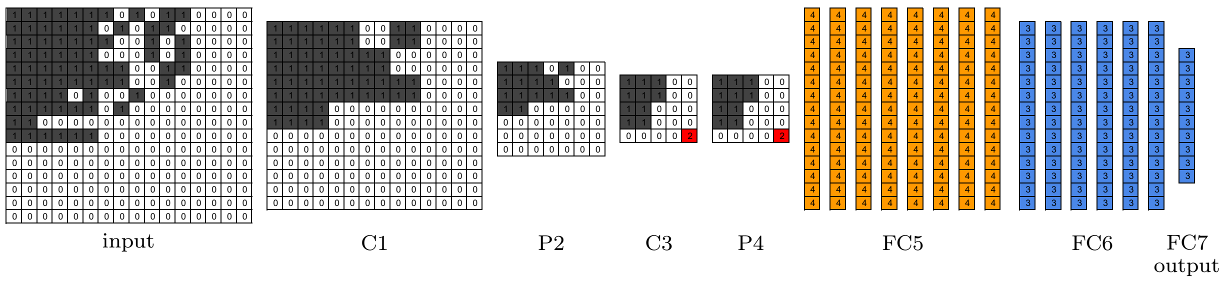


Figure 4.8: Best partitioning achieved by KLP with STM32F469xx memory restrictions when six devices were available and the input layer was free. © 2018 IEEE.

Figure 4.9 shows the best partitioning found by KLP when four devices were available and the input layer was locked to stay in the same partition during the whole partitioning algorithm. With a result slightly worse than that executed with the free input layer, as shown in Subsection 4.3.2, only two partitions were used in the convolution and pooling layers, being one of them the same as the input layer partition so that communication was reduced. In the fully connected layers, two partitions were also used, the same as the input layer partition and partition two (red), meaning that the algorithm could discard one device and used only three devices altogether. Although the communication cost was larger for this locked-input-layer experiment, KLP used fewer devices than the respective free-input-layer experiment.

Finally, Figure 4.10 shows the best partitioning when six devices were available and

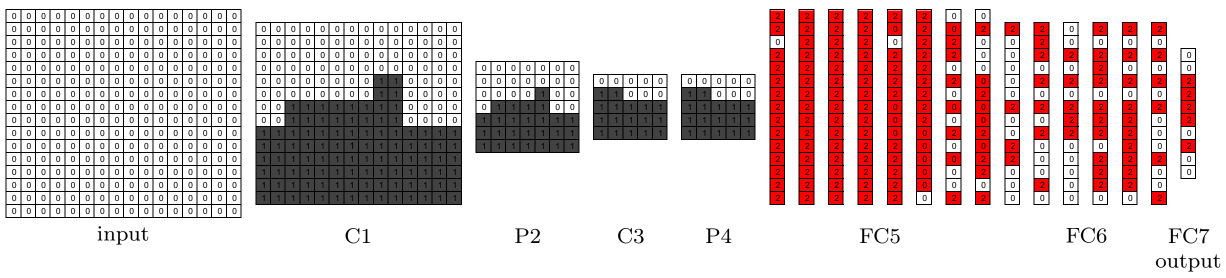


Figure 4.9: Best partitioning achieved by KLP with STM32F469xx memory restrictions when four devices were available and the input layer was locked to stay in the same partition. © 2018 IEEE.

the input layer was locked to stay in the same partition during the whole partitioning algorithm. KLP ended the partitioning algorithm using only four of the six devices available, keeping the first convolution and pooling layers virtually in the same partition as the input layer. KLP also assigned only two devices to the first layers and three devices to the fully connected layers: layer FC5 was completely assigned to partition five (green) while the last two layers were assigned to partitions four (orange, ∼60%) and zero (white, ∼40%). In this case, the user can also easily discard partition four because it uses an amount of memory that is available in partition one (dark grey). This would maintain the communication cost while reducing the total number of devices that KLP employed to three.

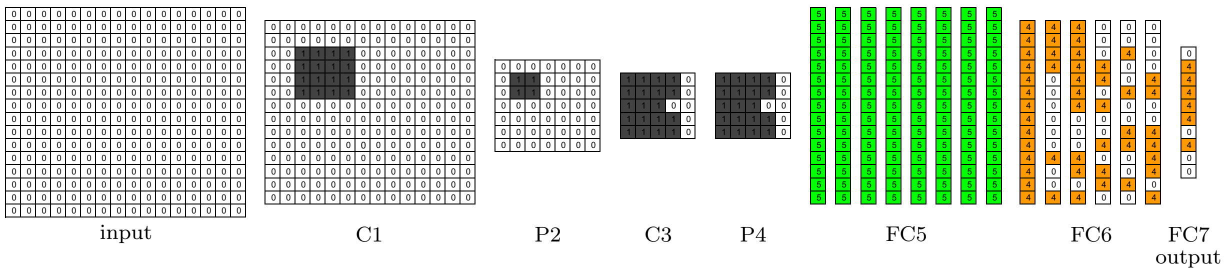


Figure 4.10: Best partitioning achieved by KLP with STM32F469xx memory restrictions when six devices were available and the input layer was locked to stay in the same partition. © 2018 IEEE.

The visual analysis of the partitionings generated by KLP shows that different layer types induce different partitioning patterns. For instance, input, convolution, and pooling layers induce the partitioning into 2D niches while fully connected layers induce the use of the smallest number of partitions to them. Additionally, the user can easily move the vertices of entire partitions to other partitions and discard the first partitions, in the case that the other partitions can provide the amount of memory required by the first ones. As KLP shows the amount of memory required by each partition at the end of the algorithm, the user does not need to perform the visual analysis to check if she/he can discard some partition. With the visual analysis of the partitionings generated by KLP, we could see that, by providing a larger number of partitions to KLP, the algorithm could achieve more organized partitionings with a smaller amount of communication and a smaller number of used devices than when we provided fewer devices to KLP. Finally, the visual analysis showed that experiments with the locked input layer could employ fewer devices.

4.3.4 Automatic Partitionings with Heterogeneous Setups

For the heterogeneous experiments, we performed four experiments using KLP with hypothetical heterogeneous devices inspired by the STM32F469xx device configuration. Two experiments employed three devices, being one a device with the same amount of memory of the STM32F469xx, another a device that presents half of the STM32F469xx memory, and the other provides 1.5 times the STM32F469xx memory. The other two experiments employed four devices: two devices with the same amount of memory of the STM32F469xx and two devices that provide half of the STM32F469xx memory. In each set of devices, the first experiment allowed all the vertices to be moved or swapped in KLP (free input layer) while the second experiment locked the input layer vertices, forcing them to stay in the same partition during the entire partitioning algorithm.

We compared each set of experiments to a different manual partitioning in which we attempted to assign each entire layer to the same partition. However, when it was not possible, we assigned the maximum possible number of vertices of one layer to the same partition and the remaining vertices to the next partition. Since this case happened to all the manual partitionings in this subsection, we show that approaches that limit the partitioning into layers such as the ones adopted by DIANNE, DeepX, and TensorFlow may not work for this set of constrained devices.

Figure 4.11 shows the amount of communication required for the manual partitionings and the minimum, median, and maximum communication costs that KLP achieved in each experiment. In the three-device experiments, KLP achieved a better result than the manual partitioning with 1.80 times less communication in the free-input-layer experiment and 1.65 times less communication in the locked-input-layer experiment. In these experiments, both the minimum and median amounts of communication achieved by KLP were smaller than the manual partitioning communication cost.

In the four-device, free-input-layer experiment, KLP found a partitioning with a communication cost of 1.21 times less than the manual partitioning communication cost. In the four-device, locked-input-layer experiment, KLP found a partitioning with a communication cost of 1.04 times less than the manual partitioning communication cost. Compared to the three-device experiments, KLP produced a smaller improvement over the manual partitioning, with only the minimum amount of communication of each experiment being smaller than the manual partitioning communication cost. Despite the same total amount of memory available in both three- and four-device experiments, the lower improvements in the four-device experiments are due to its lower flexibility: two devices that provide a lower amount of memory while, in the three-device experiments, one device makes up with its larger amount of memory for the fact that there are fewer devices.

The results in this subsection show that KLP can also provide superior partitionings for DNNs and heterogeneous constrained IoT devices. We also showed in this subsection that machine learning frameworks that employ a per-layer approach such as TensorFlow, DIANNE, and DeepX may not execute DNNs for constrained IoT devices.

4.4 Discussion

The KLP algorithm presents a computational complexity of $O(\mathrm{V}^4\mathrm{E})$, in which V is the number of vertices and E is the number of edges of the dataflow graph. If $\mathrm{E} \sim \mathrm{V}$, then the algorithm computational complexity is $O(\mathrm{V}^5)$. Since KLP presents the same computational complexity as our second algorithm, DN^{2}PCIoT, we discuss its implications

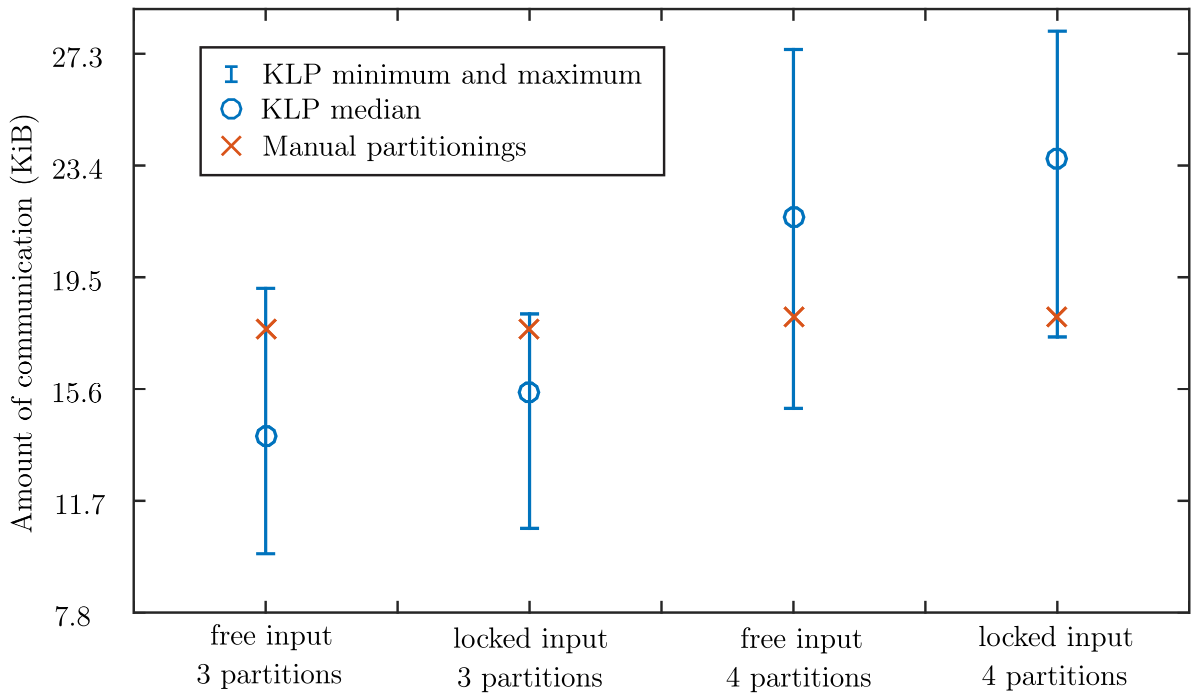


Figure 4.11: Amount of communication for the heterogeneous partitionings using KLP compared to a manual partitioning [28] (modified).

in the next chapter.

Summing up the results in this section, KLP achieved better results than partitionings that did not use any tool in all the nine experiments. To have the possibility of moving only one vertex per step also resulted in better partitionings because the final partitions were not limited to have the same number of vertices of the initial partitioning. We also achieved refined results when we provided more devices for KLP and the input layer was free: when we provided six devices, the result was 2.8 times smaller than the respective best manual partitioning and the partitioning used the same number of partitions of the four-device free-input-layer experiment since the algorithm discarded two devices. When the input was locked to stay in the same partition during the whole algorithm, KLP achieved worse results in comparison to when the input layer was free but used fewer devices. The partitioning patterns also changed, causing KLP to try to use fewer partitions in the first layers.

With the visualization of the partitionings provided by KLP and the analysis of partitionings that did not use any tool, we could infer that each layer type induces different partitioning patterns. For instance, for input, convolution, and pooling layers, the algorithm tends to group vertices into 2D niches within a layer and use the same niches assigned to the same devices along the neural network layers, which may explain the worse results when the input was locked to stay in the same partition during all the algorithm. For pooling layers, KLP can eliminate the communication cost. When there is a fully connected layer, KLP tends to use only one partition in this layer. Even if there are other adjacent fully connected layers, KLP tries to assign the same partition to all these layers but, if this is not possible, KLP will use the least number of partitions possible.

We performed four experiments with three and four heterogeneous devices and compared them to a manual partitioning that attempted to assign entire layers to partitions. This attempt was not successful and the manual partitionings had to use more than one partition per layer in some layers. This situation unveils that approaches adopted by common machine learning frameworks, such as DIANNE, DeepX, and TensorFlow, may not work with resource-constrained IoT devices. Moreover, KLP achieved a partitioning with a communication cost up to 1.80 times better than the manual partitioning, indicating the algorithm suitability for heterogeneous devices as well. Finally, all the experiments showed that it is essential to use an automatic partitioning tool to achieve an efficient DNN inference execution on constrained IoT devices and that KLP can perform this role well.

4.5 Final Remarks

In this chapter, we investigated how Convolutional Neural Networks can be partitioned for distributed inference on constrained IoT devices and proposed KLP, a Kernighan-and-Lin-based partitioning algorithm that partitions DNN models for efficient distributed execution on multiple IoT devices.

We first investigated how existing machine learning frameworks such as DIANNE, DeepX, and TensorFlow enable the users to partition neural networks for distributed execution and show that they usually require the user to manually partition the neural network in a per-layer fashion. We then partitioned the CNN called LeNet into four and six balanced partitions using manual approaches, including the per-layer approach of the existing frameworks, and showed that the KLP algorithm can find partitionings that are up to 1.47 times better than the manual approaches when memory restrictions are not considered. We also investigated how KLP compare to the manual partitioning approaches when partitioning for a set of homogeneous and heterogeneous memory-limited IoT devices. For the homogeneous setups, the KLP tool provides partitionings that are up to 2.8 times better than the ones found by the manual approaches. For the heterogeneous setups, we verified that some of the manual approaches, e.g., the per-layer approach, and the existing machine learning frameworks may not produce partitionings that are valid on memory-constrained devices. Moreover, we also showed that the KLP algorithm finds partitionings that are up to 1.8 times better than the manual ones. Finally, we run KLP maintaining the input layer within the same partition. For the homogeneous setups, KLP provides partitionings that are up to 4.5 times better than the per-layer approach while, for the heterogeneous setups, the partitionings found by KLP are up to 1.65 times better than the ones found by the manual approaches.

Although communication reduction is important because it reduces the amount of power consumed on radio operations and the amount of interference on the wireless medium, partitionings that aim at communication reduction may not lead to a better performance in the execution time or inference rate. Thus, in the next chapter, we propose another algorithm, which can use communication minimization or inference rate maximization as objective functions. Additionally, we proposed an adequate count of the memory required by the shared parameters and biases of CNNs so that we can employ the characteristics of devices that are more constrained in memory. We also compared the results of our algorithm to a state-of-the-art algorithm and other approaches that we proposed.

Chapter 5

Partitioning the LeNet Convolutional Neural Network for Inference Rate Maximization

In the previous chapter, we proposed Kernighan-and-Lin-based Partitioning (KLP) [28], an algorithm to automatically partition DNNs into constrained IoT devices, which aimed to reduce the number of communications among partitions. Communication reduction is important so that the network does not become overloaded, a situation that can be aggravated in a wireless connection shared with several devices. Even though reducing communication may help any system, in several contexts, one of the main objectives is to increase the inference rate, especially in applications that need to process a data stream [24, 70, 86, 130].

In this chapter, we extend our proposed KLP algorithm to create the Deep Neural Networks Partitioning for Constrained IoT Devices (DN^{2}PCIoT), an algorithm to automatically partition DNNs among constrained IoT devices that maximizes the inference rate or minimizes communication [76]. Additionally, for both of these objective functions, this new algorithm accounts more precisely for the amount of memory required by the shared parameters and biases of CNNs in each partition. This feature allows us to employ more constrained setups in the applications.

We use the inference rate maximization objective function to partition the LeNet CNN model using several approaches such as the per-layer partitioning provided by popular machine learning frameworks, partitionings provided by the general-purpose partitioning framework METIS, and by our algorithm DN^{2}PCIoT. We show that DN^{2}PCIoT starting from random-generated partitionings or DN^{2}PCIoT starting from partitionings generated by the other approaches results in partitionings that achieve up to 38% more inferences per second than METIS. Additionally, we also show that DN^{2}PCIoT can produce valid partitionings even when the other approaches cannot. We summarize the main contributions of the work described in this chapter as follows:

- the DN^{2}PCIoT algorithm that optimizes partitionings aiming for inference rate maximization or communication reduction while properly accounting for the memory required by the CNNs' shared parameters and biases;
- the DN^{2}PCIoT feature of starting from a partitioning obtained by another approach and trying to improve the solution based on this initial partitioning;

- a case study whose results show that the DN^{2}PCIoT algorithm can produce partitionings that achieve higher inference rates and that can produce valid partitionings for very constrained IoT setups;
- a case study that shows that popular machine learning frameworks such as TensorFlow, DIANNE, and DeepX may not be able to execute DNN models on very constrained devices due to their per-layer partitioning approach;
- a study of the METIS framework, which indicates that it is not an appropriate framework to partition DNNs for constrained IoT setups because it may not provide valid partitionings under these conditions;
- an analysis of the DNN model granularity results to show that our DNN with a larger grouping minimally affects the partitioning result;
- an analysis of how profitable it is to distribute the inference rate execution among multiple constrained devices; and
- a greedy algorithm to reduce the number of communications based on the available amount of memory of the devices.

5.1 Proposed Deep Neural Networks Partitioning for Constrained IoT Devices (DN^{2}PCIoT)

As we based the DN^{2}PCIoT on KLP, we present only the new features included in DN^{2}PCIoT in this section. For instance, the dataflow graph that represents the DNN to be partitioned now requires the number of FLoating-point OPerations (FLOP) that each vertex executes. Additionally, DN^{2}PCIoT can generate a random partitioning for the initial partitioning as in KLP or the user can define an initial partitioning in an input file. This file contains, for each vertex, the identifier of the partition assigned to it. It is worth noting that the defined initial partitioning can be the result of another partitioning framework, for instance. Neither METIS nor SCOTCH can start from a defined partitioning obtained by another algorithm.

We designed DN^{2}PCIoT to produce partitionings that maximize the neural network inference rate or minimize the amount of data transferred per inference. DN^{2}PCIoT can easily employ other objective functions due to its design. For this purpose, it is only necessary to change the *computeCost* function so that it calculates the cost of a partitioning according to the new objective function. Different from METIS, which reduces the number of communications while attempting to balance the computational load and memory requirements in the hope of achieving good computational performance, DN^{2}PCIoT directly optimizes the partitioning for inference rate maximization, using the equations explained in Section 2.6 and Section 2.8. In the inference rate maximization, the device or connection between devices that most limit the result is the maximum inference rate that some partitioning can provide.

Another feature that we included in DN^{2}PCIoT is the proper count for the amount of memory required by the shared parameters and biases of CNNs. DN^{2}PCIoT does that by counting only one set of shared parameters and biases of each layer per partition when there is one vertex of the corresponding layer assigned to the partition. This feature produces partitionings with a realistic amount of memory per partition, which is smaller than if the algorithm did not use this feature. Therefore, with this feature, DN^{2}PCIoT can produce valid partitionings for more constrained setups.

5.2 Methods and Materials

In this section, we show the LeNet models and the device characteristics that we used in the experiments as well as the experiment details and approaches.

5.2.1 LeNet Neural Network Model

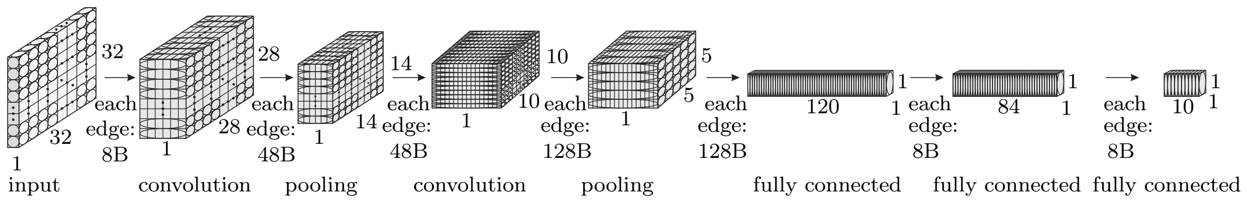
In this chapter, we also grouped the LeNet neurons to form the vertices. We again grouped the neurons in the same position of width and height but different positions in the depth dimension of the LeNet input, convolution, and pooling layers into one vertex because two neurons in these layers in the same position of width and height but different positions in depth present the same communication pattern. Thus, a partitioning algorithm would tend to assign these vertices to the same partition. For the inference rate, this modeling only affects the number of operations that a vertex will need to calculate. In the fully connected layers, as the width and height have size one, the depth was not modeled as having size one because this would limit too much the partitioning and the constrained devices able to execute this partitioning. For instance, a partitioning with this grouping would fit into only one setup of our experiments, which is the least memory-constrained setup that we used in the experiments described in this chapter.

We modeled two versions of LeNet:

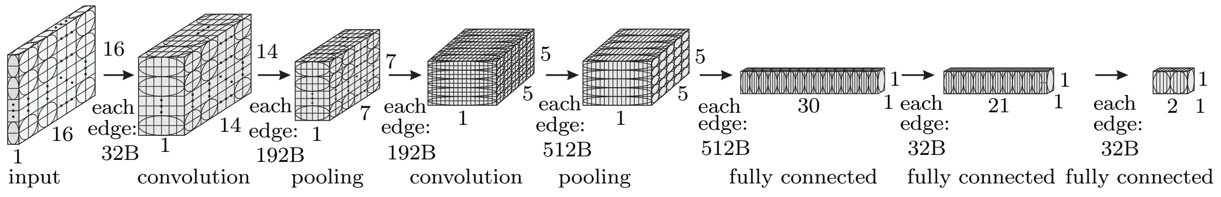
- **LeNet 1:1**: the original LeNet with 2343 vertices (except for the depth explained above); and
- **LeNet 2:1**: LeNet with 604 vertices, in which, for the input, convolution, and pooling layers, we grouped every four neurons (width and height of size two) to form one vertex in the source graph, except for the last pooling layer, and we grouped every four neurons in the depth of the fully connected layers to form one vertex in the source graph.

It is worth noting that the two LeNet versions that we used in this chapter are different from the LeNet version of the previous chapter. While the convolution and pooling layers of both LeNet 2:1 and the LeNet version of the previous chapter are equal, the fully connected layers of the LeNet version of the previous chapter are equal to the fully connected layers of LeNet 1:1, causing the LeNet version of the previous chapter to be an intermediate version between LeNet 2:1 and LeNet 1:1. Figure 5.1 shows the dataflow graph of each LeNet version with the following per-layer data: the number of vertices in height, width, and depth, the layer type, and the amount of transferred data in bytes required by each edge in each layer. In Figure 5.1, the cubes represent the original LeNet neurons and the circles and ellipses represent the dataflow graph vertices.

As we saw in Chapter 4, the grouping of the LeNet neurons reduces the dataflow graph size as we can see by the difference in the number of vertices for each graph. This reduction decreases the partitioning execution time so that we can perform more experiments in a shorter time frame. LeNet 1:1 is a finer-grained model, thus, it may achieve better results than a less fine-grained model such as LeNet 2:1. It is important to notice that this approach constrains the partitioning algorithm because it cannot assign



(a)



(b)

Figure 5.1: LeNet architecture and vertex granularity used in our experiments. Each cube stands for a CNN neuron while each circle or ellipse is a vertex in the source dataflow graph. Edges represent data transfers and are labeled with the number of bytes per inference that each edge must transfer. (a) **LeNet 1:1**: the original LeNet with 2343 vertices with the neurons in depth grouped to form only one vertex. (b) **LeNet 2:1**: LeNet with 604 vertices, in which the width and height of each convolution and pooling layer were divided by two, except for the last pooling layer, and the depth of the fully connected layers was divided by four [76].

neurons that were grouped into the same vertex to different partitions since they are now grouped. However, in this chapter, we also want to show that a coarser-grained model such as LeNet 2:1 can achieve comparable results to a finer-grained model such as LeNet 1:1 and, thus, can be employed to reduce the size of the dataflow graph and the time required to partition it. It is also important to highlight that our approach for grouping the vertices is different from the METIS multilevel approach [49], producing better results than METIS.

Table 5.1 shows the amount of memory and computation (the number of FLOP per inference) required by each vertex in each layer per LeNet model. The other characteristics that are inherent to the original LeNet-5 DNN architecture [63] that we used in this thesis can be seen in Table 4.1.

Table 5.1: Characteristics of each LeNet model used in this chapter [76].

Characteristic	**model**	**input**	**C1**	**P2**	**C3**	**P4**	**FC5**	**FC6**	**FC7**
Memory per	LeNet 1:1	8	48	48	128	128	3216	976	688
vertex (B)	LeNet 2:1	32	192	192	512	128	12864	3904	3440
Computation per	LeNet 1:1	0	306	36	765	96	51	240	168
vertex (FLOP)	LeNet 2:1	0	1224	144	3060	96	204	960	840

5.2.2 Setups and Experiments

Four different devices inspired the setups that we used in the experiments. These devices are progressively constrained in memory and computational and communication performance. These values are shown in Table 5.2. The devices belong to class D1, D2, and *others* of constrained devices that we showed in Chapter 2. The first column shows the maximum number of devices allowed to be used in each experiment. The second column shows the name of the device model that inspired each experiment. The third column shows the amount of RAM that each device provides, which is available in the respective device datasheet [8, 101–103]. The amount of RAM that each device provides varies from 16 KiB to 388 KiB. The fourth column represents the estimated computational performance of each device, which varies from 1.6 MFLOP/s to 180 MFLOP/s. Finally, communication is performed through a wireless medium. As this medium is shared with all the devices, we estimated that the communication performance decreased with the number of communication links between each device, which, in turn, depends on the number of devices allowed to be used in the experiments. For instance, for two devices, we have only one communication link, however, for four devices, we have six communication links. Therefore, with connections able to transfer up to 300 Mbits/s, we consider the communication performance for each device to vary from 9.4 KiB/s to 6103.5 KiB/s. It is worth noting that we assume a homogeneous communication performance between all the devices and a constant communication performance during the whole algorithm.

Table 5.2: Device data and the maximum number of devices allowed to be used in the experiments [76].

Number of devices allowed to be used in the experiments	**Device model**	**Device amount of RAM (KiB)**	**Device estimated computational power (FLOP/s)**	**Communication performance between each device (KiB/s)**
2	STM32F469xx [102]	388	180×10^6	6103.5
4	Atmel SAM G55G [8]	176	120×10^6	3051.8
11	STM32L433 [103]	64	80×10^6	332.9
56	STM32L151VB [101]	16	1.6×10^6	11.9
63	STM32L151VB [101]	16	1.6×10^6	9.4

The reasoning for the maximum number of devices allowed to participate in the partitioning is the following. As the amount of memory provided by each device decreases, we need to employ more devices to enable a valid partitioning. Furthermore, the memory of shared parameters and biases should be taken into account when choosing the number of devices in an experiment because of the experiments that start with random-generated partitionings. To produce valid random-generated partitionings, each device should be able to contain at least one vertex of each neural network layer and its respective shared parameters and bias. This condition, in some cases, may increase the number of needed devices. For instance, the memory needed for LeNet (to store intermediate results, parameters, and biases) is 546.625 KiB if each layer is entirely assigned to one device. If the devices provide up to 64 KiB, it is possible to achieve valid partitionings using nine devices to fit the LeNet model. However, to start with random-generated partitionings and, thus, requiring that each device can contain at least one vertex of each layer and its respective shared parameters and bias, the number of required devices increases to 11 to produce valid random-generated partitionings.

For each experiment, the communication links between each device present the same performance, which is constant during the whole partitioning algorithm. The difference in the communication performance for the most constrained setups (with 56 and 63 devices) is due to our estimation based on the number of devices sharing the same wireless connection. Thus, for the experiment in which the system can employ up to 63 devices for the partitioning, the communication links perform a little worse than when the system can employ up to 56 devices, although the same device models with the same available memory and computational performance are used.

5.2.3 Types of Input Layers in the Experiments

For each setup in Table 5.2, we performed two types of experiments with the input layers:

- the **free-input-layer** experiment, in which all the LeNet model vertices were free to be swapped or moved; and
- the **locked-input-layer** experiment, in which the LeNet input layer vertices were initially assigned to the same device and, then, they were locked, i.e., the input layer vertices could not be swapped or moved during the whole algorithm.

These two types of experiments are the same as the types of experiments in the previous chapter. The free-input-layer experiments allow all the vertices to freely move from one partition to the others, including the input layer vertices. These experiments represent situations in which the device that produces the input data cannot process any part of the neural network and, thus, must send its data to nearby devices. In this case, we would have to add more communication to send the input data (the LeNet input layer) from the device that contains these data to the devices chosen by the approaches in this work. However, as the increased amount of transferred data involved in sending the input data to nearby devices is fixed, it does not need to be shown here. On the other hand, the locked-input-layer experiments represent situations in which the device that produces the input data can also perform some processing, therefore, no additional cost is involved in this case.

We employed nine partitioning approaches for each experiment listed in this section (for each setup and free and locked inputs). We explain these approaches in the next subsections and show the corresponding visual partitionings for the approaches that cause it to be necessary. It is worth noting that we do not consider these visual partitionings as the results of this chapter and we show them here for clarification of the approaches.

5.2.4 Per Layers: User-Made Per-Layer Partitioning (Equivalent to Popular Machine Learning Frameworks)

The first approach to performing the experiments is the per-layer partitioning performed by the user. In this approach, we perform the partitioning per layers, i.e., we assign a whole layer to a device. Popular machine learning frameworks such as TensorFlow, DIANNE, and DeepX offer this type of partitioning. TensorFlow allows a fine-grained partitioning, but only if the user does not use its implemented functions for each neural network layer type.

Considering the LeNet model [63] used in our experiments, it is possible to calculate the layer that requires the largest amount of memory. This layer is the second fully connected layer (the last but one LeNet layer), which requires 376.875 KiB for the parameters, the biases, and to store the layer final result. Thus, when considering the constrained devices chosen for our experiments (Table 5.2), it is possible to see that there is only one setup that can provide the necessary amount of memory that a LeNet per-layer partitioning requires. This setup is the least constrained in our experiments and allows a maximum of two devices in the partitioning.

In the per-layer partitioning approach, the user performs the partitioning, so we partitioned LeNet for the first setup and show the resultant partitioning in Figure 5.2. In this figure, we show only the partitioning for LeNet 2:1 because the partitioning for LeNet 1:1 is equivalent. It is worth noting that each color in this figure and in all the figures that represent visual partitionings corresponds to a different partition.

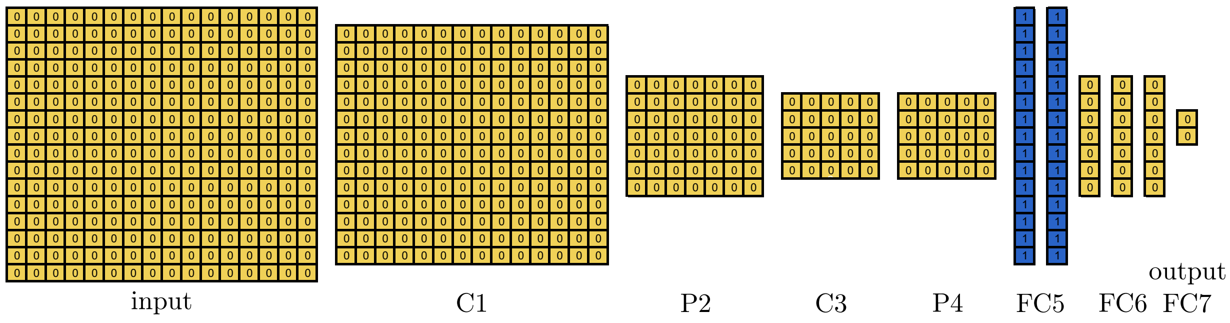


Figure 5.2: LeNet 2:1 user-made per-layer partitioning [76].

5.2.5 Greedy: A Greedy Algorithm for Communication Reduction

The second approach is a simple algorithm that aims to reduce communication. In this algorithm, whose pseudocode is listed in Algorithm 3, we assign the layers to the same device in order until it has no memory to fit some layer. Next, if there is any space left in the device and the layer type is convolution, pooling, or input, then a 2D matrix of vertices (width and height) that fits into the rest of the memory of this device is assigned to it or, if the layer is fully connected, then the algorithm assigns to the device a number of vertices that fit into the rest of the memory of this device. After that or if there is any space left in the device, we assign the next layer or the rest of the current layer to the next device and the process goes on until the algorithm assigns all the vertices to a device. This greedy algorithm assumes that there is a sufficient amount of memory in the setups for the neural network model. Furthermore, the algorithm can partition graphs using fewer devices than the total number of devices provided. This algorithm contains two loops that depend on the number of layers (L) and the number of devices (or partitions, P) of the setup, which renders the algorithm complexity equal to $O(L+P)$. However, it is worth noticing that both L and P are usually much smaller than the number of neurons in the neural network.

Figure 5.3 shows the visual partitioning using the greedy algorithm for each experiment. This algorithm works both for the free-input-layer and the locked-input-layer experiments because the input layer could be entirely assigned to the same device for all setups. Furthermore, as the partitioning scheme is similar for LeNet 2:1 and LeNet 1:1

in the experiments with 2, 4, and 11 devices, we show only the partitionings for LeNet 2:1 in Figure 5.3a, 5.3b, and 5.3c for the sake of simplicity. For the experiments with 56 and 63 devices, the greedy algorithm results in the same partitioning because these setups employ the same device model. However, as LeNet 2:1 uses 44 devices and LeNet 1:1 employs 38 devices, we show both results in Figure 5.3d and 5.3e, respectively.

Algorithm 3 Greedy algorithm for communication reduction [76].

```
1: function GREEDYALGORITHM($lenet$, $setup$)
2:     for $l \leftarrow 1$ to $lenet.numberOfLayers$ do
3:         for $device \leftarrow setup.first$ to $setup.last$ do
4:             if $lenet[l].remainingMem \neq 0$ then
5:                 if $lenet[l].remainingMem \leq device.availableMemory$ then
6:                     assign $l$ to $device$;
7:                 else if $lenet[l].type = conv$ or $pooling$ or $input$ then
8:                     assign a 2D matrix of vertices that fit into $device$;
9:                 else if $lenet[l].type = fully\ connected$ then
10:                    assign the number of vertices that fits into $device$;
11:                end if
12:                $device.availableMemory \leftarrow device.availableMemory - assigned$;
13:                $lenet[l].remainingMem \leftarrow lenet[l].remainingMem - assigned$;
14:            else
15:                break;
16:            end if
17:        end for
18:    end for
19: end function
```

5.2.6 iRgreedy: User-Made Partitioning Aiming for Inference Rate Maximization

The third approach is a partitioning performed by the user that aims for the inference rate maximization. The rationale behind this greedy approach is to equally distribute the vertices of each layer to each device since all the experiments present a homogenous setup. Thus, this approach employs all the devices provided for the partitioning. Besides that, again, we divide the layers into 2D matrices for the input, convolution, and pooling layers.

Figure 5.4a shows the visual partitioning for the 11-device free-input LeNet 2:1 experiment. For the two- and four-device free-input experiments, the partitioning follows the same pattern. For the 2-, 4-, and 11-device locked-input experiments, we changed only the input layer partitioning and assigned it to only one device. Thus, we do not show these partitionings here, but we provide all the partitionings produced by the greedy algorithm, Inference-rate greedy approach (iRgreedy) approach, and METIS together with

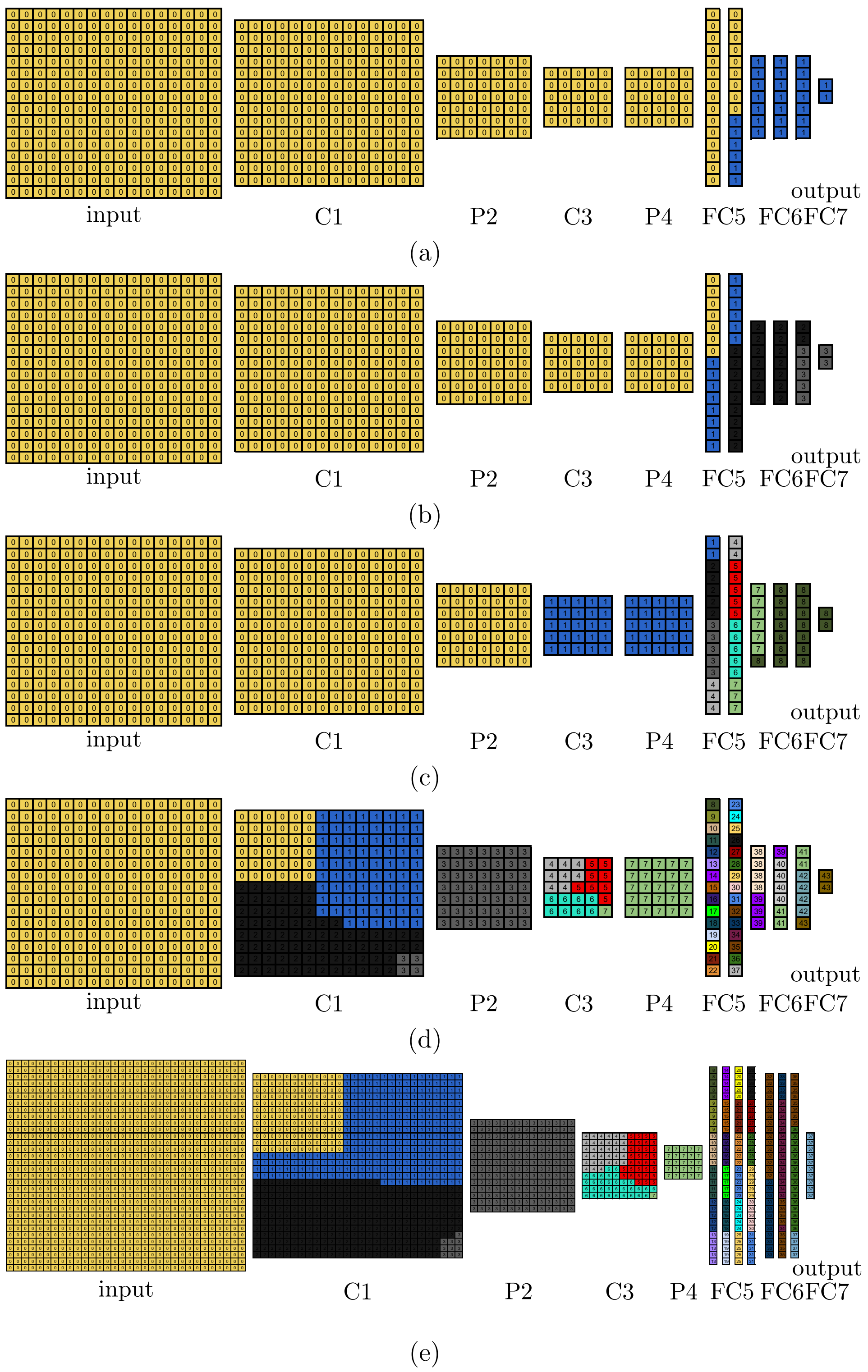


Figure 5.3: Partitionings using the greedy algorithm: (a) LeNet 2:1 for the two-device experiments; (b) LeNet 2:1 for the four-device experiments; (c) LeNet 2:1 for the 11-device experiments (used nine devices); (d) LeNet 2:1 for the 56- and 63-device experiments (used 44 devices); and (e) **LeNet 1:1** for the 56- and 63-device experiments (used 38 devices) [76].

the source code of all the algorithm implementations [74]. We used these partitionings as initial partitionings for *DN²PCIoT after approaches*, as explained in Subsection 5.2.9.

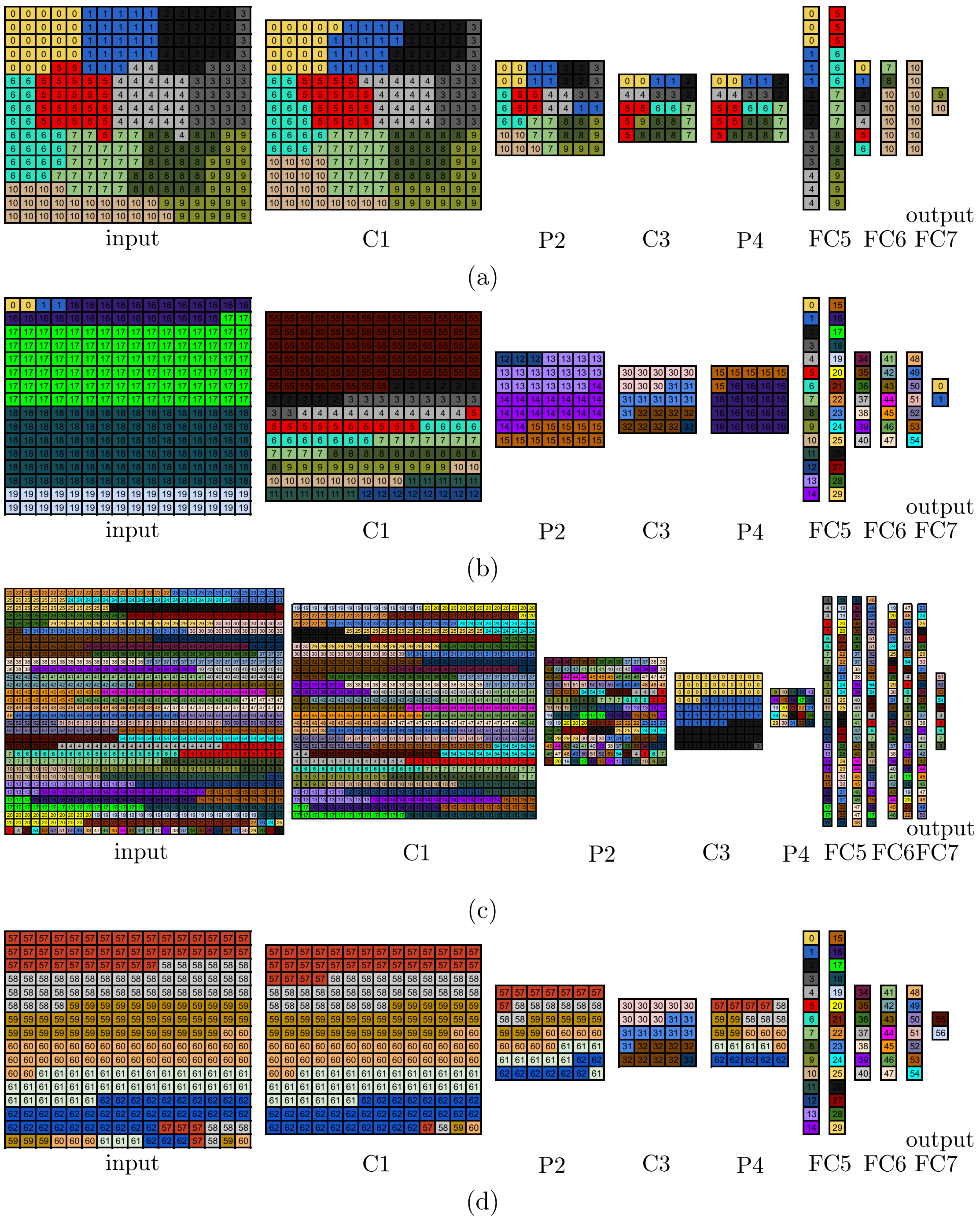


Figure 5.4: Partitionings using the inference rate greedy approach: (a) LeNet 2:1 for the 11-device experiments; (b) LeNet 2:1 for the 56-device experiments; (c) **LeNet 1:1** for the 56-device experiments; and (d) LeNet 2:1 for the 63-device experiments [76].

For the 56- and 63-device experiments, it was not possible to employ the same rationale due to memory issues. Thus, for these experiments, the rationale was to start by the layers that require most memory and assign to the same device the largest number of vertices possible of that layer. Furthermore, in these experiments, we assigned the vertices in a per-line way because the layers were not equally distributed to all the available devices.

This approach reduces the number of copies of the shared parameters and biases and, thus, allows for a valid partitioning. For the locked-input experiments, besides changing the input layer by assigning it entirely to only one device, we had to perform some adjustments to produce valid partitionings. Figure 5.4b and 5.4c show the visual partitionings for the 56-device free-input LeNet 2:1 and LeNet 1:1 partitioning, respectively. Additionally, in the 63-device experiments with LeNet 2:1, we assigned the vertices in the same positions of the first layers to the same devices to reduce communication. We show the visual partitioning for this case in Figure 5.4d. As the approach for 63 devices in LeNet 1:1 was the same for the 56 devices, we do not show the visual partitioning for 63 devices in LeNet 1:1 here either. The two approaches detailed in this subsection are greedy and, therefore, we call them Inference-rate greedy approach (iRgreedy) in the rest of this chapter.

5.2.7 METIS

In this approach, we used the program *gpmetis* from METIS to automatically partition LeNet and compare the results with our approaches. The reason to choose METIS is that it is considered a widely known state-of-the-art framework used to automatically partition graphs for general purpose.

METIS offers several parameters that the user can modify like the number of different partitionings to compute, the number of iterations for the refinement algorithms at each stage of the uncoarsening process, the maximum allowed load imbalance among the partitions, and the algorithm's objective function. The number of partitions corresponds to the maximum number of devices in each setup described in Section 5.2.2. Thus, as METIS attempts to balance all the constraints, it always employs the maximum number of devices in each experiment. We varied all the other parameters that we listed here in our tests and, for our inference rate maximization objective function, we replaced the maximum allowed load imbalance among the partitions parameter for the maximum allowed load imbalance among partitions per constraint, which allows using different values for memory and computational load. It is important to note that METIS considers the computational load as a constraint and, thus, attempts to balance it. For the objective function parameter, we used both functions: *edgecut* minimization and total communication volume minimization. We can use these two functions because our source graphs represent the actual underlying communications, as METIS requires. These parameters are detailed in the METIS manual [48]. We report the METIS parameters that led to the METIS results in this chapter in Appendix A.

For the locked-input experiments, we removed the LeNet graph vertices from the input layer to run METIS with a small difference between the constraints proportion (target weights in METIS) related to the amount of memory and computational load that the input layer requires. After METIS performs the partitioning, we plugged the input layer back into the LeNet graph and calculated the cost (inference rate or amount of transferred data) and if this partitioning is valid.

5.2.8 DN^{2}PCIoT 30R

The fifth approach that we used for the experiments was the application of DN^{2}PCIoT starting from random-generated partitionings. This approach executed DN^{2}PCIoT 30 times starting from different random-generated partitionings and we report the best value achieved in these 30 executions. We executed this approach only for the LeNet 2:1 model due to the more costly execution of the LeNet 1:1 model starting from a random partitioning. This was the only approach that did not employ LeNet 1:1. Furthermore, DN^{2}PCIoT can discard some devices when they are not necessary, i.e., if DN^{2}PCIoT finds a better partitioning (larger inference rate or smaller amount of communication, depending on the objective function) with fewer devices.

5.2.9 DN^{2}PCIoT after Approaches

The last approach corresponds to the execution of the proposed DN^{2}PCIoT starting from partitionings obtained by the other approaches that we considered in this chapter. Thus, we performed four experiments with this approach: DN^{2}PCIoT after per layers, DN^{2}PCIoT after greedy, DN^{2}PCIoT after *iRgreedy*, and DN^{2}PCIoT after METIS. This approach also allows the partitionings to employ fewer devices than the maximum number of devices allowed in each experiment. It is worth noting that no other approach in this thesis can start from a partitioning obtained by another approach and try to improve the solution based on this initial partitioning.

5.3 Experimental Results

In this section, we show the results for all the experiments performed with the setups and the approaches discussed in Section 5.2 (varying the number of devices, free and locked input layer, and all the approaches) for the inference rate maximization objective function. After that, we show the pipeline parallelism factor for each setup to compare the performance of a single device to the distribution performance. Finally, we plot the results of the inference rate maximization along with the results for communication minimization to see how optimizing for one objective function affects the other. We compared our approaches (greedy algorithm, iRgreedy approach, DN^{2}PCIoT 30R, and DN^{2}PCIoT after all the other approaches) to two literature approaches: the per-layer approach (equivalent to popular machine learning frameworks such as TensorFlow, DIANNE, and DeepX) and METIS. We implemented DN^{2}PCIoT using C++ and executed the experiments on Linux-based operating systems.

5.3.1 Inference Rate Maximization

Table 5.3 summarizes the results for the inference rate maximization objective function for the approaches detailed in Section 5.2. In this subsection, we compare these approaches to DN^{2}PCIoT. Table 5.4 shows the results for the inference rate maximization objective function for DN^{2}PCIoT 30R and DN^{2}PCIoT after all the approaches in Table 5.3. It is worth noting that both Table 5.3 and Table 5.4 present normalized results, i.e., we normalized these results by the maximum inference rate achieved in each experiment. For instance, in the free-input two-device experiments, considering both Table 5.3 and Table 5.4, DN^{2}PCIoT after METIS with LeNet 2:1 achieved the maximum inference rate. We take this value and divide it by each result of the free-input two-device experiments. Thus, we have a value of 1.0 for the maximum inference rate in the experiment with DN^{2}PCIoT after METIS and LeNet 2:1 and the values of the other approaches reflect how many times the inference rate was worse than the maximum inference rate. We colored the table so that the red color represents the worst results and the green color represents the best results, i.e., results that are close to 1.0. The yellow color represents intermediate results.

In the first column of both tables, the number indicates the maximum number of devices allowed in each experiment, “free” refers to the free-input-layer experiments, and “locked” refers to the locked-input-layer experiments. As discussed in Section 5.2, some approaches could not produce valid partitionings and this is represented by an “x”. For LeNet 1:1, as it is a large graph with 2343 vertices, we had to interrupt some executions and we report the best value found until the interruption. We marked these executions with an asterisk (“*”).

As general results, it is possible to see in Table 5.3 and Table 5.4 that DN^{2}PCIoT 30R and DN^{2}PCIoT after approaches led to the best values for all the experiments. DN^{2}PCIoT 30R produced results that range from intermediate to the best results, with only 20% of the experiments yielding intermediate results. The DN^{2}PCIoT 30R results show the robustness of DN^{2}PCIoT, which can achieve reasonable results even when starting from random partitionings.

There are some important conclusions that we can draw from Table 5.3. The per-layer partitioning is the most limiting approach when considering constrained devices because it could only partition the model for the least constrained device setup, which used two devices. It is worth noting that this approach is the one offered by popular machine learning frameworks such as TensorFlow, DIANNE, and DeepX. Thus, we show that these frameworks cannot execute DNNs in very constrained setups. Moreover, the per-layer partitioning produced suboptimal results for the only setup that it could produce valid partitionings. The quality of these results was due to the heavy unbalanced partitioning in the per-layer approach, which overloaded one device while assigned a low load to the other device, as the least constrained setup offered two devices.

The state-of-the-art framework METIS also led to suboptimal results because it attempts to balance all the constraints, which are memory and computational load. Additionally, several partitionings provided by METIS were invalid because METIS does not consider a limit for the amount of memory in each partition. METIS could not produce any valid partitionings at all for the 56- and 63-device experiments because METIS is not capable of properly accounting for the memory required by the shared parameters and biases of CNNs. One way to solve this issue would be to add the memory required by the shared parameters and biases to every vertex that needs them, even if METIS assigned the vertices to the same partition. However, this solution would require much more memory and no partitioning using this solution would be valid for the setups used in this chapter. Thereby, we gave METIS the LeNet model without the memory information required by the shared parameters and biases hoping that METIS would produce valid partitionings. METIS had the conditions to produce valid partitionings in this situation since our setups provided a sufficient amount of memory for LeNet and one full set of shared parameters and biases for each device. Unfortunately, METIS did not produce any valid partitionings in any of the 56- and 63-device constrained setups.

Finally, the greedy algorithm and the *iRgreedy* approach are simple approaches. Although they produced poor results, they could produce valid partitionings for all the proposed setups. Thus, considering the ability to produce valid partitionings, these approaches demonstrated to be better than METIS and the per-layer partitioning offered by popular machine learning frameworks in the proposed setups.

In Table 5.4, we can see that DN^{2}PCIoT starting from the partitionings produced by the other approaches achieved results that range from intermediate to the best results,

Table 5.3: Normalized results for the naive approaches. The minimum and maximum consider Tables 5.3 and Table 5.4 [76].

Setup	**Median of 30 Random**	**Per layers 2:1**	**Per layers 1:1**	**Greedy 2:1**	**Greedy 1:1**	**iRgreedy 2:1**	**iRgreedy 1:1**	**METIS 2:1**	**METIS 1:1**
2 free	6.35	1.67	1.67	1.59	1.59	1.61	1.36	1.13	1.23
4 free	4.09	x	x	2.06	2.06	1.43	1.21	1.09	1.14
11 free	2.32	x	x	4.49	4.49	1.56	1.67	1.40	1.38
56 free	2.12	x	x	29.25	29.53	24.00	1.45	x	x
63 free	1.92	x	x	27.53	27.80	6.59	1.32	x	x
2 locked	5.25	1.37	1.37	1.31	1.31	1.73	1.52	1.11	1.12
4 locked	4.83	x	x	2.03	2.03	1.90	1.68	1.27	1.33
11 locked	3.25	x	x	4.08	4.08	3.33	2.83	1.29	1.34
56 locked	2.74	x	x	17.50	17.66	11.25	1.34	x	x
63 locked	2.15	x	x	14.74	14.88	3.53	1.28	x	x

Table 5.4: Normalized results for DN^{2}PCIoT 30R and DN^{2}PCIoT after approaches [76].

		DN^{2}PCIoT after							
Setup	**30R 2:1**	**per layers 2:1**	**per layers 1:1**	**greedy 2:1**	**greedy 1:1**	**iRgreedy 2:1**	**iRgreedy 1:1**	**METIS 2:1**	**METIS 1:1**
2 free	1.13	1.20	1.38	1.06	1.37	1.02	1.18	1.00	1.01
4 free	1.38	x	x	1.16	1.25	1.16	1.14	1.00	1.01
11 free	1.19	x	x	1.34	1.42*	1.18	1.09	1.04	1.00
56 free	1.12	x	x	2.62	5.14*	2.12	1.00*	x	x
63 free	1.00	x	x	2.59	5.84*	2.71	1.21*	x	x
2 locked	1.00	1.09	1.08	1.01	1.12	1.12	1.11	1.02	1.02
4 locked	1.27	x	x	1.29	1.25	1.00	1.45	1.25	1.23
11 locked	1.29	x	x	1.26	1.18	1.00	1.01	1.10	1.22
56 locked	1.46	x	x	2.50	4.07*	1.91	1.00*	x	x
63 locked	1.17	x	x	2.17	3.41*	2.14	1.00*	x	x

like the DN^{2}PCIoT 30R approach. When comparing to the state-of-the-art framework METIS, DN^{2}PCIoT after METIS could improve the METIS result by up to 38%. Additionally, DN^{2}PCIoT after approaches is a better approach when compared to DN^{2}PCIoT 30R because DN^{2}PCIoT after approaches do not need to run 30 times to attempt to find a better partitioning as in the DN^{2}PCIoT 30R approach. Furthermore, the single execution required by DN^{2}PCIoT after each approach may run faster than DN^{2}PCIoT 30R because it starts from the intermediate result achieved by the other approaches instead of a random partitioning that usually requires several epochs to converge.

The DN^{2}PCIoT after the greedy approach result also shows the DN^{2}PCIoT robustness because the greedy algorithm produced the worst results mostly. Nevertheless, DN^{2}PCIoT after the greedy algorithm could improve the poor results of the greedy algorithm up to 11.1 times, yielding at least intermediate results in comparison to the other approaches.

The LeNet 1:1 model runs in a considerably larger time than the LeNet 2:1 model due to the difference in the number of vertices and edges between the graphs. When comparing the two LeNet models used in the experiments, it is possible to see that DN^{2}PCIoT for LeNet 2:1 led to the best result or a result close to the best result in 80% of the experiments. The LeNet 2:1 model became more restrictive only in the scenarios with the locked input layer and the most constrained devices, i.e., with 56 and 63 devices. Thus, the results for the proposed setups suggest that it is possible to employ LeNet 2:1 for faster partitionings with a limited impact on the results.

To conclude, our results show that DN^{2}PCIoT starting from 30 random-generated partitionings and DN^{2}PCIoT after the other approaches achieved the best results for the inference rate maximization in all the proposed experiments and should be employed when partitioning CNNs for execution on multiple constrained IoT devices. On the other hand, the approach offered by popular machine learning frameworks such as TensorFlow and DIANNE may not be used for very constrained devices. Additionally, the state-of-the-art partitioning algorithm METIS cannot produce valid partitionings for very constrained devices either.

5.3.2 Pipeline Parallelism Factor

After showing the results for the inference rate maximization objective function, it is interesting to look at the pipeline parallelism factor to check if there is gain or loss when distributing the neural network execution. In the first column of Table 5.5, we have the device model and the maximum number of devices allowed in each setup. The second column shows the inference rate if the entire LeNet model fits into one device's memory, i.e., the inference rate based on the computational performance of each device. In this column, it is possible to see that the diminishing computational performance affects the inference rate performance, as expected. In the third column, there is the best inference rate achieved in the corresponding experiments of the previous subsection. Finally, the fourth column shows the pipeline parallelism factor, which is the best inference rate achieved in the experiments (third column) divided by the inference rate if the entire LeNet fit into one device's memory (second column). It is worth noting that the larger is the parallelism factor, the better, and the result that is smaller than one indicates that there is performance loss when executing the neural network inference in a distributed fashion.

Table 5.5: Pipeline parallelism factor for each setup [76] (modified).

Setups	**Single device inference rate* (inferences/s)**	**Best inference rate in the experiments (inferences/s)**	**Pipeline parallelism factor**
2x STM32F469xx	507.265	864.22	1.70
4x Atmel SAM G55G	338.177	757.03	2.24
11x STM32L433	225.451	162.65	0.72
56x STM32L151VB	4.509	21.14	4.69
63x STM32L151VB	4.509	17.65	3.91

* If the device fits into the memory required by the whole LeNet model.

The communication performance among the devices limited all the experiments in Table 5.5. It is possible to note that there is a gain in the inference rate performance in using 2, 4, 56, and 63 devices. For the 11-device experiment, the amount of communication among the devices surpasses the distribution computational gain and negatively affects performance. In this case, the distributed execution achieved only 72% of the performance offered by a single device. However, we have to remember that this device alone cannot execute this model due to its memory limit.

For the last device model, used in the 56- and 63-device experiments, we have different values for the best inference rate in the experiments due to the communication links among them, which are less powerful in the 63-device experiment and, consequently, its result is worse than for 56 devices. Furthermore, the computational performance of the most constrained devices used in these experiments (STM32L151VB) is so low that we have gains of 4.7 and 3.9 when distributing LeNet, even considering the communication overhead. These results show that, even if we could execute LeNet in a single device, it would be more profitable to distribute the execution to achieve a higher inference rate, except for the 11-device setup.

It is important to note that, with this distribution, we enable such a constrained system to execute a CNN like LeNet. This would not be possible if we employed only a single constrained device due to the lack of memory. However, in our most constrained setups, the inference rate may be low. This can be the case, for instance, in an anomaly detection application that classifies incoming images from a camera. As most surveillance cameras generate 6–25 frames per second [42], most of the setups presented in this chapter satisfy the inference rate requirement for this application. Nonetheless, the most constrained setups do not satisfy the ideal inference rate requirement of a surveillance application, thus, the system may lose some frames. In the worst case, we still have 71% of the maximum required inference rate (17.65/25), allowing the system to execute the application, even if the inference rate is not ideal.

5.3.3 Inference Rate versus Communication

Minimizing communication is important to reduce interference in the wireless medium and to reduce the power consumed by radio operations. Common real-time applications that need to process data streams in a small period such as anomaly detection from camera images, for instance, the detection of vehicle crashes and robberies, may require a minimum inference rate so that there is no frame loss. In this type of application, reducing communication or even energy consumption is desirable so that the network is not overloaded and device energy life is augmented. On the other hand, applications that may process data at a lower rate such as non-real-time image processing may require a small amount of communication so that device battery life is extended while desirable characteristics are the network non-overload and inference rate maximization.

In this subsection, we want to show how optimizing for one of the objective functions, for instance, inference rate maximization, affects the other, for instance, communication reduction. For this purpose, Figure 5.5 presents the results of Subsection 5.3.1 for the inference rate maximization along with their respective values for the amount of transferred data per inference for each partitioning. We also plotted in these graphs results for the communication reduction objective function, which allows for a fair comparison in the amount of transferred data. For instance, when the objective function is the inference rate, the amount of transferred data may be larger than when the objective function is communication reduction. The inverse may also occur for the inference rate. We obtained the results for the communication reduction objective function by executing all the approaches discussed in Section 5.2, including DN^{2}PCIoT 30R and DN^{2}PCIoT after the other approaches with the communication reduction objective function.

Each graph in Figure 5.5 corresponds to one setup. In Figure 5.5a, “comm” in the legend parentheses stands for when the approach used the communication reduction objective function, “inf” stands for when the approach used the inference rate maximization objective function, “free” stands for the free-input-layer experiment, and “locked” stands for the locked-input-layer experiment. It is worth noting that each approach in the legend corresponds to two points in the graphs of Figure 5.5, one for the execution of LeNet 2:1 and one for LeNet 1:1. DN^{2}PCIoT 30R is an exception because we executed it only for LeNet 2:1, thus, each approach with DN^{2}PCIoT 30R in the legend corresponds to only one point in the graphs. Another exception is the per-layer partitioning, which yielded the same result for both LeNet models and, thus, its results correspond to only one point in the graphs. In this subsection, we do not distinguish the two LeNet versions employed in this chapter because our focus is on the approaches and so that the charts do not get polluted.

As we want to maximize the inference rate and minimize the amount of transferred data, the best trade-offs are the ones on the right and bottom side of the graph, i.e., in the southeast position. We draw the Pareto curve [50] using the results for the inference rate maximization and communication reduction achieved by all the approaches listed in Section 5.2 to show the best trade-offs and we divided the graphs into four quadrants according to the minimum and maximum values for each objective function. These quadrants help the visualization and show within which improvement region each approach fell.

In Figure 5.5b, for the two-device experiments, the Pareto curve contains two points, which correspond to the result of the free-input DN^{2}PCIoT after METIS for the inference rate maximization and most of the results of the locked-input DN^{2}PCIoT after approaches for communication reduction. The only approach that fell within the southeast quadrant is the free-input DN^{2}PCIoT after METIS for the inference rate maximization, which is the best trade-off between the inference rate and the amount of transferred data for this setup. Although several points fell within the southeast quadrant, it is worth noting that the three points that are closest to this best trade-off all correspond to the results of the free-input DN^{2}PCIoT for the inference rate maximization, showing the robustness of DN^{2}PCIoT.

In Figure 5.5c, for the four-device experiments, the approach that fell both in the Pareto curve and in the southeast quadrant is the free-input DN^{2}PCIoT after *iRgreedy* when reducing communication. Therefore, this approach presents the best trade-off for the four-device setup. In this setup, all points that fell in the Pareto curve correspond to DN^{2}PCIoT again.

Six points compose the Pareto curve for the 11-device experiments in Figure 5.5d. Three of these points fell in the best trade-off quadrant and are the results of the free-input DN^{2}PCIoT after *iRgreedy* for communication reduction and free- and locked-input METIS for inference rate maximization. In this case, the final choice for the best trade-off depends on which condition is more important: if the application requires a larger inference rate, then METIS is the appropriate choice. On the other hand, if the application requires a smaller amount of communication, then DN^{2}PCIoT after *iRgreedy* for communication reduction is a better approach.

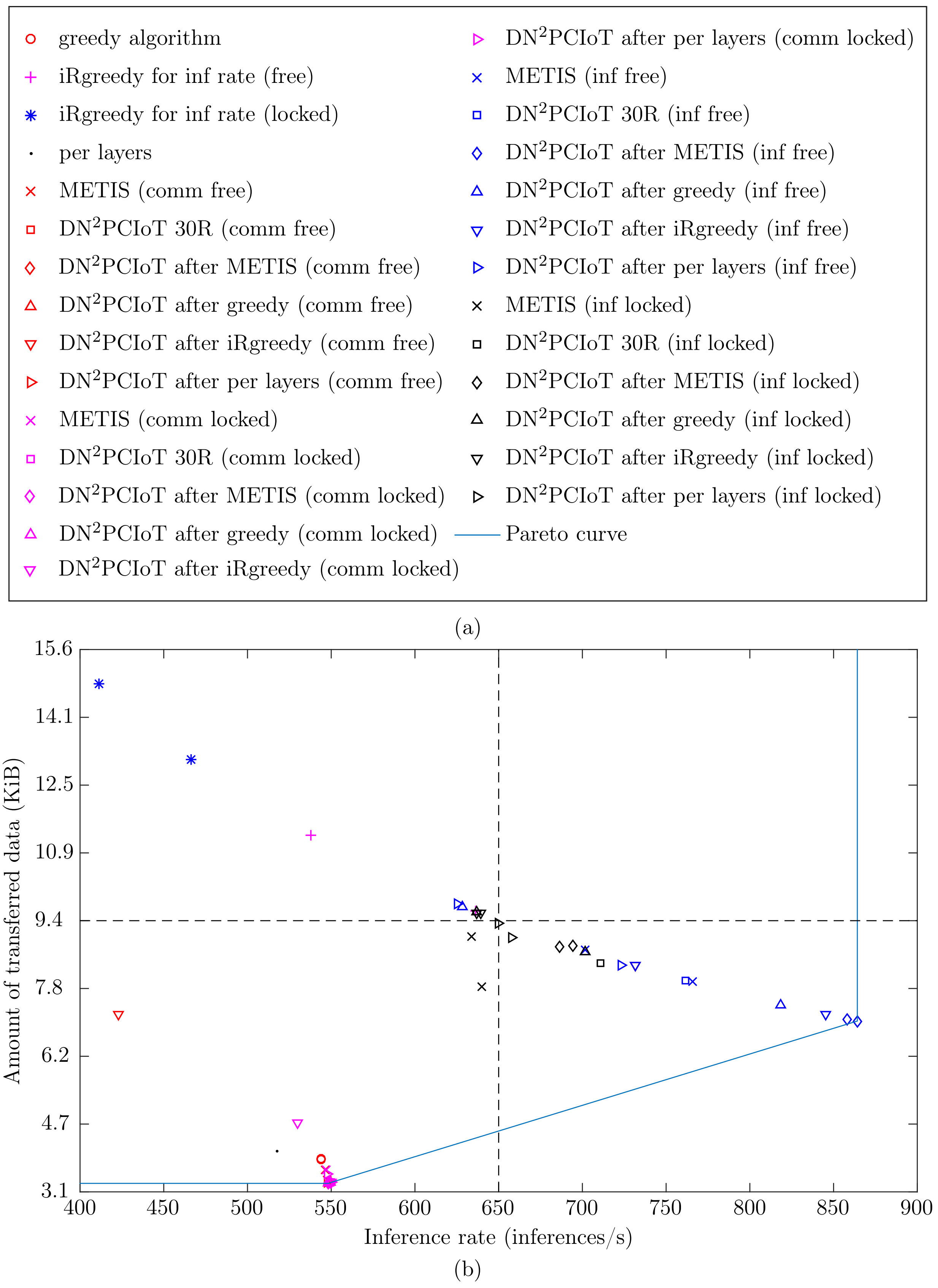
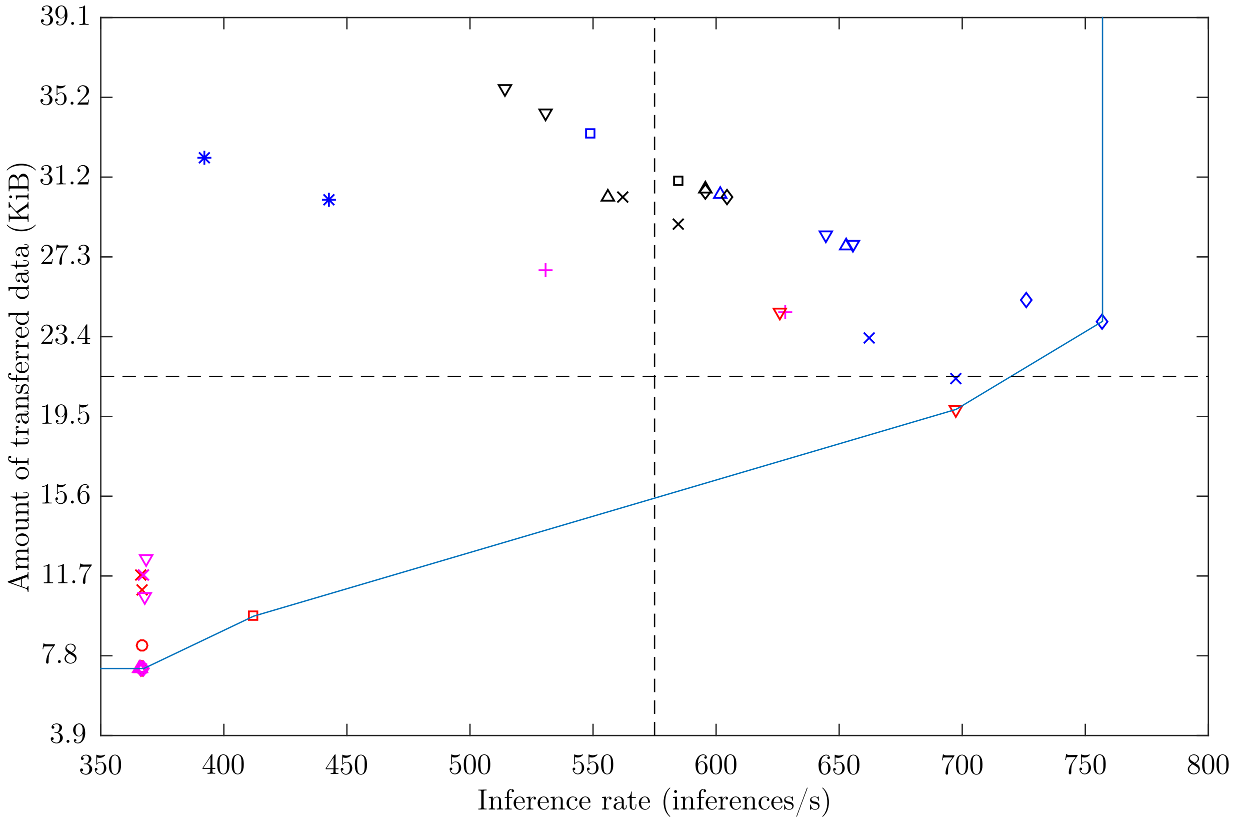
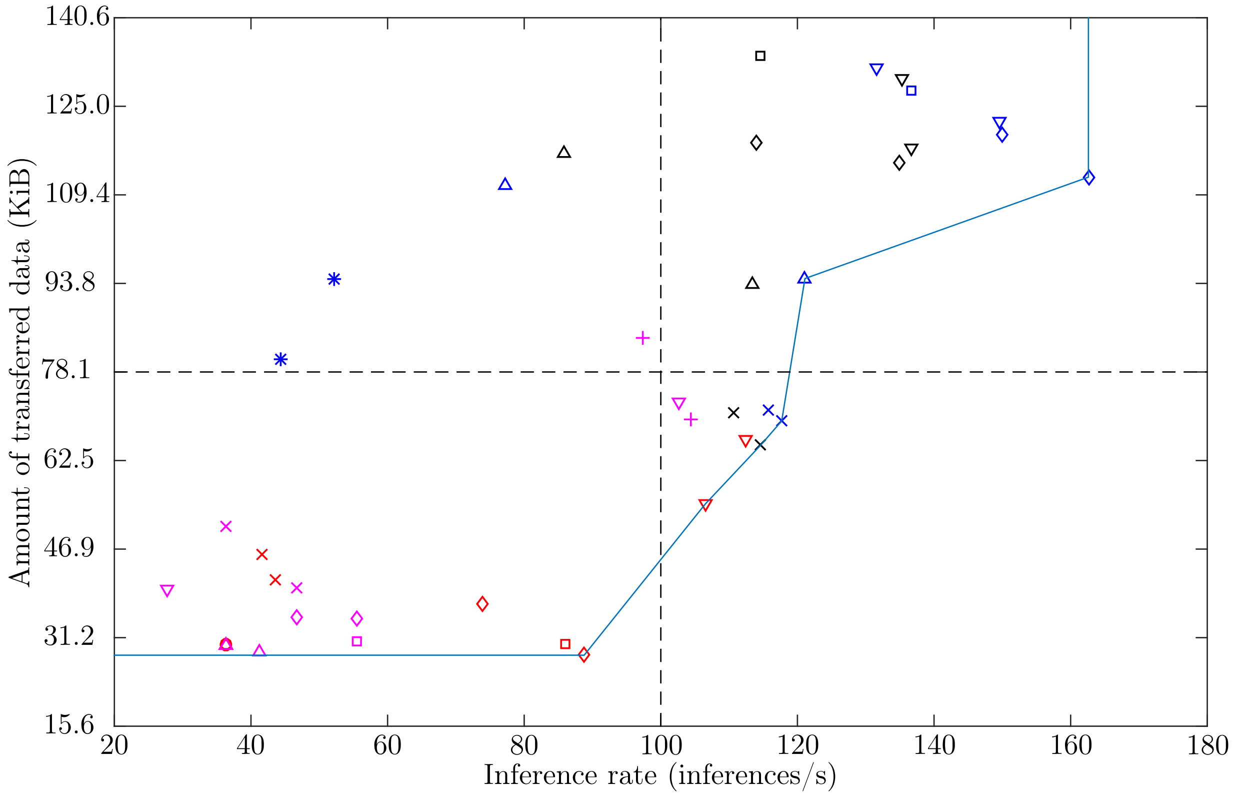


Figure 5.5: (a) Legend for all graphs and (b) inference rate and communication values for the two-device experiments [76] (modified).

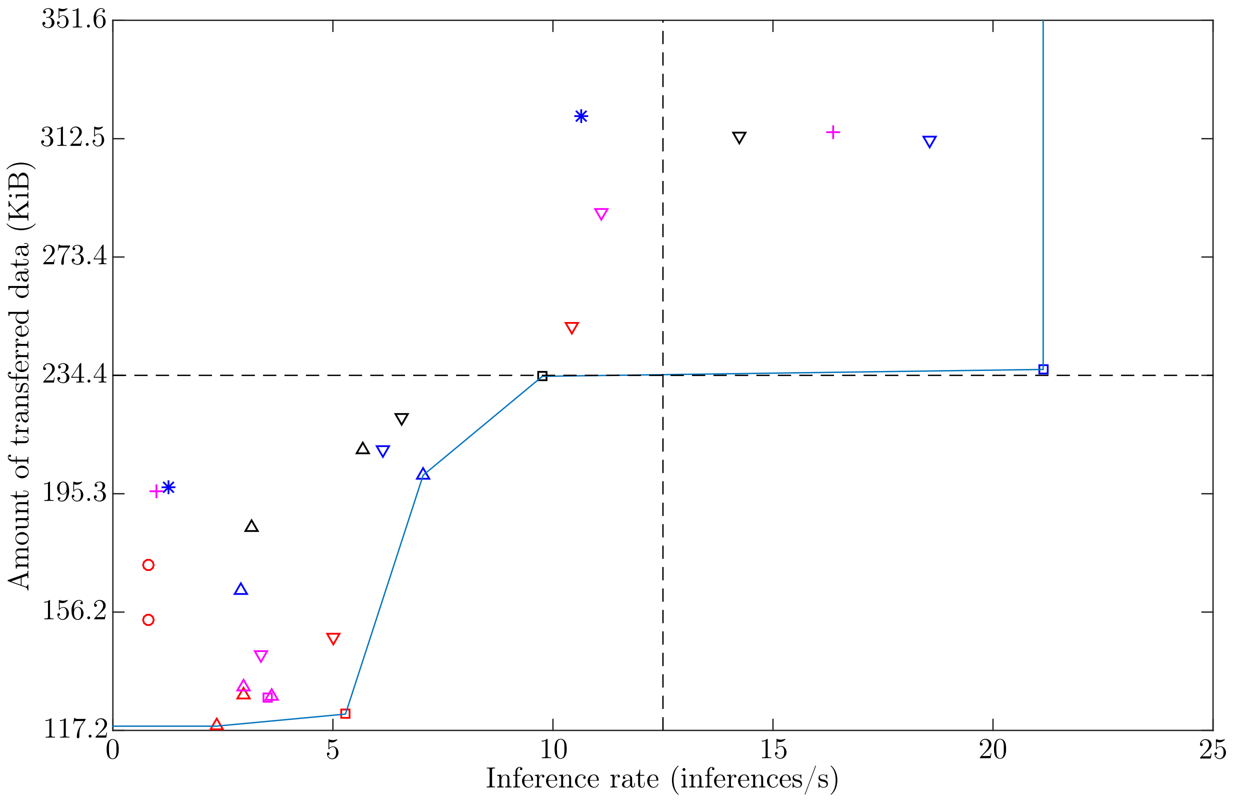


(c)

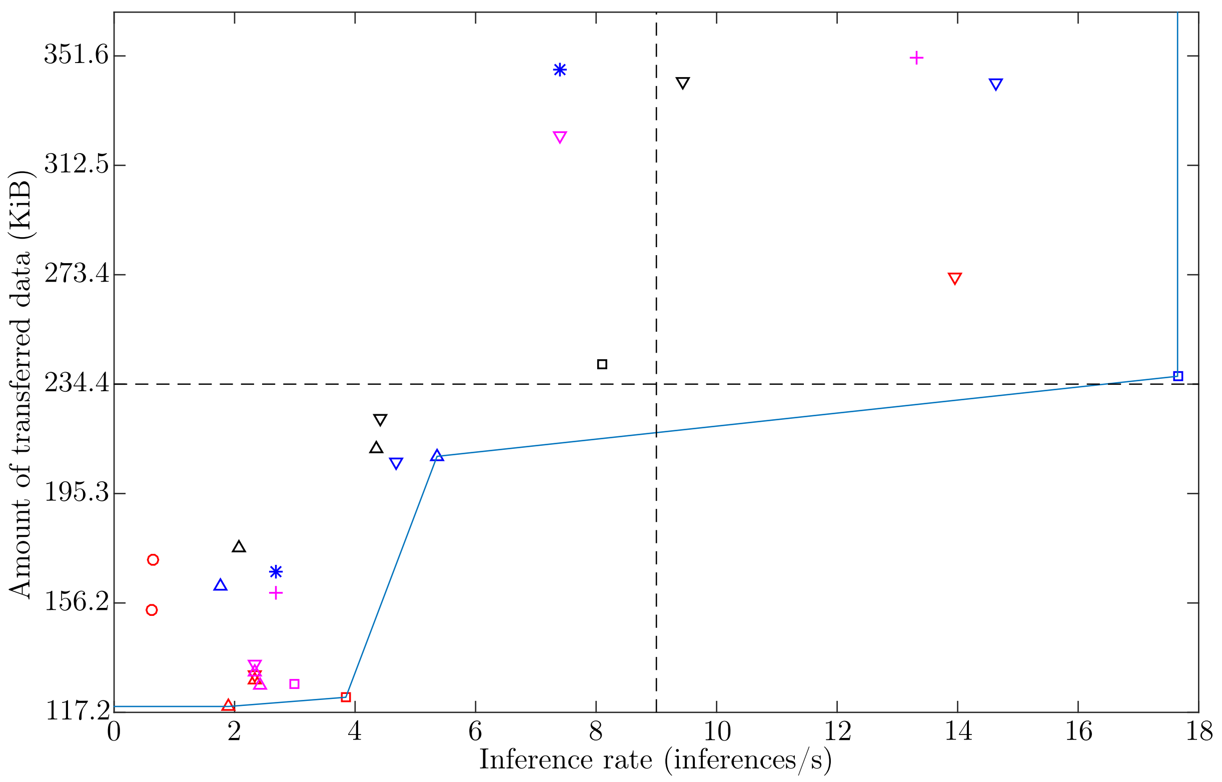


(d)

Figure 5.5: (Continued) Inference rate and communication values for the: (c) four-device experiments; and (d) 11-device experiments [76] (modified).



(e)



(f)

Figure 5.5: (Continued) Inference rate and communication values for the: (e) 56-device experiments; and (f) 63-device experiments [76] (modified).

Six points also compose the Pareto curve for the 56-device experiments in Figure 5.5e. In this graph, the approach that fell both in the Pareto curve and closest to the southeast quadrant is the free-input DN^{2}PCIoT 30R when maximizing the inference rate. Therefore, this approach presents the best trade-off for the 56-device setup. In this setup, all points that fell in the Pareto curve correspond to the DN^{2}PCIoT results.

Finally, in Figure 5.5f, for the 63-device experiments, the approach that fell both in the Pareto curve and closest to the southeast quadrant is the free-input DN^{2}PCIoT 30R when maximizing the inference rate. This approach presents the best trade-off for the 63-device setup. Again, in this setup, all points that fell in the Pareto curve correspond to the DN^{2}PCIoT results.

Back to the example of anomaly detection in Subsection 5.3.2, in which the application requirements involve an ideal inference rate of around 24 inferences per second while reducing communication is desirable, we can choose the best trade-offs for each setup analyzed in this subsection. In Figure 5.5b, Figure 5.5c, and Figure 5.5d, for the setups with 2, 4, and 11 devices, respectively, all the points in the Pareto curve satisfy the application requirement of the ideal inference rate. Thus, we can choose the points that provide the minimum amount of communication so that we do not overload the network. However, in Figure 5.5e and Figure 5.5f, for the setups with 56 and 63 devices, respectively, the points in the Pareto curve with the minimum amount of communication do not satisfy the ideal inference rate application requirement. Hence, we have to choose the points with the largest inference rate in the Pareto curve of each setup, which requires more communication. These results evidence the lower computational performance of the devices used in the 56- and 63-device setup.

Our results suggest that our algorithm also delivers the best trade-offs between the inference rate and communication, with DN^{2}PCIoT providing more than 90% of the results belonging to the Pareto curve. DN^{2}PCIoT after the approaches or DN^{2}PCIoT starting from 30 random partitionings achieved the best trade-offs for the proposed setups, even though these approaches aim at only one objective function. Thus, DN^{2}PCIoT 30R and DN^{2}PCIoT after approaches are adequate strategies when we need both communication reduction and inference rate maximization, although it is possible to improve DN^{2}PCIoT with a multi-objective function containing both objectives.

5.4 Discussion

In this section, we discuss the limitations of our approach. Our algorithm presents a computational complexity of $O(\mathrm{V}^4\mathrm{E})$, in which V is the number of vertices and E is the number of edges of the dataflow graph. If $\mathrm{E} \sim \mathrm{V}$, then the algorithm computational complexity is $O(\mathrm{V}^5)$. Thus, the grouping of the neural network neurons may be necessary so that the algorithm executes in a feasible time. As our results show, in most cases, the LeNet version that groups more neurons presents a limited impact on the results while the algorithms may execute faster, as the problem size is smaller. Other algorithms such as METIS perform an aggressive grouping and, thus, can also execute in a feasible time. However, it is worth noting that, with 30 executions, our DN^{2}PCIoT algorithm achieves results that are close to the best result that DN^{2}PCIoT achieved for an experiment, which includes DN^{2}PCIoT after the other approaches. On the other hand, we had to execute METIS with many different parameters to achieve valid partitionings and find the best result that METIS can get, adding up to more than 98,000 executions. Thus, METIS execution time is also not negligible.

More modern and larger CNNs such as VGGNet and ResNet would require more devices and/or devices with a larger amount of memory so that partitioning algorithms can produce valid partitionings. However, as these CNNs are also composed of convolution, pooling, and fully connected layers, the partitioning patterns [28] tend to be similar. Additionally, as current CNNs present more neurons, strategies that group more neurons similar to LeNet 2:1 or in multilevel partitioning algorithms such as METIS [49] may also be required so that the partitioning algorithm executes in a feasible time.

Other strategies that we can use to reduce the algorithm execution time are to start from partitionings obtained with other frameworks and to interrupt execution as soon as the partitioning achieves a target value or the improvements are smaller than a specified threshold. We can also combine our algorithm with other strategies such as the multilevel approach, which automatically groups graph vertices [49], but without the shortcomings of METIS such as invalid partitionings. In the next chapter, we apply the multilevel strategy to solve some of the limitations of our algorithm. It is important to notice that, even with these limitations, our results suggest that there is a large space for improvements when we consider constrained devices and compare the results to well-known approaches.

5.5 Final Remarks

In this chapter, we partitioned the LeNet Convolutional Neural Network for distributed inference into constrained Internet-of-Things devices using nine different approaches and we proposed Deep Neural Networks Partitioning for Constrained IoT Devices (DN^{2}PCIoT), an algorithm that partitions graphs representing Deep Neural Networks for distributed execution on multiple constrained IoT devices aiming for inference rate maximization or communication reduction. This algorithm adequately treats the memory required by the shared parameters and biases of CNNs so that DN^{2}PCIoT can produce valid partitionings for very constrained devices. Additionally, we can easily modify DN^{2}PCIoT to use other objective functions as well.

We partitioned two versions of the LeNet model with different levels of neuron grouping using five different setups aiming for inference rate maximization. We employed several approaches for the partitionings, including the per-layer approach, which is the approach offered by popular machine learning frameworks such as TensorFlow, DIANNE, and DeepX, and the widely used framework METIS. We compared the results produced by these approaches to the results produced by DN^{2}PCIoT and showed that either the approaches could not produce valid partitionings for more constrained setups or they yielded suboptimal results, with DN^{2}PCIoT achieving up to 38% more inferences per second than METIS. We also calculated the inference rate for a single device of each experiment assuming that the memory of this device was sufficient to execute the whole LeNet. We showed that, even if it was possible to execute the inference on a single device, there may be performance advantages of distributing its execution among multiple devices such as gains from 1.7 to 4.69 times in the inference rate provided by DN^{2}PCIoT. Finally, we plotted the results for the inference rate maximization objective function along with the respective amount of transferred data so that it was possible to see how optimizing for one objective function affects the other. Our results suggest that our algorithm can also deliver the best trade-offs between the inference rate and communication, with DN^{2}PCIoT providing more than 90% of the results belonging to the Pareto curve. The partitionings for both versions of LeNet achieved comparable results, with the less fine-grained LeNet model leading to the best results in 80% of the experiments. Thus, we showed that we can use a less fine-grained model following our grouping strategies in the partitionings with a limited impact on the results.

In the next chapter, to partition larger graphs in a feasible time, we propose the last algorithm in this thesis. This algorithm employs the multilevel approach, which gradually groups the graph vertices, executes a partitioning algorithm in the coarsest graph, and gradually returns to the original graph, refining the partitioning at each subgraph in this phase [49].

Chapter 6

Multilevel Deep Neural Networks Partitioning for Constrained IoT Devices

In this chapter, we propose another algorithm, the Multilevel Deep Neural Networks Partitioning for Constrained IoT Devices (MDN^{2}PCIoT), which explores techniques to enhance DN^{2}PCIoT and solve the limitations discussed in the previous chapter. Its main technique is the multilevel approach, in which we gradually reduce the graph size by grouping vertices, execute a partitioning algorithm in the smallest graph, and, then, gradually return to the original graph, refining the partitioning at each subgraph [49]. We used the DN^{2}PCIoT algorithm to partition the smallest graph and refine the partitioning at each subgraph. Furthermore, we applied some techniques to DN^{2}PCIoT so that it executes faster within the MDN^{2}PCIoT, thus, we call this new version of DN^{2}PCIoT *faster-DN^{2}PCIoT*. We designed and experimented with new approaches to perform the initial automatic vertex grouping, coarsest graph partitioning, and uncoarsening phase. In the experiments, we partition the LeNet and the AlexNet CNN models and compare our approach to METIS and to a simple approach based on the Best Fit approach [20, 92] applied to the original graph that represents the CNNs.

We summarize the main contributions of the work described in this chapter as follows:

- a new multilevel algorithm using a modified version of DN^{2}PCIoT as partitioning algorithm;
- a more flexible heavy-edge matching coarsening phase that, depending on the amount of memory of the devices in the partitioning, groups the vertices limiting to either a percentage of the amount of memory provided by the device that provides the smallest amount of memory or to the total amount of memory required by all the vertices, excluding the memory required by the shared parameters and biases;
- the DN^{2}PCIoT feature of factoring redundant edges out of the cost computation by modifying the edge treatment when grouping vertices, which also allows for a larger reduction in the number of edges in the subgraphs;
- an always-valid coarse partitioning for the smallest, coarsest graph, which satisfies memory constraints since the algorithm beginning;

- a more flexible uncoarsening phase that executes either one epoch of DN^{2}PCIoT or the whole DN^{2}PCIoT algorithm depending on the number of vertices of the subgraphs and the number of devices in the partitioning; and
- a study case to validate our algorithm and to experiment with a larger CNN for the inference rate maximization and communication reduction objective functions.

6.1 Proposed MDN^{2}PCIoT

In the previous chapters, we investigated manual groupings of the LeNet neurons of each layer. These groupings allowed us to reduce the dataflow graph size and, thus, to perform more experiments in a shorter time frame since the input was smaller. However, this process of manually grouping vertices is prone to errors and may consume too much time as it is not automatic. Thus, another solution is to automatically group the vertices and apply the multilevel approach. With this grouping, we can generate subgraphs with decreasing sizes.

The multilevel approach has three phases: the coarsening phase, the coarse partitioning phase, and the uncoarsening phase [49]. Initially, the algorithm automatically and gradually groups the vertices so that it reduces the graph size until it has a few hundred vertices. This first stage is the coarsening phase. Then, in the coarse partitioning phase, the algorithm partitions the coarsest graph produced in the previous step using, in our case, the Best Fit algorithm and the faster-DN^{2}PCIoT algorithm. After this partitioning, the algorithm gradually ungroups the vertices that it previously grouped, passing through each subgraph produced in the coarsening phase. During this process, for every subgraph, the algorithm refines the partitioning obtained in the previous subgraph to improve the partitioning. In our case, we also apply the faster-DN^{2}PCIoT algorithm to perform these refinements. This final phase is the uncoarsening phase. Figure 6.1 shows the multilevel approach steps.

We may explore new approaches to each step separately. In this section, we explain each algorithm phase detailing the approaches that we employed from METIS and our new proposals. We used some techniques from METIS, for instance, in the coarsening phase, we applied the heavy-edge matching technique. In this technique, for each vertex that is not grouped yet, we choose the edge with the largest weight and group the two vertices that this edge connects. Thus, we employed the heavy-edge matching because it tends to produce subgraphs with a reduced amount of communication. We visit the vertices by their degree order, like METIS, so that every vertex has the chance to be grouped. After the heavy-edge matching, if any vertices were not grouped, we perform a two-hop matching in the graph. In the two-hop matching, we can group two vertices if they were not grouped before in this subgraph and if they both have an edge that connects a vertex in common. Unlike METIS, we build deterministic subgraphs, which require only one execution.

When we group vertices, we have to limit the vertex size so that the grouped vertices do not affect the balance among the vertices in the subgraph and/or memory restrictions. METIS does not allow grouped vertices whose size is larger than a percentage of the sum of the size of all vertices. In the case that there is only one constraint, this percentage is 1.5% and, in the case that there is more than one constraint, 7.5%. In our problem, the sum of the size of all vertices is equivalent to the total amount of memory required by the vertices, excluding the memory required by the shared parameters and biases. We used

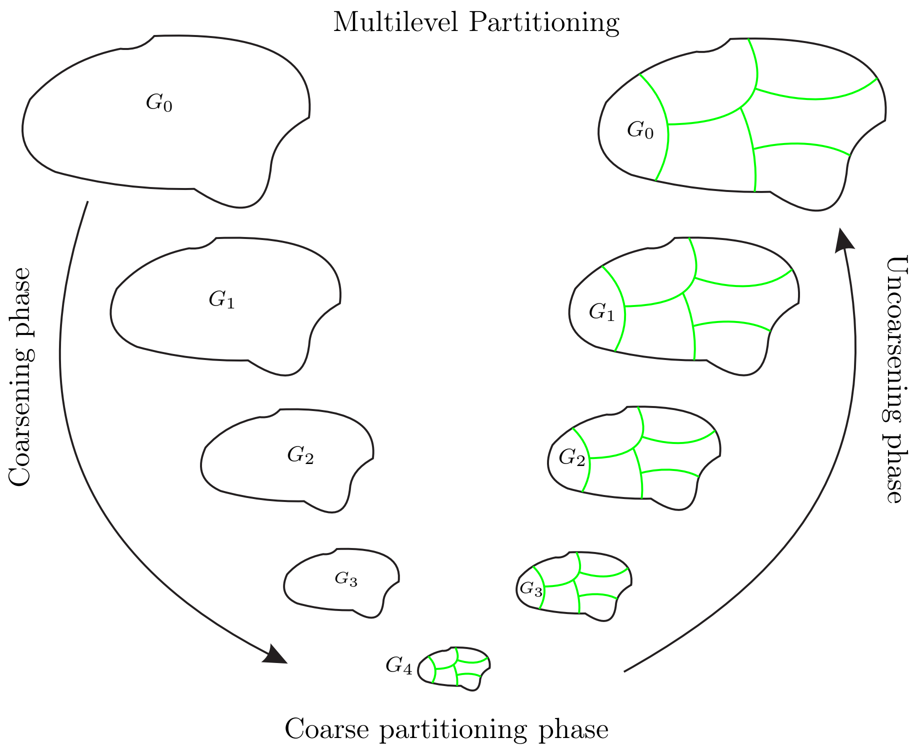


Figure 6.1: The three phases of the multilevel approach: coarsening, coarse partitioning, and uncoarsening. G_n represents the graphs used in the algorithm: G_0 is the source graph, G_1 to G_3 are increasingly coarser graphs generated by the approach, and G_4 is the coarsest graph, also generated by the multilevel approach. This figure is present in the METIS manual [48] (modified).

this value to constrain the size of grouped vertices in the setup whose devices provided the smallest amount of memory. For the other setups, we did not allow that any grouped vertex had a size larger than 0.25 or 0.03125 times the amount of memory provided by the device that provides the smallest amount of memory in the setup. We chose these values based on several tests with different values for the LeNet model.

It is important to note that we build subgraphs differently from METIS. We do that by maintaining some of the original edges in the subgraphs to factor redundant edges out of the cost computation. When METIS groups two vertices, if both vertices have an edge to a vertex in common, METIS sums the weights in these edges and builds a subgraph with only one edge. We, on the other hand, maintain these edges since they have different sources. If we group two vertices that have edges with the same source and weight, then we discard one of these edges. With this process, we can greatly reduce the number of edges, at a larger rate than METIS, while also factoring redundant edges out. Figure 6.2a shows an example of a heavy-edge matching, which groups vertices a and c and vertices b and d according to the edge with the heaviest weight in each vertex. Figure 6.2b shows an example of a two-hop matching, which groups vertices a and b and vertices c and d. We use this example to show how we can factor redundant edges out. We do not sum the edge weights to form only one edge between the grouped vertices S_{ab} and S_{cd}. Our proposal includes data about the vertex sources so that, for instance, when the algorithm groups vertices c and d, it removes the repeated edges with repeated sources. These modifications also work for the heavy-edge matching. In the coarsening phase, METIS generates a fixed number of subgraphs while MDN^{2}PCIoT can receive the number of subgraphs as a parameter defined by the user for larger graphs. For smaller graphs, MDN^{2}PCIoT also defines a fixed number of subgraphs. The ability to define the number of subgraphs influence directly in the amount of time required by the partitioning algorithm as well as the partitioning result.

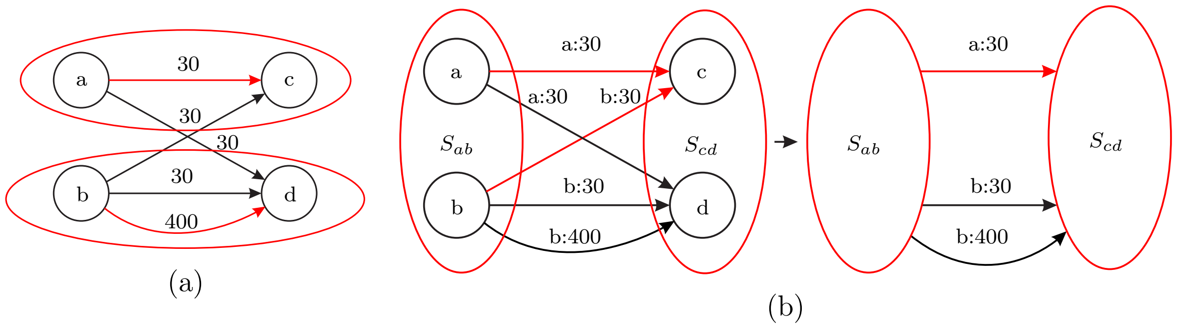


Figure 6.2: An example of (a) a heavy-edge matching and (b) a two-hop matching. We do not sum the edge weights to form only one edge between the grouped vertices S_{ab} and S_{cd}. Our proposal includes data about the vertex sources so that, for instance, when the algorithm groups vertices c and d, it removes the repeated vertices with repeated sources. These modifications also work for the heavy-edge matching.

In the coarse partitioning phase, METIS uses the graph growing approach. In this approach, we apply a breadth-first search in the graph until we visit the total number of vertices divided by the number of partitions. Then, we assign the vertices found by this search into one partition and repeat this process until we assign all the vertices to the partitions. This approach helps in producing coarse partitionings with a smaller amount of communication than random partitionings, for instance. As we want to respect memory constraints, we used the Best Fit approach [20, 92] in the coarse partitioning phase. In the Best Fit approach, we assign each vertex to the partition that fits this vertex and will contain the smallest amount of available memory after the vertex assignment. This approach assigns as many vertices as possible to the same partition, trying to fill it, before using another partition. For homogeneous setups, the Best Fit approach helps in producing partitionings with an amount of communication that is smaller than the amount of communication produced by the graph growing approach. Additionally, it also respects memory constraints.

Also in the coarse partitioning phase, we execute faster-DN^{2}PCIoT to improve the partitioning produced by the Best Fit approach. METIS uses the Kernighan and Lin algorithm based on the modification of Fiduccia and Mattheyses [31] in this phase but stops it if the algorithm produces the same result after 50 moves. We use a similar approach in the faster-DN^{2}PCIoT. For each vertex v, DN^{2}PCIoT searches for the best vertex u that produces the largest improvement in the cost function when DN^{2}PCIoT swaps v for u. In this search, if the cost function produces the same result after a defined number of vertices, then faster-DN^{2}PCIoT stops the search and selects the current best vertex u. The user can choose this number of vertices for larger graphs and we call it swap stabilization. Furthermore, for each vertex v, DN^{2}PCIoT also searches for the best move operation for it. During the search for the best operation and best vertex or vertices that perform them, if the cost function produces the same result after another defined number of vertices, then faster-DN^{2}PCIoT stops the search and chooses the current best vertex v and its respective best operation. The user must choose this number of vertices and we call it step stabilization. With these modifications, the faster-DN^{2}PCIoT executes faster than the original DN^{2}PCIoT in Chapter 5.

After the algorithm finishes partitioning the smallest subgraph, the uncoarsening phase takes place. In this phase, the multilevel algorithm reproduces the partitioning obtained for the smallest subgraph to the subgraph of the next level, making its way back to the original graph. At each subgraph, the multilevel algorithm performs some refinements in the partitioning. We can perform refinements in the partitioning because, for the higher level, larger subgraphs, we have a finer granularity and, in these larger subgraphs, we can move to different partitions vertices that, in the smaller subgraphs, had to be assigned to the same partition because they were grouped. Similar to METIS, in the refinements, we only consider vertices that present communication to vertices at a different partition. Furthermore, METIS applies another version of the Kernighan and Lin algorithm. In this version, the algorithm executes only one epoch for each subgraph so that the partitioning algorithm executes faster. We perform a similar approach in the faster-DN^{2}PCIoT, however, we execute all epochs if the number of devices is smaller than 12 or if the number of devices is smaller than 50 and the subgraph size is smaller than 700 vertices. We defined these numbers based on the LeNet validation tests. It is worth noting that, in the uncoarsening phase, we also used the same modifications listed in the coarse partitioning phase for the faster-DN^{2}PCIoT. With these modifications, the algorithm only considering vertices that present communication to vertices at a different partition, and the epochs that do not execute, the faster-DN^{2}PCIoT algorithm executes even faster than in the coarse partitioning phase.

Algorithm 4 lists the pseudocode for MDN^{2}PCIoT with the three phases explained in this section. First, we allocate a list of subgraphs and add a copy of the original CNN graph into the first position (Line 3). Next, in the coarsening phase, a loop builds the subgraphs, each one based on the previous subgraph (Lines 5 – 7). After that, in the coarse partitioning phase, the algorithm applies the Best Fit approach to the coarsest subgraph and saves the resultant partitioning to *bestP* (Line 9). Then, the algorithm executes the faster-DN^{2}PCIoT algorithm to improve the Best Fit partitioning and saves the result to *bestP* (Line 10). The last phase is the uncoarsening, in which a loop runs through the other subgraphs until the original graph, executing the faster-DN^{2}PCIoT algorithm for each subgraph and the original graph to improve the partitioning (Lines 12 – 14).

6.2 Methods and Materials

In this section, we discuss the CNN models used in this chapter, the setups for each model, and the algorithms that we executed.

6.2.1 Convolutional Neural Network Models

We employed the two LeNet models from Chapter 5, LeNet 2:1 and LeNet 1:1, for the validation of MDN^{2}PCIoT. As a larger neural network, we also used the AlexNet model [54] to build a dataflow graph containing 65916 vertices, which we show in Figure 6.3 with the following per-layer data: the number of vertices in height, width, and depth, the layer type, and the amount of transferred data in bytes required by each edge in each layer. We used AlexNet because it is a more powerful CNN than LeNet and requires a lot more

Algorithm 4 MDN^{2}PCIoT algorithm.

```
function MDN$^2$PCIoT($sourceGraph, numberOfCoarsenedGraphs$)
    $subgraph[numberOfCoarsenedGraphs + 1]$;
    $subgraph[0] \leftarrow sourceGraph$;
    /* Coarsening phase */
    for $n \leftarrow 1$ to $numberOfCoarsenedGraphs$ do
        $subgraph[n] \leftarrow CoarsenGraph(subgraph[n-1])$;
    end for
    /* Coarse partitioning phase */
    $bestP \leftarrow BestFit(subgraph[numberOfCoarsenedGraphs])$;
    $bestP \leftarrow$faster-DN$^2$PCIoT($bestP, subgraph[numberOfCoarsenedGraphs]$);
    /* Uncoarsening phase */
    for $n \leftarrow numberOfCoarsenedGraphs - 1$ to 0 do
        $bestP \leftarrow$faster-DN$^2$PCIoT($bestP, subgraph[n]$);
    end for
    return $bestP$;
end function
```

resources. Furthermore, it was the first CNN that became famous in the computer vision field, however, we can use any neural network that can be represented as a DAG when considering the inference rate maximization and for any neural network when considering the communication reduction, as explained in Section 2.5 of Chapter 2.

In this AlexNet dataflow graph, we again grouped the neurons of the input layer that are in the same position of height and width but different positions in the depth into the same vertex. We performed the same approach in each convolution and pooling layers. We modeled the other neurons as one vertex each. This grouping in the AlexNet model is similar to the grouping in the LeNet 1:1 version. Table 6.1 shows the amount of memory and computation (the number of FLOP per inference) required by each vertex in each layer, the amount of memory required by the shared parameters and biases for each layer, the filter size and stride at each convolution and pooling layers, and the depth size of each layer.

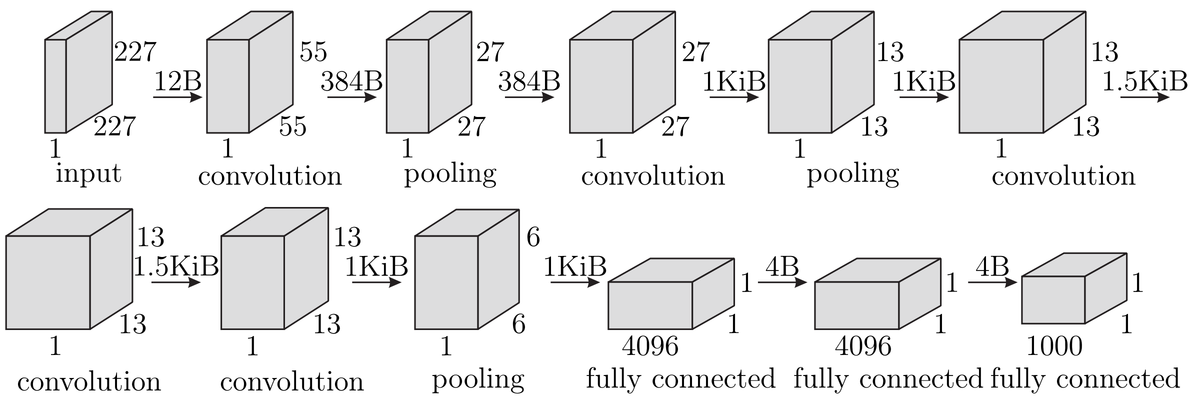


Figure 6.3: AlexNet architecture. Edges represent data transfers and are labeled with the number of bytes per inference that each edge must transfer.

Table 6.1: Characteristics of the AlexNet model used in this chapter.

Feature	**input**	**C1**	**P2**	**C3**	**P4**	**C5**	**C6**	**C7**	**P8**	**FC9**	**FC10**	**FC11**
Memory per vertex (KiB)	0.01	0.38	0.38	1	1	1.5	1.5	1	1	36	16	16
Computation per vertex (kFLOP)	0	69.9	0.86	5	2.3	5.1	7.7	7.7	2.3	18.4	8.2	8.2
Memory of shared parameters and biases per layer (MiB)	0	0.13	0	2.3	0	3.4	5.1	3.4	0	0	0	0
Filter size	N/a*	11x11	3x3	5x5	3x3	3x3	3x3	3x3	3x3	N/a	N/a	N/a
Stride	N/a	4	2	1	2	1	1	1	2	N/a	N/a	N/a
Depth size	3	96	96	256	256	384	384	256	256	4096	4096	1000

* Not applicable.

6.2.2 Device Characteristics

We used the same setups of Chapter 5 for the LeNet model but executed MDN^{2}PCIoT only for the communication reduction objective function. Table 6.2 shows the setups for the AlexNet model with hypothetical devices. All these devices belong to class *others* of constrained devices that we showed in Chapter 2. It is worth noting that, for the AlexNet experiments, no device from class D2 or below could participate in the partitionings. Our setups offer different amounts of memory than common embedded devices because, in many cases, these devices perform other tasks that require memory and only a fraction of the device's memory is available for the CNN application. In the AlexNet experiments, we executed the algorithms for both objective functions, communication minimization and inference rate maximization. We partitioned AlexNet into 2, 4, 8, 16, and 40 devices, decreasing the amount of memory and the computational performance as the number of devices allowed to be used in the experiments increased. We used the same communication performance for all the setups so that the experiments with a smaller number of devices and higher computational power tend to get limited by the communication performance while the experiments with a larger number of devices and lower computational power tend to get limited by the computational performance of the devices. Again, in each experiment, the communication links between each device present the same performance, which is constant during the whole partitioning algorithm.

6.2.3 Algorithms

We employed three algorithms in the experiments. The first algorithm is the Best Fit applied to the original graph representing the CNN. This algorithm presents a complexity of $O(\mathrm{V} \log \mathrm{V})$. The second algorithm is *gpmetis* from METIS, varying the same parameters that we used in Chapter 5: the number of partitions according to the setups, the number of

Table 6.2: Device data and the maximum number of devices allowed to be used in the AlexNet experiments.

Number of devices allowed to be used in the experiments	**Device amount of RAM (MiB)**	**Device estimated computational performance (GFLOP/s)**	**Communication performance between each device (Mbit/s)**
2	183	312	
4	76	156	
8	46	78	300
16	31	39	
40	8	10	

different partitionings to compute, the number of iterations for the refinement algorithms at each stage of the uncoarsening process, the maximum allowed load imbalance among the partitions, and the algorithm's objective function, which can be the *edgecut* minimization or the total communication volume minimization. We report the METIS parameters that led to the METIS results in this chapter in Appendix A. Finally, we employed our new MDN^{2}PCIoT. We executed it with different numbers of subgraphs and chose the number of subgraphs that led to the smallest amount of communication or the largest inference rate after the Best Fit algorithm application in the coarse partitioning phase. Then, we executed MDN^{2}PCIoT completely with the chosen number of subgraphs.

6.3 Experimental Results

In this section, we show and discuss the results for each algorithm of Subsection 6.2.3 for both objective functions and both the LeNet and AlexNet models.

6.3.1 Communication Reduction

In this subsection, we show and discuss the results for communication reduction and both the LeNet and AlexNet models.

LeNet Model

Figure 6.4 shows the communication results for the LeNet 2:1 and LeNet 1:1 models when the objective function is communication reduction. We also plotted in this figure the best results reported in Chapter 5, which were achieved by DN^{2}PCIoT, so that we know the best results achieved in these experiments so far. We normalized the communication cost for each setup and LeNet model, in which 1 represents the best result and the other values indicate how many times the result was worse (larger) than the best result. Additionally, we included, for each setup and LeNet model, the minimum communication cost achieved in the experiments in bytes, which corresponds to the result equal to 1 in each experiment. We indicated these experiments with a red arrow. The error bars in the MDN^{2}PCIoT results show the minimum, median, and maximum values that MDN^{2}PCIoT achieved in

30 executions. Finally, we represented the cases in which METIS could not produce any valid partitionings with an “x”.

We can see that the MDN^{2}PCIoT achieved better results than METIS for all the setups and LeNet models. Comparing to the best results reported in Chapter 5, achieved by DN^{2}PCIoT, MDN^{2}PCIoT was better than or equal to those results in 60% of the experiments. For the other 40% of the experiments, MDN^{2}PCIoT was between 0.6% and 48% worse than DN^{2}PCIoT. Comparing to the Best Fit algorithm applied to the original LeNet 2:1 and LeNet 1:1 models, MDN^{2}PCIoT was better than it for all the setups, except for the 4-device setup with the LeNet 1:1 version, which was 16.7% worse than the Best Fit result. Although MDN^{2}PCIoT did not achieve the best results seen so far, which are the results achieved by DN^{2}PCIoT in Chapter 5, it was still better than the state-of-the-art framework METIS and mostly better than the simple Best Fit algorithm.

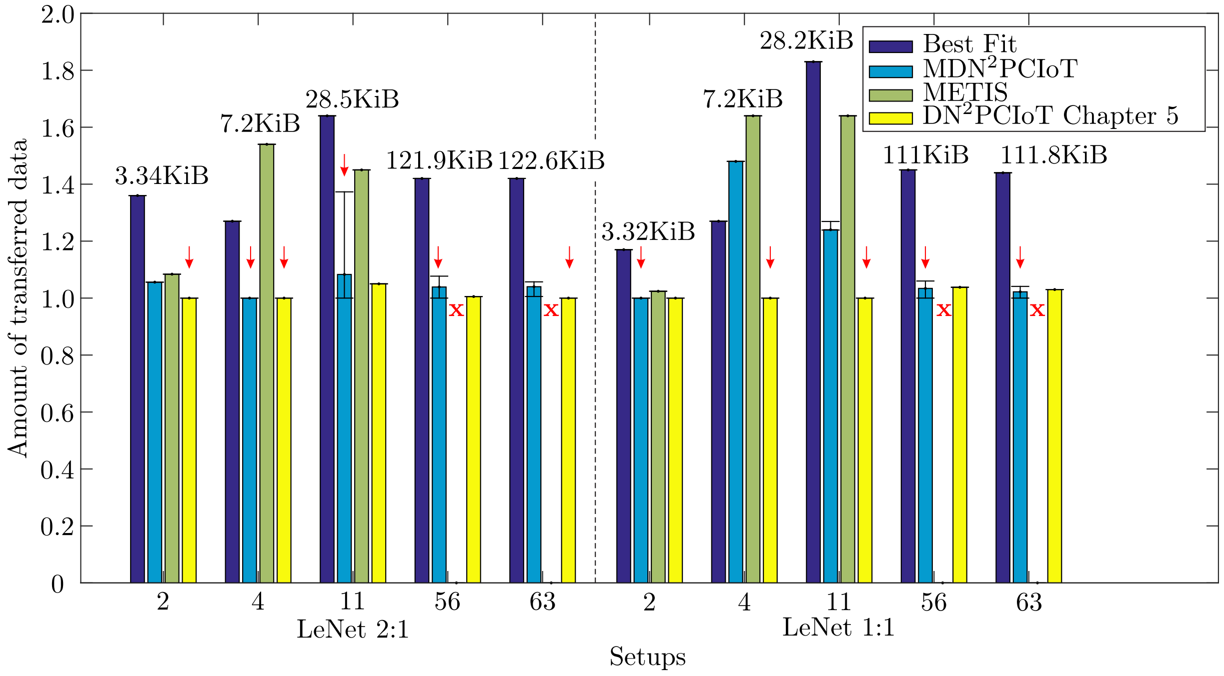


Figure 6.4: Normalized results for the LeNet 2:1 and LeNet 1:1 models with the communication reduction objective function.

AlexNet Model

After the results for the LeNet models using the communication reduction, we considered that MDN^{2}PCIoT was validated and we performed the experiments with the AlexNet model. As DN^{2}PCIoT partitions large graphs in a considerably large time, we proposed MDN^{2}PCIoT to partition these graphs faster and, thus, we did not employ DN^{2}PCIoT in the AlexNet experiments. Figure 6.5 shows the communication results for the AlexNet model when the objective function is communication reduction. We also normalized these results considering the smallest value in each setup equal to 1 and the other values represent how many times the result was larger than the best result. We included, for each setup, the minimum communication cost achieved in the experiments in bytes, which corresponds to the result equal to 1 in each experiment. Additionally, we represented the cases in which METIS could not produce any valid partitionings with an “x”. In this figure, we can see that, different from the LeNet experiments, the Best Fit algorithm applied to the original graph achieved the lowest results. In three setups, the Best Fit algorithm used fewer devices than the provided number of devices. The setup that used devices with 46 MiB used only five devices, although it could have used up to eight devices. The setup that used devices with 31 MiB used only eight devices, although it could have used up to 16 devices, and the setup that used devices with 8 MiB used only 31 devices, although it could have used up to 40 devices. These results show the trend of the Best Fit algorithm of attempting at filling entire devices before assigning computation to another device, using fewer devices than the provided number of devices, if possible.

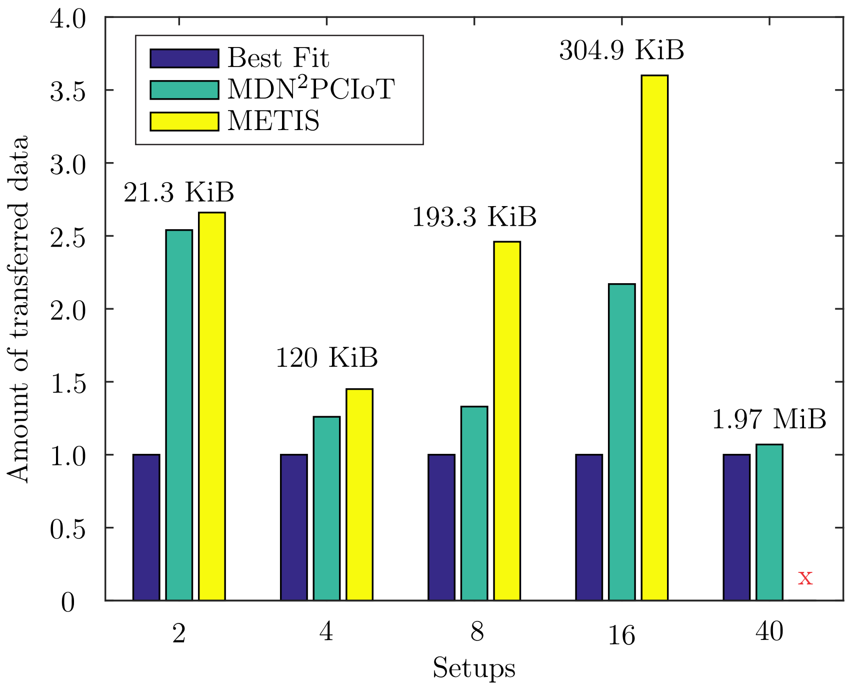


Figure 6.5: Normalized results for the AlexNet model with the communication reduction objective function. The numbers in bytes indicate the communication cost for the smallest result in each setup.

The results achieved by MDN^{2}PCIoT varied from 1.07 to 2.54 times larger than the Best Fit results. The METIS results varied from 1.05 to 1.85 times larger than the MDN^{2}PCIoT results. As the number of devices increases, the difference in the results achieved by METIS and MDN^{2}PCIoT also increases. However, for the most constrained setup, METIS could not achieve any valid partitioning while MDN^{2}PCIoT produced a result that is only 1.07 times larger than the Best Fit result. Thus, although the MDN^{2}PCIoT algorithm achieved worse results than the Best Fit algorithm, the MDN^{2}PCIoT algorithm still achieved partitionings that are always valid for all the setups and results that are better than the METIS results. Nonetheless, as shown in the LeNet results, the Best Fit algorithm does not always lead to the best results.

We can analyze data relative to how MDN^{2}PCIoT and METIS work to understand why MDN^{2}PCIoT is better than METIS. Table 6.3 shows the coarsest graph data for METIS and MDN^{2}PCIoT for each AlexNet setup. The coarsest graph data are the number of subgraphs and the number of vertices and edges in each subgraph. We can see that the number of subgraphs for each algorithm is similar, being equal for three setups, with 2, 8, and 16 devices. However, METIS reduced the number of vertices between 21 and 81 times and the number of edges between 3.5 and 49 times related to the original graph. MDN^{2}PCIoT further reduced the original graph, between 64 and 455 times for the number of vertices and between 26 and 86 for the number of edges related to the original graph. We can explain the larger reduction in the number of vertices due to the different limits for the size of the grouped vertices used by METIS and MDN^{2}PCIoT. For the number of edges, we can explain the larger reduction due to the MDN^{2}PCIoT feature of factoring redundant edges out.

With these data, it is possible to see that MDN^{2}PCIoT not only produces a smaller graph than METIS but also produces a smaller communication cost because it eliminates redundant edges directly in the subgraphs, instead of only in the cost computation. Additionally, the use of the faster-DN^{2}PCIoT algorithm in the coarse partitioning and uncoarsening phases can lead to better results than METIS, as suggested by the results in Chapter 5, which used DN^{2}PCIoT.

Table 6.3: Coarsest graph data for METIS and MDN^{2}PCIoT for each AlexNet setup.

Number of devices allowed to be used in the experiments	**Algorithm**	**Number of subgraphs**	**Number of vertices**	**Number of edges**
2	METIS	7	3,085	6,050,764
	MDN^{2}PCIoT	7	519	487,472
4	METIS	8	1,614	842,932
	MDN^{2}PCIoT	6	1,036	833,690
8	METIS	9	1,124	469,223
	MDN^{2}PCIoT	9	145	250,216
16	METIS	9	818	436,744
	MDN^{2}PCIoT	9	520	303,804
40	METIS	9	1,062	817,106
	MDN^{2}PCIoT	11	300	417,061

6.3.2 Inference Rate Maximization

Figure 6.6 shows the inference rate results for the AlexNet model when the objective function is the inference rate maximization. We again represented the METIS invalid partitionings by an "x". In this figure, we can see that the results of all algorithms were similar for all the setups, except for the most constrained setup with 40 devices. The results were similar because, as we doubled the number of devices in the experiments, we used half of the computational performance for each device, maintaining the total computational performance as we can see in Table 6.2. MDN^{2}PCIoT achieved the best results for most setups, although with a slight improvement. In the 40-device setup, METIS could not produce any valid partitionings due to the memory required by the shared parameters and biases as explained in Chapter 5. The Best Fit algorithm produced a very low inference rate, with the MDN^{2}PCIoT result being 3.09 times better than the Best Fit algorithm result.

The MDN^{2}PCIoT results in this subsection were better than when the objective function was the communication reduction, although most improvements were small, except for the most constrained setup. Nonetheless, METIS could not produce any valid partitionings for very constrained devices. In our scenarios, we aim to maximize the inference rate under very constrained devices. Thus, our results suggest that MDN^{2}PCIoT is an adequate algorithm to partition CNNs among constrained devices, especially very constrained devices.

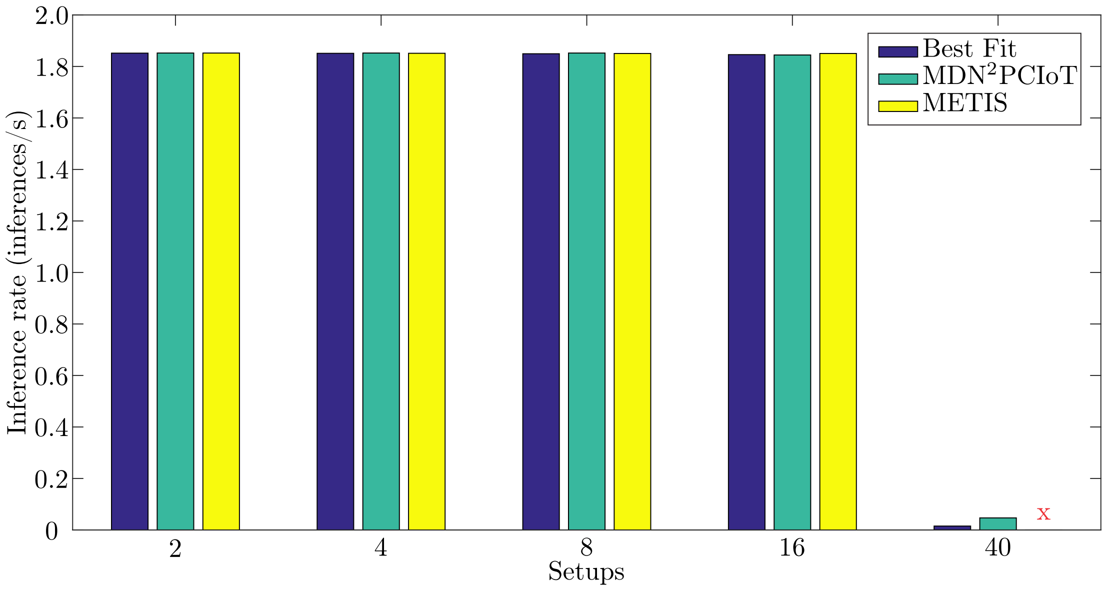


Figure 6.6: Inference rate results for the AlexNet model with the inference rate maximization objective function.

6.4 Discussion

In this section, we analyze data relative to how the MDN^{2}PCIoT algorithm work to understand why the Best Fit algorithm applied to the original graph achieved the best results in the communication minimization objective function. Figure 6.7 shows the result of the Best Fit algorithm applied to each subgraph produced by MDN^{2}PCIoT for the AlexNet model and the two-device setup. This result would be the initial communication value that MDN^{2}PCIoT would try to improve if we chose the respective number of subgraphs. When we compare these initial values to the Best Fit algorithm applied to the original graph (21.3 KiB), we can see that the first subgraphs led to very large communication values, with the first subgraph producing 81 times more communication than the Best Fit algorithm applied to the original graph. The coarse partitioning values decrease until we reach the seventh subgraph, after it, the coarse partitioning increases again. The worsening in the first subgraphs indicates that the grouping in the coarsening phase is harming our subgraphs, at least in the first subgraphs and when using the Best Fit algorithm as the coarse partitioning. This happens even though the coarsening function chooses edges that present the largest weight first, which should help the communication reduction objective function. Thus, the heavy-edge matching may not be the appropriate approach to group vertices when partitioning CNNs among constrained IoT devices for communication reduction.

As the AlexNet data in Figure 6.7 indicate that the coarsening phase may harm our results by increasing the communication cost of the coarse partitioning, we can visually

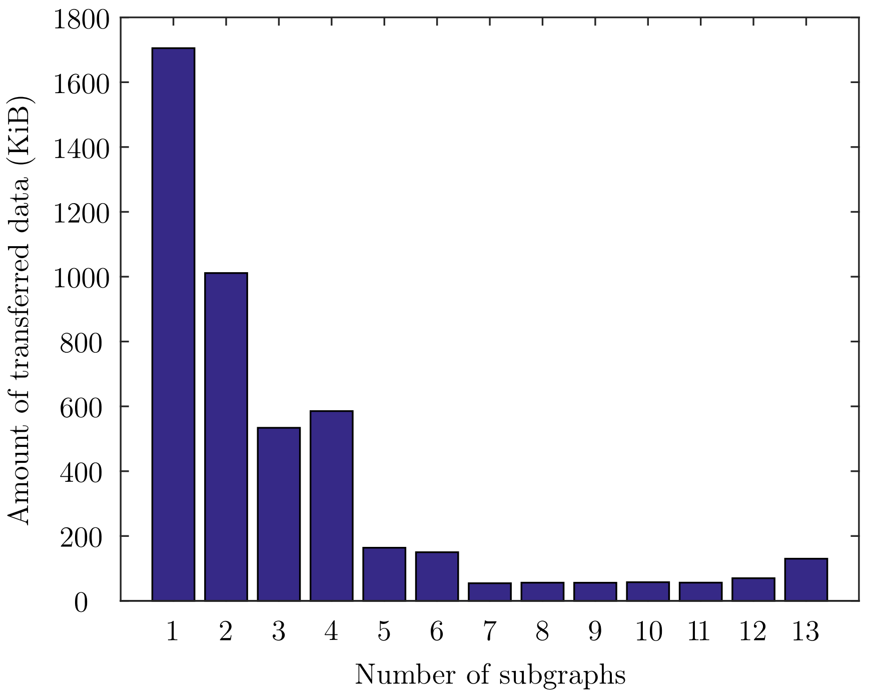


Figure 6.7: Communication cost applying the Best Fit algorithm to each subgraph produced in the coarsening phase.

analyze the grouping in this phase to see why this happened. We chose to analyze the grouping of the LeNet 2:1 model because it is a smaller CNN, thus, easier to visualize. Figure 6.8 shows the LeNet 2:1 grouping in the first step of the coarsening phase. In this figure, the algorithm groups vertices from left to right and from the top to the bottom in each layer. However, the algorithm starts with the vertices with the smallest degree and choose vertices in increasing order of their degree. To show this order, we painted in shades of green the first grouped vertices. They started by the corner vertices in the third layer (P2) because these vertices present the smallest degree. The coarsening phase starts with the heavy-edge matching with modifications. In this step, the algorithm groups vertices from different layers because CNNs only have connections between vertices from different layers. For instance, the algorithm groups the first vertex of layer P2 with the first vertex of layer C3. After the green-colored vertices, we painted in shades of red the next grouped vertices, comprising vertices near the corner in the first and second layers and the first vertex of the last and last but one layers. Next, we painted the grouped vertices in shades of blue, then orange/brown, yellow, grey, and wine. Finally, the two-hop matching takes place and we painted the grouped vertices in shades of purple. In the two-hop matching, the algorithm groups vertices from the same layer. The white vertex in layer FC1 remains ungrouped after the coarsening phase. We also numbered the vertices so that, in the electronic version of this text, it is possible to zoom in and see which vertices the algorithm grouped, which are vertices with equal numbers, and the order in which the algorithm grouped the vertices, starting from zero. We can see, in the first layer, that some grouped purple vertices are not contiguous. Additionally, after the first step of the coarsening phase, the algorithm changes the ordering of the vertices concerning the layers. These two situations harm the communication cost because they increase it. Thus, we can confirm that the heavy-edge matching may not be the appropriate approach to group vertices when partitioning CNNs among constrained IoT devices for communication reduction.

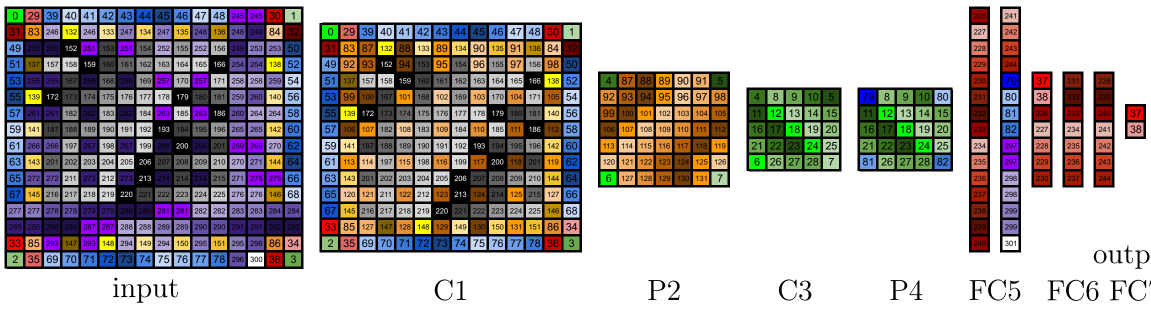


Figure 6.8: LeNet 2:1 grouping in the first step of the coarsening phase.

To solve the limitation in the coarsening phase, we can perform another automatic grouping that is similar to the manual approach of the previous chapters, whose results suggest that it presents a limited impact on the results. As the MDN^{2}PCIoT algorithm could not produce results equal to or better than the Best Fit approach applied to the original graph, another possibility is trying other coarse partitionings in the smallest subgraph so that the algorithm does not fall into local minima, which may also have happened.

6.5 Final Remarks

In this chapter, we proposed the Multilevel Deep Neural Networks Partitioning for Constrained IoT Devices (MDN^{2}PCIoT), a multilevel algorithm that enhances DN^{2}PCIoT, improving its performance and enabling the partitioning of large graphs. In the multilevel approach, the algorithm first coarsens the input graph until it has a small size and partitions this small graph. After that, the algorithm refines the solution until it reaches back the original input graph. We proposed some modifications in relation to METIS in the coarsening and coarse partitioning phases and in DN^{2}PCIoT for the coarse partitioning and uncoarsening phases, leading to the faster-DN^{2}PCIoT algorithm.

We performed experiments using the LeNet models from Chapter 5 and the AlexNet CNN model for five different setups for each CNN. The objective functions were the communication reduction for all the models and the inference rate maximization for the AlexNet model. For the LeNet model and the communication reduction, MDN^{2}PCIoT achieved the best results considering the algorithms used in this chapter, which were the Best Fit algorithm applied to the original graph, METIS, and MDN^{2}PCIoT. MDN^{2}PCIoT was better than or equal to the best LeNet results seen so far, which were the results achieved by DN^{2}PCIoT in Chapter 5, for 60% of the experiments. For the AlexNet model and the communication reduction objective function, the Best Fit algorithm achieved the best results for all the setups, followed by MDN^{2}PCIoT and, then, by METIS, which could not provide any valid partitionings for the most constrained setup. Finally, for the inference rate maximization, the algorithms produced similar results, with MDN^{2}PCIoT achieving the best results for most setups and being 3.09 times better than the Best Fit algorithm. Again, METIS produced only invalid partitionings in the most constrained setup. Thus, our results suggest that MDN^{2}PCIoT is an adequate algorithm to partition CNNs among constrained devices, especially very constrained devices.

We also investigated why MDN^{2}PCIoT did not lead to the best results in the experiments. We plotted the communication results of the Best Fit algorithm applied to each subgraph produced in the coarsening phase for the AlexNet model. In the first subgraph, the communication cost increased 81 times, indicating that either the coarsening phase may harm our results or the Best Fit algorithm may make MDN^{2}PCIoT fall into local minima. To understand why this happened, we painted the LeNet 2:1 model with the grouping after the first step of the coarsening phase and confirmed that the heavy-edge matching may not be an adequate coarsening algorithm for CNNs.

In the next chapter, we conclude this thesis summarizing our work and highlighting the proposed algorithms and our contributions. In the end, we list our publications and present future directions for this research.

Chapter 7

General Conclusions and Future Perspectives

In this chapter, we conclude this thesis with our contributions, publications arisen from this work, limitations and difficulties we encountered during the project execution, and the future perspectives.

7.1 Contributions

The Deep Neural Network partitioning is a solution to the problem of executing the inference of large DNNs on constrained IoT devices. We can execute a partitioning algorithm to find better distributions of the DNN model computations according to some specific objective function. This process can enable the inference execution of large DNNs on constrained IoT devices, which would not be possible if we used only a single constrained device due to memory limitations. Additionally, we can improve the performance of this execution with the DNN model computation distribution. This thesis contributes to the areas of Convolutional Neural Networks, Internet of Things, and partitioning algorithms by proposing new algorithms to partition CNNs among constrained IoT devices, which consider CNN specificities and device resource limitations. These algorithms can also be used for other domains, thus, they are general-purpose partitioning algorithms that take into account the system resources. We analyzed CNNs within constrained IoT scenarios and showed that it is viable to execute the inference of CNNs on very constrained devices.

We proposed four algorithms, three of which were inspired by the Kernighan and Lin algorithm. The other algorithm is a simple greedy algorithm that aims for communication reduction. The first algorithm aims for communication reduction, takes into account the memory restriction of each device, always produces a valid partitioning, factor redundant edges out of the cost computation, and exhibits better performance than the per-layer distribution approach adopted by popular machine learning and machine learning for IoT frameworks such as TensorFlow, DIANNE, and DeepX. The second algorithm can partition CNNs both for communication reduction or inference rate maximization, also takes into account the memory restriction of each device, always producing a valid partitioning, and models properly the amount of memory required by the shared parameters and biases of CNNs so that we can employ devices that are more constrained in memory. It also performs better than the approaches adopted by popular machine learning frameworks and by the general-purpose partitioning framework METIS. Finally, the third algorithm wraps the second algorithm in a multilevel approach. In this approach, the algorithm automatically reduces the graph size so that the partitioning algorithm executes faster and, after that, the algorithm performs refinements at each subgraph produced in the first phase until it reaches the original graph and the algorithm finishes its execution. All the features of the second algorithm are present in the third algorithm, except for initializing from a defined partitioning. However, the third algorithm did not lead to the best results for all CNN models and setups, losing to the second algorithm in some cases or to the Best Fit algorithm in others.

We list the main contributions of this work as follows:

i. We proposed the Kernighan-and-Lin-based Partitioning algorithm, which partitions CNNs for distributed execution of the inference on constrained IoT devices. This algorithm aims for communication reduction, always produces partitionings that respect the amount of memory of the devices, and considers the amount of memory of each device independently, allowing devices with different amounts of memory. Additionally, different from current machine learning frameworks, this algorithm allows the neurons of each layer to be assigned to different devices. We proposed several manual partitionings, being the per-layer partitioning equivalent to popular machine learning frameworks such as TensorFlow, DIANNE, and DeepX, and showed that KLP can achieve better results than all the approaches and that different layers with different communication patterns induce different partitionings. We also showed experiments using several different homogeneous and heterogeneous scenarios that support our results.

ii. We extended KLP to propose the Deep Neural Networks Partitioning for Constrained IoT Devices algorithm, which optimizes the partitioning for inference rate maximization or communication reduction. This new algorithm counts more precisely the amount of memory required by the shared parameters and biases of CNNs in each partition, which allows valid partitionings even when we employ more constrained setups in the applications.

iii. We performed five case studies using DN^{2}PCIoT and several scenarios. In the first case study, the results show that the DN^{2}PCIoT algorithm can produce partitionings that achieve higher inference rates and can produce valid partitionings for very constrained IoT setups. We also performed another case study by employing partitioning strategies offered by popular machine learning frameworks such as TensorFlow, DIANNE, and DeepX and showed that they may not be able to execute DNN models on very constrained devices due to their per-layer partitioning approach. The third study involved the METIS framework and indicated that it is not an appropriate framework to partition DNNs for constrained IoT setups because it may not provide valid partitionings under these conditions. We performed an analysis of the DNN model granularity results to show that our DNN with more grouping minimally affects the partitioning result. We also performed an analysis of how profitable it is to distribute the inference execution among multiple constrained devices for the proposed setups. Finally, we analyzed the trade-off between communication and inference rate, showing that DN^{2}PCIoT provides the best trade-offs for all the proposed setups.

iv. We proposed a greedy algorithm to reduce the number of communications based on the available amount of memory of the devices.

v. We proposed the Multilevel Deep Neural Networks Partitioning for Constrained IoT Devices (MDN^{2}PCIoT), a multilevel algorithm that enhances DN^{2}PCIoT and uses the faster-DN^{2}PCIoT algorithm as the partitioning algorithm. The multilevel approach aims at decreasing the DN^{2}PCIoT execution time, allowing the partitioning of larger graphs. The faster-DN^{2}PCIoT algorithm includes the smaller number of combinations that the algorithm tests and the smaller number of epochs that the algorithm executes for each subgraph of the multilevel approach. We proposed a more flexible heavy-edge matching coarsening phase that, depending on the amount of memory of the devices in the partitioning, groups the vertices limiting to either a percentage of the amount of memory provided by the device that provides the smallest amount of memory in the partitioning or to the total amount of memory required by the vertices, excluding the memory required by the shared parameters and biases. Also in the coarsening phase, we allow the DN^{2}PCIoT feature of factoring redundant edges out by modifying the edge treatment when grouping vertices. We proposed an always-valid coarse partitioning for the smallest, coarsest graph, which satisfies memory constraints since the algorithm beginning. Finally, we proposed a more flexible uncoarsening phase that executes either one epoch of DN^{2}PCIoT or the whole DN^{2}PCIoT algorithm depending on the number of vertices of the subgraphs and the number of devices in the partitioning. We performed experiments to validate the MDN^{2}PCIoT algorithm and also experiments with a larger CNN for the inference rate maximization and communication reduction objective functions.

vi. Finally, we conclude that we can generalize our three main algorithms, which are KLP, DN^{2}PCIoT, and MDN^{2}PCIoT, to produce partitionings to any hardware and any kind of computation that can be expressed as a DAG for the inference rate maximization and any graph for the communication reduction. We also enable DNN developers to easily choose how to assign the CNN computations to IoT devices, expanding and, thus, facilitating the spread of intelligent sensors in the world.

7.2 Publications Arisen from this Thesis

This work resulted in the following publications (items represented by the star symbol are directly related to this thesis):

Journal papers:

⋆ Fabíola Martins Campos de Oliveira and Edson Borin. Partitioning Convolutional Neural Networks to Maximize the Inference Rate on Constrained IoT Devices. *Future Internet*, 11(10), 2019. ISSN 1999-5903. DOI: 10.3390/fi11100209. URL: `https://www.mdpi.com/1999-5903/11/10/209`.

• Lucas de Magalhães Araújo, Fabíola Martins Campos de Oliveira, Jorge Henrique Faccipieri, Tiago Antonio Coimbra, Sandra Avila, Martin Tygel, and Edson Borin. Detecção de estruturas em dados sísmicos com Deep Learning. *Boletim SBGf*, 104, 18-21, 2018. ISSN 2177-9090. URL: `https://sbgf.org.br/noticias/2018/08/21/693/`

Conference papers:

- João Antonio Magri Rodrigues, Fabíola Martins Campos de Oliveira, Renata Spolon Lobato, Roberta Spolon, Aleardo Manacero, and Edson Borin. Improving Virtual Machine Consolidation for Heterogeneous Cloud Computing Datacenters. In *Proceedings of the 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Campo Grande, Brazil*, pages 176–179, 15–18 October 2019. DOI: 10.1109/SBAC-PAD.2019.00037. URL: `http://doi.org/10.1109/SBAC-PAD.2019.00037`.

⋆ Fabíola Martins Campos de Oliveira and Edson Borin. Partitioning Convolutional Neural Networks for Inference on Constrained Internet-of-Things Devices. In *Proceedings of the 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Lyon, France*, pages 266–273, 24–27 September 2018. DOI: 10.1109/CAHPC.2018.8645927. URL: `https://doi.org/10.1109/CAHPC.2018.8645927`.

Book chapter:

⋆ Flávia Pisani, Fabíola Martins Campos de Oliveira, Eduardo de Souza Gama, Roger Immich, Luiz Fernando Bittencourt, and Edson Borin. Fog Computing on Constrained Devices: Paving the Way for the Future IoT. In *Series: Advances in Parallel Computing, Ebook: Advances in Edge Computing: Massive Parallel Processing and Applications*. Fatos Xhafa and Arun Kumar Sangaiah, Editors, IOS Press, 2020. 35, pages 22–60. ISBN: 978-1-64368-062-0. DOI: 10.3233/APC200003. URL: `https://doi.org/10.3233/APC200003`.

7.3 Limitations and Difficulties

In this section, we discuss the limitations and difficulties found in this research work. For instance, inherent to any experimentation that demands long execution times, both energy and equipment failures and the budget limit for cloud execution motivated the implementation of some features in our algorithms. We stored the state of the current epoch produced by DN^{2}PCIoT through the store of the current epoch partitioning and the objective function result of this partitioning, i.e., the communication cost or inference rate. Furthermore, we implemented the DN^{2}PCIoT feature of starting from a defined partitioning so that DN^{2}PCIoT could start from the stored state in the case of failures.

Also considering the possibility of equipment failures and the large computational complexity of DN^{2}PCIoT, we parallelized the DN^{2}PCIoT execution using OpenMP to accelerate it. We parallelized the function *findBestValidOperation()* so that DN^{2}PCIoT searches for the best operation for each vertex in parallel.

We implemented other features related to large execution time in MDN^{2}PCIoT such as the multilevel approach and the faster-DN^{2}PCIoT algorithm. The faster-DN^{2}PCIoT algorithm can execute only one epoch per subgraph in MDN^{2}PCIoT and can reduce the search for the best operation in the function *findBestValidOperation()* using the step and swap stabilizations. Another possibility is to use an early stop approach so that the algorithm stops if, after two epochs, the algorithm does not improve the result by a defined threshold.

We intended to use TensorFlow to generate the neural network dataflow graph, however, TensorFlow does not provide a full expanded graph with each vertex and edge, instead, it provides the dataflow graph at the level of operations, for instance, convolution or pooling. Thus, we had to write a program to generate a dataflow graph representing the CNN, which currently always groups the neurons in the depth dimension. This program needs the complete data of CNNs: number of layers, number of shared parameters and biases per layer, layer depth, width, and height, filter size, and stride in each operation.

We had to perform several experiments to determine empirically the number of executions that were sufficient to achieve the best result in our algorithm. We varied the number of executions up to 300 executions but found that 30 executions were a reasonable number of executions.

We implemented several programs to generate dataflow graphs with increasing granularity. In the case of LeNet, we used graphs that did not group any neurons in the width and height dimensions (LeNet 1:1), graphs that grouped 2x2 neurons in these dimensions (LeNet 2:1), graphs that grouped 4x4, 8x8, and 16x16 neurons, and a per-layer graph. While the LeNet with the 8x8 grouping achieved the best result found with 30 executions of LeNet 1:1 and LeNet 2:1 once in 100 executions, the LeNet with the 16x16 grouping did not achieve this result even in 300 executions. Thus, we concluded that these groupings are too limiting for the partitioning. Finally, with the 4x4 grouping, we had unequal layer groupings and unequal data transfers of each edge within a layer, therefore, we had to change our algorithm to consider these cases. However, we could not devise a solution that managed these unequal data transfers and could not perform any experiment with this grouping. Fortunately, this granularity level was not essential to our work.

The Kernighan and Lin's algorithm chooses the best swap through the ordering of the vertices of each partition according to the gain in the communication cost that each vertex provides if it is moved to another partition. However, when we added the feature of factoring redundant edges out of the cost computation, this gain of each vertex is dynamic and depends on the partition that the vertex is currently assigned and the partition to where the vertex can be moved. We attempted at a differential solution that computed the new cost of a partitioning after a vertex move using the current cost, however, we could not devise such a solution. If this solution exists, it would reduce the computational complexity of our algorithms.

SCOTCH cannot partition graphs whose vertices do not perform any computation, i.e., vertices whose size is zero. At first, we followed this condition in the experiments with DN^{2}PCIoT and METIS, setting the size of the input layer vertices, which do not perform any computation, as one and the size of vertices that performed some computation as at least 1000 times larger than the input layer vertices. Then, we realized that both our algorithms and METIS allowed vertices whose size is zero and reexecuted the experiments.

To partition the coarsest graph in the MDN^{2}PCIoT algorithm, we also attempted at using the Worst Fit approach [20, 92]. In this approach, as opposed to the Best Fit approach, we assign each vertex to the partition that will contain the largest amount of memory after the vertex is assigned. To the best of our knowledge, for homogeneous setups, this approach distributes the vertices evenly and is a better approach when the objective function is the inference rate maximization. However, we could not provide valid partitionings and, thus, used the Best Fit approach for both objective functions.

Our algorithms do not consider the possibility of failures, disconnection, and mobility of the devices. Although these conditions may happen in an IoT scenario, they were out of the scope of this work. Nonetheless, an execution framework for DNNs on IoT devices should consider these conditions and we leave such a framework for future work.

Finally, a limitation of our algorithms is execution time. Although we had a large improvement in the execution time with MDN^{2}PCIoT, this improvement was smaller than our expectations. In the next section, we discuss future directions to this research work, including future work related to this section.

7.4 Future Perspectives

There exist many challenges to face in the design of distributed execution of DNNs to IoT devices. As a consequence of this work, some challenges should be investigated. We started investigating some of these challenges, however, we consider that they need a deeper investigation.

From the work with DN^{2}PCIoT, one may employ the iRgreedy approach that we used in LeNet 1:1 as initial partitioning for any CNN when the objective function is the inference rate maximization. Additionally, all the proposed algorithms can be explored with different objective functions such as energy consumption or multi-objective functions such as reducing communication while achieving a desired value for the inference rate.

Regarding the MDN^{2}PCIoT algorithm, one may perform more analyses for the multilevel algorithm and improve it so that it achieves results closer to the DN^{2}PCIoT results. For this purpose, experiments can be performed to evaluate which phase of the multilevel approach should be improved and why we got results that are worse than the Best Fit approach. It is possible to investigate the relation between the subgraphs created by MDN^{2}PCIoT and the Best Fit result in each subgraph. One may also propose an automatic grouping that is similar to the manual approach of Chapters 4 and 5. Another possibility is trying other coarse partitionings in the smallest subgraph. Finally, it is possible to improve the refinement phase to achieve better results, trading execution time. One may also improve the method validation with experiments that use different CNN models and heterogeneous and more constrained setups. Further investigation can be performed in the experiments for inference rate maximization so that one knows if the results were limited by the communication performance or by the device computational performance.

Finally, as general directions, it is possible to execute the proposed algorithms to partition larger CNNs and other types of DNNs, including any type of DNN when considering communication reduction and any type of DNN that can be represented as a DAG when considering the inference rate maximization. The proposed algorithms can also be executed to partition DNN dataflow graphs that do not group the depth neurons. This would allow a finer-grained partitioning and allow devices that are even more constrained. Additionally, one may analyze how different communication technologies such as 5G, LoRa, ZigBee, LTE, and others affect the neural network partitioning performance. Another possibility is to test more scenarios with heterogeneous configurations. One may develop an Application Programming Interface (API) and a programming model to allow automatic code generation and portability. Furthermore, the distributed DNNs can be experimentally tested to check our model and results. Finally, it is possible to build or use a framework that can employ the proposed algorithms and allows for a pipelined execution and an online partitioning and scheduling for fault tolerance. This framework must consider the possibility of failures, disconnection, and mobility of the devices.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: a System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16), Savannah, GA, USA*, pages 265–283, May 2–4 November 2016. ISBN 978-1-931971-33-1.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. URL `http://tensorflow.org/`.

[3] Nurulain Abd Mubin, Eiswary Nadarajoo, Helmi Zulhaidi Mohd Shafri, and Alireza Hamedianfar. Young and mature oil palm tree detection and counting using convolutional neural network deep learning method. *International Journal of Remote Sensing*, 40(19, SI):7500–7515, OCT 2 2019. ISSN 0143-1161. doi: 10.1080/01431161.2019.1569282.

[4] Ossama Abdel-Hamid, Li Deng, and Dong Yu. Exploring Convolutional Neural Network structures and optimization techniques for speech recognition. In Frédéric Bimbot, Christophe Cerisara, Cécile Fougeron, Guillaume Gravier, Lori Lamel, François Pellegrino, and Pascal Perrier, editors, *INTERSPEECH 2013, 14th Annual Conference of the International Speech Communication Association, Lyon, France, August 25-29, 2013*, pages 3366–3370. ISCA, 2013. URL `http://www.isca-speech.org/archive/interspeech_2013/i13_3366.html`.

[5] Ossama Abdel-Hamid, Abdel-Rahman Mohamed, Hui Jiang, Li Deng, Gerald Penn, and Dong Yu. Convolutional Neural Networks for Speech Recognition. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 22(10):1533–1545, October 2014. ISSN 2329-9290. doi: 10.1109/TASLP.2014.2339736. URL `http://doi.org/10.1109/TASLP.2014.2339736`.

[6] Zakwan Al-Arnaout, Jonathan Hart, Qiang Fu, and Marcus Frean. MP-DNA: a novel distributed replica placement heuristic for WMNs. In *Proceedings of the 37th Annual IEEE Conference on Local Computer Networks, Clearwater, USA*, pages 593–600, Oct. 22–25 October 2012.

[7] Angelos Angelopoulos, Emmanouel T. Michailidis, Nikolaos Nomikos, Panagiotis Trakadas, Antonis Hatziefremidis, Stamatis Voliotis, and Theodore Zahariadis. Tackling Faults in the Industry 4.0 Era—A Survey of Machine-Learning Solutions and Key Aspects. *Sensors*, 20(1), 2020. ISSN 1424-8220. doi: 10.3390/s20010109. URL `https://www.mdpi.com/1424-8220/20/1/109`.

[8] Atmel. Atmel SAM G55G. `http://ww1.microchip.com/downloads/en/devicedoc/Atmel-11289-32-bit-Cortex-M4-Microcontroller-SAM-G55_Datasheet.pdf`, 2016. Accessed: July 24, 2019.

[9] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787 – 2805, 2010. ISSN 1389-1286. doi: 10.1016/j.comnet.2010.05.010. URL `http://www.sciencedirect.com/science/article/pii/S1389128610001568`.

[10] Pinkesh Badjatiya, Shashank Gupta, Manish Gupta, and Vasudeva Varma. Deep Learning for Hate Speech Detection in Tweets. In *Proceedings of the 26th International Conference on World Wide Web Companion*, WWW '17 Companion, pages 759–760. International World Wide Web Conferences Steering Committee, April 2017. ISBN 978-1-4503-4914-7. doi: `10.1145/3041021.3054223`.

[11] José I. Benedetto, Luis A. González, Pablo Sanabria, Andrés Neyem, and Jaime Navón. Towards a practical framework for code offloading in the Internet of Things. *Future Generation Computer Systems*, 92:424 – 437, 2019. ISSN 0167-739X. doi: 10.1016/j.future.2018.09.056. URL `http://www.sciencedirect.com/science/article/pii/\S0167739X18302310`.

[12] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU Math Expression Compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

[13] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog Computing and Its Role in the Internet of Things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16. ACM, 2012. ISBN 978-1-4503-1519-7. doi: 10.1145/2342509.2342513. URL `http://doi.acm.org/10.1145/2342509.2342513`.

[14] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. *Fog Computing: A Platform for Internet of Things and Analytics*, pages 169–186. Springer International Publishing, 2014. ISBN 978-3-319-05029-4. doi: 10.1007/978-3-319-05029-4_7. URL `http://dx.doi.org/10.1007/978-3-319-05029-4_7`.

[15] Carsten Bormann, Mehmet Ersue, and Ari Keranen. Terminology for Constrained-Node Networks. Technical report, Internet Engineering Task Force, May 2014. URL `http://www.rfc-editor.org/info/rfc7228`. Accessed on April 4, 2019.

[16] Bo Cao, Xiaofeng Gao, Guihai Chen, and Yaohui Jin. NICE: network-aware VM Consolidation scheme for Energy Conservation in Data Centers. In *Proceedings of the 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS), Hsinchu, Taiwan*, pages 166–173, Dec. 16–19 December 2014.

[17] Mung Chiang and Tao Zhang. Fog and IoT: An Overview of Research Opportunities. *IEEE Internet of Things Journal*, 3(6):854–864, December 2016. ISSN 2327-4662. doi: 10.1109/JIOT.2016.2584538.

[18] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 571–582. USENIX Association, 2014. ISBN 978-1-931971-16-4. URL `https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chilimbi`.

[19] Inc. Cisco Systems. Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update. `https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-738429.html`, 2019. Accessed: July 22, 2019.

[20] Edward Grady Coffman, Michael Randolf Garey, and David Stifler Johnson. *Approximation Algorithms for Bin-Packing — An Updated Survey*, pages 49–106. Springer Vienna, Vienna, 1984. ISBN 978-3-7091-4338-4. doi: 10.1007/978-3-7091-4338-4_3. URL `https://doi.org/10.1007/978-3-7091-4338-4_3`.

[21] François Pellegrini. Distillating knowledge about SCOTCH. In Uwe Naumann, Olaf Schenk, Horst D. Simon, and Sivan Toledo, editors, *In Combinatorial Scientific Computing, Proceedings of the Dagstuhl Seminar, Dagstuhl, Germany, 3–8 May 2009.* Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik: Dagstuhl, Germany, 2009. 12 pages.

[22] Ronan Collobert, Samy Bengio, and Johnny Marithoz. Torch: A Modular Machine Learning Software Library. `https://infoscience.epfl.ch/record/82802/files/rr02-46.pdf`, 2002.

[23] Juan Colonna, Tanel Peet, Carlos Abreu Ferreira, Alípio M. Jorge, Elsa Ferreira Gomes, and João Gama. Automatic Classification of Anuran Sounds Using Convolutional Neural Networks. In *Proceedings of the Ninth International C* Conference on Computer Science & Software Engineering*, C3S2E '16, pages 73–78. ACM, 2016. ISBN 978-1-4503-4075-5. doi: 10.1145/2948992.2949016. URL `http://doi.acm.org/10.1145/2948992.2949016`.

[24] Marcos Dias de Assunção, Alexandre Silva Veith, and Rajkumar Buyya. Distributed Data Stream Processing and Edge Computing. *Journal of Network and Computer Applications*, 103(C):1–17, February 2018. ISSN 1084-8045. doi: 10.1016/j.jnca.2017.12.001. URL `https://doi.org/10.1016/j.jnca.2017.12.001`.

[25] Elias de Coninck, Tim Verbelen, Bert Vankeirsbilck, Steven Bohez, Sam Leroux, and Pieter Simoens. DIANNE: Distributed Artificial Neural Networks for the Internet of Things. In *Proceedings of the 2nd Workshop on Middleware for Context-Aware*

Applications in the IoT (M4IoT 2015), Vancouver, Canada, pages 19–24, Dec. 7–11 December 2015. ISBN 978-1-4503-3731-1.

[26] Elias de Coninck, Tim Verbelen, Bert Vankeirsbilck, Steven Bohez, Pieter Simoens, Piet Demeester, and Bart Dhoedt. Distributed neural networks for Internet of Things: the Big-Little approach. In *Proceedings of the 2nd EAI International Conference on Software Defined Wireless Networks and Cognitive Technologies for IoT*, pages 1–9, Rome, Italy, 26–27 October 2015.

[27] Lucas de Magalhães Araújo, Fabíola Martins Campos de Oliveira, Jorge Henrique Faccipieri, Tiago Antonio Coimbra, Sandra Avila, and Martin Tygel. Detecção de estruturas em dados sísmicos com Deep Learning. *Boletim SBGf*, 1(104):18–21, Jul 2018. ISSN 2177-9090.

[28] Fabíola Martins Campos de Oliveira and Edson Borin. Partitioning Convolutional Neural Networks for Inference on Constrained Internet-of-Things Devices. In *Proceedings of the 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Lyon, France*, pages 266–273, 24–27 September 2018. URL https://doi.org/10.1109/CAHPC.2018.8645927. © 2018 IEEE. Reprinted (modified), with permission.

[29] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large Scale Distributed Deep Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems*, NIPS'12, pages 1223–1231. Curran Associates Inc., 2012. URL http://dl.acm.org/citation.cfm?id=2999134.2999271.

[30] Samira Ebrahimi Kahou, Vincent Michalski, Kishore Konda, Roland Memisevic, and Christopher Pal. Recurrent Neural Networks for Emotion Recognition in Video. In *Proceedings of the 2015 ACM on International Conference on Multimodal Interaction*, ICMI '15, pages 467–474. ACM, 2015. ISBN 978-1-4503-3912-4. doi: 10.1145/2818346.2830596. URL http://doi.acm.org/10.1145/2818346.2830596.

[31] Charles M. Fiduccia and Robert M. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. In *19th Design Automation Conference*, pages 175–181, June 1982. doi: 10.1109/DAC.1982.1585498.

[32] Mouzhi Ge, Hind Bangui, and Barbora Buhnova. Big Data for Internet of Things: A Survey. *Future Generation Computer Systems*, 87:601 – 614, 2018. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2018.04.053. URL http://www.sciencedirect.com/science/article/pii/S0167739X17316953.

[33] Elsy Gómez-Ramos and Francisco Venegas-Martínez. A Review of Artificial Neural Networks: How Well Do They Perform in Forecasting Time Series? *Analítika*, 6(2):7–15, December 2013. URL https://ideas.repec.org/a/inp/inpana/v6y2013i2p7-15.html.

[34] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press: Cambridge, MA, USA, 2016; ISBN 978-0262035613.

[35] Matteo Grimaldi, Valerio Tenace, and Andrea Calimera. Layer-Wise Compressive Training for Convolutional Neural Networks. *Future Internet*, 11(1), 2018. ISSN 1999-5903. doi: 10.3390/fi11010007. URL https://www.mdpi.com/1999-5903/11/1/7.

[36] Visual Geometry Group. VGGNet model. http://www.robots.ox.ac.uk/~vgg/research/very_deep/, 2014. Accessed: March 15, 2017.

[37] Alessio Guerrieri and Alberto Montresor. Distributed Edge Partitioning for Graph Processing. *Computer Research Repository*, abs/1403.6270, 2014. URL http://arxiv.org/abs/1403.6270.

[38] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic Network Surgery for Efficient DNNs. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS'16), Barcelona, Spain*, pages 1387–1395, 5–10 December 2016. ISBN 978-1-5108-3881-9. URL http://dl.acm.org/citation.cfm?id=3157096.3157251.

[39] Song Han, Jeff Pool, John Tran, and William Dally. Learning both Weights and Connections for Efficient Neural Network. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28, Proceedings of the 29th Conference on Neural Information Processing Systems*, pages 1135–1143. Curran Associates, Inc.: Red Hook, USA, 2015, Montréal, Canada, 7–12 December 2015. URL http://papers.nips.cc/paper/5784-learning-both-weights-and-connections-for-efficient-neural-network.pdf.

[40] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. ResNet model. https://github.com/gcr/torch-residual-networks, 2015. Accessed: March 15, 2017.

[41] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, June 2016. doi: 10.1109/CVPR.2016.90.

[42] John Honovich. Frame Rate Guide for Video Surveillance. https://ipvm.com/reports/frame-rate-surveillance-guide, 2014. Accessed: July 14, 2019.

[43] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-Excitation Networks. *Computer Research Repository*, abs/1709.01507, 2017. URL http://arxiv.org/abs/1709.01507.

[44] Yanping Huang, Yonglong Cheng, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, and Zhifeng Chen. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. *Computer Research Repository*, abs/1811.06965, 2018. URL http://arxiv.org/abs/1811.06965.

[45] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, pages 675–678. ACM, 2014. ISBN 978-1-4503-3063-3. doi: 10.1145/2647868.2654889. URL http://doi.acm.org/10.1145/2647868.2654889.

[46] Zhihao Jia, Sina Lin, Charles Ruizhongtai Qi, and Alex Aiken. Exploring Hidden Dimensions in Accelerating Convolutional Neural Networks. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 2274–2283, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR. URL `http://proceedings.mlr.press/v80/jia18a.html`.

[47] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In *Proceedings of the 2nd Conference on Systems and Machine Learning (SysML)*, pages 1–13, 31–2 Apr 2019. URL `https://mlsys.org/Conferences/2019/`.

[48] George Karypis. METIS A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version 5.1.0. `http://glaros.dtc.umn.edu/gkhome/fetch/sw/metis/manual.pdf`, 2019. Accessed: March 30, 2019.

[49] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1): 359–392, December 1998. ISSN 1064-8275. doi: 10.1137/S1064827595287997. URL `http://dx.doi.org/10.1137/S1064827595287997`.

[50] Edward M. Kasprzak and Kemper E. Lewis. Pareto analysis in multiobjective optimization using the collinearity theorem and scaling method. *Structural and Multidisciplinary Optimization*, 22(3):208–218, Oct 2001. ISSN 1615-1488. doi: 10.1007/s001580100138. URL `https://doi.org/10.1007/s001580100138`.

[51] Brian Wilson Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, Feb. 1970. ISSN 0005-8580. doi: 10.1002/j.1538-7305.1970.tb01770.x.

[52] SeungJun Kim, Jaemin Chun, and Anind K. Dey. Sensors Know When to Interrupt You in the Car: Detecting Driver Interruptibility Through Monitoring of Peripheral Interactions. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 487–496. ACM, 2015. ISBN 978-1-4503-3145-6. doi: 10.1145/2702123.2702409. URL `http://doi.acm.org/10.1145/2702123.2702409`.

[53] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *Computer Research Repository*, abs/1404.5997, 2014. URL `http://arxiv.org/abs/1404.5997`.

[54] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., Dec. 2012.

[55] Stanford Vision Lab. ImageNet Large Scale Visual Recognition Challenge. `http://www.image-net.org/challenges/LSVRC`, 2017. Accessed: January 25, 2020.

[56] UNC Vision Lab. Large Scale Visual Recognition Challenge 2016 (ILSVRC2016). `http://image-net.org/challenges/LSVRC/2016/results`, 2016. Accessed: March 15, 2017.

[57] UNC Vision Lab. Large Scale Visual Recognition Challenge 2016 (ILSVRC2016). `http://image-net.org/challenges/LSVRC/2017/results`, 2017. Accessed: January 25, 2020.

[58] Nicholas D. Lane and Petko Georgiev. Can Deep Learning Revolutionize Mobile Sensing? In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, HotMobile '15, pages 117–122. ACM, 2015. ISBN 978-1-4503-3391-7. doi: 10.1145/2699343.2699349. URL `http://doi.acm.org/10.1145/2699343.2699349`.

[59] Nicholas D. Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesiand, Lei Jiao, Lorena Qendro, and Fahim Kawsar. DeepX: a Software Accelerator for Low-Power Deep Learning Inference on Mobile Devices. In *Proceedings of the 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN), Vienna, Austria*, pages 1–12, April 11–14 April 2016. doi: 10.1109/IPSN.2016.7460664.

[60] Nicholas D. Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, and Fahim Kawsar. An Early Resource Characterization of Deep Learning on Wearables, Smartphones and Internet-of-Things Devices. In *Proceedings of the 2015 International Workshop on Internet of Things Towards Applications*, IoT-App '15, pages 7–12. ACM, 2015. ISBN 978-1-4503-3838-7. doi: `10.1145/2820975.2820980`.

[61] Nicholas D. Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, and Fahim Kawsar. Demo: Accelerated Deep Learning Inference for Embedded and Wearable Devices Using DeepX. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services Companion*, MobiSys '16 Companion, pages 109–109. ACM, 2016. ISBN 978-1-4503-4416-6. doi: 10.1145/2938559.2949718. URL `http://doi.acm.org/10.1145/2938559.2949718`.

[62] Michael LeBeane, Shuang Song, Reena Panda, Jee Ho Ryoo, and Lizy K. John. Data Partitioning Strategies for Graph Workloads on Heterogeneous Clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 1–12, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450337236. doi: 10.1145/2807591.2807632. URL `https://doi.org/10.1145/2807591.2807632`.

[63] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proc IEEE*, 86(11):2278–2324, Nov. 1998. ISSN 0018-9219. doi: 10.1109/5.726791.

[64] Charles Eric Leiserson and James Benjamin Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, Jun 1991. ISSN 1432-0541. doi: 10.1007/BF01759032. URL `https://doi.org/10.1007/BF01759032`.

[65] Sam Leroux, Steven Bohez, Tim Verbelen, Bert Vankeirsbilck, Pieter Simoens, and Bart Dhoedt. Resource-constrained classification using a cascade of neural network layers. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7, 2015. doi: 10.1109/IJCNN.2015.7280601.

[66] Sam Leroux, Steven Bohez, Elias De Coninck, Pieter Van Molle, Bert Vankeirsbilck, Tim Verbelen, Pieter Simoens, and Bart Dhoedt. Multi-fidelity deep neural networks for adaptive inference in the internet of multimedia things. *Future Generation Computer Systems*, 97:355 – 360, 2019. ISSN 0167-739X. doi: 10.1016/j.future.2019.03.001. URL `http://www.sciencedirect.com/science/article/pii/\S0167739X17324664`.

[67] He Li, Kaoru Ota, and Mianxiong Dong. Learning IoT in Edge: Deep Learning for the Internet of Things with Edge Computing. *IEEE Network*, 32(1):96–101, Jan 2018. ISSN 0890-8044. doi: 10.1109/MNET.2018.1700202.

[68] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling Distributed Machine Learning with the Parameter Server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 583–598. USENIX Association, 2014. ISBN 978-1-931971-16-4. URL `http://dl.acm.org/citation.cfm?id=2685048.2685095`.

[69] Shancang Li, Li Da Xu, and Shanshan Zhao. The Internet of Things: a survey. *Information Systems Frontiers*, 17(2):243–259, 2015. ISSN 1572-9419. doi: 10.1007/s10796-014-9492-7. URL `http://dx.doi.org/10.1007/s10796-014-9492-7`.

[70] Jie Lin, Wei Yu, Nan Zhang, Xinyu Yang, Hanlin Zhang, and Wei Zhao. A Survey on Internet of Things: Architecture, Enabling Technologies, Security and Privacy, and Applications. *IEEE Internet Things Journal*, 4(5):1125–1142, Oct. 2017. doi: 10.1109/JIOT.2017.2683200.

[71] Efren Lopez-Jimenez, Juan Irving Vasquez-Gomez, Miguel Angel Sanchez-Acevedo, Juan Carlos Herrera-Lozada, and Abril Valeria Uriarte-Arcia. Columnar cactus recognition in aerial images using a deep learning approach. *Ecological Informatics*, 52:131–138, JUL 2019. ISSN 1574-9541. doi: 10.1016/j.ecoinf.2019.05.005.

[72] Muhammad Junaid Malik, Thomas Fahringer, and Radu Prodan. Execution Time Prediction for Grid Infrastructures Based on Runtime Provenance Data. In *Proceedings of the 8th Workshop on Workflows in Support of Large-Scale Science*, WORKS '13, pages 48–57. ACM, 2013. ISBN 978-1-4503-2502-8. doi: 10.1145/2534248.2534253. URL `http://doi.acm.org/10.1145/2534248.2534253`.

[73] Yuyi Mao, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B. Letaief. A Survey on Mobile Edge Computing: The Communication Perspective. *IEEE COMMUNICATIONS SURVEYS AND TUTORIALS*, 19(4):2322–2358, 2017. doi: {10.1109/COMST.2017.2745201}.

[74] Fabíola Martins Campos de Oliveira. Source code for KLP, DN^{2}PCIoT, and MDN^{2}PCIoT. `https://bitbucket.org/FabiolaOliveira/mdn2pciot`, 2020. Accessed: March 23, 2020.

[75] Fabíola Martins Campos de Oliveira. Source code for KLP, DN^{2}PCIoT, and MDN^{2}PCIoT. `https://github.com/lmcad-unicamp/MDN2PCIoT`, 2020. Accessed: March 23, 2020.

[76] Fabíola Martins Campos de Oliveira and Edson Borin. Partitioning Convolutional Neural Networks to Maximize the Inference Rate on Constrained IoT Devices. *Future Internet*, 11(10), 2019. ISSN 1999-5903. doi: 10.3390/fi11100209. URL `https://www.mdpi.com/1999-5903/11/10/209`.

[77] Yasir Mehmood, Farhan Ahmad, Ibrar Yaqoob, Asma Adnane, Muhammad Imran, and Sghaier Guizani. Internet-of-Things-Based Smart Cities: Recent Advances and Challenges. *IEEE Communications Magazine*, 55(9):16–24, Sep. 2017. ISSN 0163-6804. doi: 10.1109/MCOM.2017.1600514.

[78] Massimo Merenda, Carlo Porcaro, and Demetrio Iero. Edge machine learning for ai-enabled iot devices: A review. *Sensors*, 20(9), 2020. ISSN 1424-8220. doi: 10.3390/s20092533. URL `https://www.mdpi.com/1424-8220/20/9/2533`.

[79] Tomáš Mikolov, Anoop Deoras, Daniel Povey, Lukáš Burget, and Jan Černocký. Strategies for training large scale neural network language models. In *2011 IEEE Workshop on Automatic Speech Recognition Understanding*, pages 196–201, Dec 2011. doi: 10.1109/ASRU.2011.6163930.

[80] Mahdi H. Miraz, Maaruf Ali, Peter S. Excell, and Richard Picking. Internet of Nano-Things, Things and Everything: Future Growth Trends. *Future Internet*, 10 (8), 2018. ISSN 1999-5903. doi: 10.3390/fi10080068. URL `https://www.mdpi.com/1999-5903/10/8/68`.

[81] Maryam M. Najafabadi, Flavio Villanustre, Taghi M. Khoshgoftaar, Naeem Seliya, Randall Wald, and Edin Muharemagic. Deep learning applications and challenges in big data analytics. *Journal of Big Data*, 2(1):1, Feb 2015. ISSN 2196-1115. doi: 10.1186/s40537-014-0007-7. URL `https://doi.org/10.1186/s40537-014-0007-7`.

[82] Andrew Ng. Machine Learning. `https://www.coursera.org/learn/machine-learning`, 2020. Accessed: August 17, 2020.

[83] Liu Ningbo, Xu Yanan, Tian Yonghua, Ma Hongwei, and Wen Shuliang. Background classification method based on deep learning for intelligent automotive radar target detection. *Future Generation Computer Systems*, 94:524–535, MAY 2019. ISSN 0167-739X. doi: 10.1016/j.future.2018.11.036.

[84] Rodrigo Frassetto Nogueira, Roberto de Alencar Lotufo, and Rubens Campos Machado. Fingerprint Liveness Detection Using Convolutional Neural Networks. *IEEE Transactions on Information Forensics and Security*, 11(6):1206–1213, June 2016. ISSN 1556-6013. doi: 10.1109/TIFS.2016.2520880.

[85] Beng Chin Ooi, Kian-Lee Tan, Sheng Wang, Wei Wang, Qingchao Cai, Gang Chen, Jinyang Gao, Zhaojing Luo, Anthony K.H. Tung, Yuan Wang, Zhongle Xie, Meihui Zhang, and Kaiping Zheng. SINGA: a Distributed Deep Learning Platform. In *Proceedings of the 23rd ACM International Conference on Multimedia*, MM '15, pages 685–688. ACM, Oct. 2015. ISBN 978-1-4503-3459-4. doi: `10.1145/2733373.2807410`.

[86] Group OpenFog Consortium Architecture Working Group. OpenFog Reference Architecture for Fog Computing. `https://www.iiconsortium.org/pdf/OpenFog_Reference_Architecture_2_09_17.pdf`, 2017. Accessed: July 22, 2019.

[87] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8026–8037. Curran Associates, Inc., 2019. URL `http://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[88] Flávia Pisani, Fabíola Martins Campos de Oliveira, Eduardo de Souza Gama, Roger Immich, Luiz Fernando Bittencourt, and Edson Borin. Fog Computing on Constrained Devices: Paving the Way for the Future IoT. In Fatos Xhafa and Arun Kumar Sangaiah, editors, *Advances in Edge Computing: Massive Parallel Processing and Applications*, volume 35 of *Advances in Parallel Computing*, pages 22–60. IOS Press, Amsterdam, Netherlands, 2020. ISBN 978-1-64368-063-7. URL `http://doi.org/10.3233/APC200003`.

[89] Gopika Premsankar, Mario Di Francesco, and Tarik Taleb. Edge Computing for the Internet of Things: A Case Study. *IEEE Internet of Things Journal*, 5(2): 1275–1284, April 2018. ISSN 2372-2541. doi: 10.1109/JIOT.2018.2805263.

[90] Fatemeh Rahimian, Amir H. Payberah, Sarunas Girdzijauskas, and Seif Haridi. Distributed Vertex-Cut Partitioning. In Kostas Magoutis and Peter Pietzuch, editors, *In Lecture Notes in Computer Science, Proceedings of the Distributed Applications and Interoperable Systems, Berlin, Germany, June 3-5, 2014*, pages 186–200, Berlin, Heidelberg, Germany, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-43352-2.

[91] Matt Reynolds. New computer vision challenge wants to teach robots to see in 3D. `https://www.newscientist.com/article/2127131-new-computer-vision-challenge-wants-to-teach-robots-to-see-in-3d/`, 2017. Accessed: January 25, 2020.

[92] João Antonio Magri Rodrigues, Fabíola Martins Campos de Oliveira, Renata Spolon Lobato, Roberta Spolon, Aleardo Manacero, and Edson Borin. Improving Virtual Machine Consolidation for Heterogeneous Cloud Computing Datacenters. In *2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 176–179, Oct 2019. doi: 10.1109/SBAC-PAD.2019.00037.

[93] Jürgen Schmidhuber. Deep learning in neural networks: An overview . *Neural Networks*, 61:85 – 117, 2015. ISSN 0893-6080. doi: https://doi.org/10.1016/j.neunet.2014.09.003. URL `http://www.sciencedirect.com/science/article/pii/S0893608014002135`.

[94] Jie Shao, Xiaoteng Zhang, Zhengyan Ding, Yixin Zhao, Yanjun Chen, Jianying Zhou, Wenfei Wang, Lin Mei, and Chuanping Hu. Good Practices for Deep Feature Fusion. `http://image-net.org/challenges/talks/2016/Trimps-Soushen@ILSVRC2016.pdf`, 2016. Accessed: March 15, 2017.

[95] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. Mesh-TensorFlow: Deep Learning for Supercomputers. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 10414–10423. Curran Associates, Inc., 2018. URL `http://papers.nips.cc/paper/8242-mesh-tensorflow-deep-learning-for-supercomputers.pdf`.

[96] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[97] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. Technical report, 2014.

[98] SMEVenture. Impending Major Challenges for Smart Cities mission in India 2019. `https://www.smeventure.com/major-challenges-smart-cities-india-2019`, 2019. Accessed: August 16, 2020.

[99] José Angel Carvajal Soto, Marc Jentsch, Davy Preuveneers, and Elisabeth Ilie-Zudor. CEML: Mixing and Moving Complex Event Processing and Machine Learning to the Edge of the Network for IoT Applications. In *Proceedings of the 6th International Conference on the Internet of Things*, IoT'16, pages 103—-110, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450348140. doi: 10.1145/2991561.2991575. URL `https://doi.org/10.1145/2991561.2991575`.

[100] Stanford. CS231n: Convolutional Neural Networks for visual recognition. `http://cs231n.github.io/convolutional-networks/`, 2018. Accessed: May 20, 2018.

[101] STMicroelectronics. STM32L151x6/8/B. `https://www.st.com/resource/en/datasheet/stm32l151vb.pdf`, 2016. Accessed: July 24, 2019.

[102] STMicroelectronics. STM32F469xx. `https://www.st.com/resource/en/datasheet/stm32f469ae.pdf`, 2018. Accessed: July 24, 2019.

[103] STMicroelectronics. STM32L433xx. `https://www.st.com/resource/en/datasheet/stm32l433cc.pdf`, 2018. Accessed: July 24, 2019.

[104] Kai Sun, Shao-Hsuan Huang, David Shan-Hill Wong, and Shi-Shang Jang. Design and Application of a Variable Selection Method for Multilayer Perceptron Neural Network With LASSO. *IEEE Transactions on Neural Networks and Learning Systems*, 28(6):1386–1396, 2017. ISSN 2162-237X. doi: 10.1109/TNNLS.2016.2542866.

[105] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE*, 105 (12):2295–2329, Dec 2017. ISSN 1558-2256. doi: 10.1109/JPROC.2017.2761740.

[106] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragom Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andre Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, June 2015. doi: `10.1109/CVPR.2015.7298594`.

[107] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A. Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, page 4278–4284. AAAI Press, 2017.

[108] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. DeepFace: Closing the Gap to Human-Level Performance in Face Verification. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1701–1708, June 2014. doi: 10.1109/CVPR.2014.220.

[109] Ao Tang, Ke Lu, Yufei Wang, Jie Huang, and Houqiang Li. A Real-Time Hand Posture Recognition System Using Deep Neural Networks. *ACM Transactions on Intelligent Systems and Technology*, 6(2):21:1–21:23, March 2015. ISSN 2157-6904. doi: 10.1145/2735952. URL `http://doi.acm.org/10.1145/2735952`.

[110] Antonis Tzounis, Nikolaos Katsoulas, Thomas Bartzanas, and Constantinos Kittas. Internet of things in agriculture, recent advances and future challenges. *Biosystems Engineering*, 164:31 – 48, 2017. ISSN 1537-5110. doi: https://doi.org/10.1016/j.biosystemseng.2017.09.007. URL `http://www.sciencedirect.com/science/article/pii/S1537511017302544`.

[111] Luis Miguel Vaquero and Luis Rodero-Merino. Finding Your Way in the Fog: Towards a Comprehensive Definition of Fog Computing. *ACM SIGCOMM Computer Communication Review*, 44(5):27–32, 2014. ISSN 0146-4833. doi: 10.1145/2677046.2677052. URL `http://doi.acm.org/10.1145/2677046.2677052`.

[112] Algimantas Venckauskas, Vytautas Stuikys, Robertas Damaševičius, and Nerijus Jusas. Modelling of Internet of Things units for estimating security-energy-performance relationships for quality of service and environment awareness. *Security Communication Networks*, 9(16):3324–3339, 2016. doi: 10.1002/sec.1537.

[113] Tim Verbelen, Tim Stevens, Filip De Turck, and Bart Dhoedt. Graph partitioning algorithms for optimizing software deployment in mobile cloud computing. *Future Generation Computer Systems*, 29(2):451 – 459, 2013. ISSN 0167-739X. doi: 10.1016/j.future.2012.07.003. URL `http://www.sciencedirect.com/science/article/pii/\S0167739X12001513`.

[114] Wei W, Xu Xia, Marcin Wozniak, Xunli Fan, Robertas Damaševičius, and Ye Li. Multi-sink distributed power control algorithm for Cyber-physical-systems in coal mine tunnels. *Computer Networks*, 161:210 – 219, 2019. ISSN 1389-1286. doi: 10.1016/j.comnet.2019.04.017. URL `http://www.sciencedirect.com/science/article/pii/S1389128618310673`.

[115] Ji Wan, Dayong Wang, Steven Chu Hong Hoi, Pengcheng Wu, Jianke Zhu, Yongdong Zhang, and Jintao Li. Deep Learning for Content-Based Image Retrieval:

A Comprehensive Study. In *Proceedings of the 22nd ACM International Conference on Multimedia*, MM '14, pages 157–166. ACM, 2014. ISBN 978-1-4503-3063-3. doi: 10.1145/2647868.2654948. URL http://doi.acm.org/10.1145/2647868.2654948.

[116] Wei Wang, Gang Chen, Anh Tien Tuan Dinh, Jinyang Gao, Beng Chin Ooi, Kian-Lee Tan, and Sheng Wang. SINGA: Putting Deep Learning in the Hands of Multimedia Users. In *Proceedings of the 23rd ACM International Conference on Multimedia*, MM '15, pages 25–34. ACM, 2015. ISBN 978-1-4503-3459-4. doi: 10.1145/2733373.2806232. URL http://doi.acm.org/10.1145/2733373.2806232.

[117] Pijika Watcharapichat, Victoria Lopez Morales, Raul Castro Fernandez, and Peter Pietzuch. Ako: decentralised Deep Learning with Partial Gradient Exchange. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 84–97. ACM, Oct. 2016. ISBN 978-1-4503-4525-5. doi: 10.1145/2987550.2987586.

[118] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Managed Communication and Consistency for Fast Data-parallel Iterative Analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 381–394. ACM, 2015. ISBN 978-1-4503-3651-2. doi: 10.1145/2806777.2806778. URL http://doi.acm.org/10.1145/2806777.2806778.

[119] Xitao Wen, Kai Chen, Yan Chen, Yongqiang Liu, Yong Xia, and Chengchen Hu. VirtualKnotter: online Virtual Machine Shuffling for Congestion Resolving in Virtualized Datacenter. In *Proceedings of the IEEE 32nd International Conference on Distributed Computing Systems Workshop, Macau, China*, pages 12–21, June 18–21 June 2012.

[120] Marilyn Wolf. Chapter 5 - Program Design and Analysis. In Marilyn Wolf, editor, *Computers as Components*. 4th ed. edition. ISBN 978-0-12-805387-4. URL http://www.sciencedirect.com/science/article/pii/\B9780128053874000054. Morgan Kaufmann: Burlington, MA, USA, 2017; pp. 221–319, ISBN 978-0-12-805387-4.

[121] Li Da Xu, Wu He, and Shancang Li. Internet of Things in Industries: a Survey. *IEEE Transactions on Industrial Informatics*, 10(4):2233–2243, Nov. 2014. ISSN 1551-3203. doi: 10.1109/TII.2014.2300753.

[122] Xiaolong Xu, Daoming Li, Zhonghui Dai, Shancang Li, and Xuening Chen. A Heuristic Offloading Method for Deep Learning Edge Services in 5G Networks. *IEEE Access*, 7:67734–67744, 2019. ISSN 2169-3536. doi: 10.1109/ACCESS.2019.2918585.

[123] Xiaolong Xu, Qingxiang Liu, Yun Luo, Kai Peng, Xuyun Zhang, Shunmei Meng, and Lianyong Qi. A computation offloading method over big data for IoT-enabled cloud-edge computing. *Future Generation Computer Systems*, 95:522 – 533, 2019. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2018.12.055. URL http://www.sciencedirect.com/science/article/pii/S0167739X18319770.

[124] Feng Yan, Olatunji Ruwase, Yuxiong He, and Trishul Chilimbi. Performance Modeling and Scalability Optimization of Distributed Deep Learning Systems. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery*

and Data Mining, KDD '15, pages 1355–1364. ACM, August 2015. ISBN 978-1-4503-3664-2. doi: 10.1145/2783258.2783270.

[125] Shuochao Yao, Yiran Zhao, Aston Zhang, Lu Su, and Tarek Abdelzaher. DeepIoT: Compressing Deep Neural Network Structures for Sensing Systems with a Compressor-Critic Framework. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SenSys '17), Delft, Netherlands*, pages 1–14, 5–8 November 2017; 4. ISBN 978-1-4503-5459-2. URL http://doi.acm.org/10.1145/3131672.3131675.

[126] Dong Yu, Adam Eversole, Mike Seltzer, Kaisheng Yao, Oleksii Kuchaiev, Yu Zhang, Frank Seide, Zhiheng Huang, Brian Guenter, Huaming Wang, Jasha Droppo, Geoffrey Zweig, Chris Rossbach, Jie Gao, Andreas Stolcke, Jon Currey, Malcolm Slaney, Guoguo Chen, Amit Agarwal, Chris Basoglu, Marko Padmilac, Alexey Kamenev, Vladimir Ivanov, Scott Cypher, Hari Parthasarathi, Bhaskar Mitra, Baolin Peng, and Xuedong Huang. An Introduction to Computational Networks and the Computational Network Toolkit. Technical report, 2014. URL https://www.microsoft.com/en-us/research/publication/an-introduction-to-computational-networks-and-the-computational-network-toolkit/.

[127] Wei Yu, Fan Liang, Xiaofei He, William Grant Hatcher, Chao Lu, Jie Lin, and Xinyu Yang. A Survey on the Edge Computing for the Internet of Things. *IEEE Access*, 6:6900–6919, 2018. ISSN 2169-3536. doi: 10.1109/ACCESS.2017.2778504.

[128] Sergey Zagoruyko and Nikos Komodakis. Wide Residual Networks. In Edwin R. Hancock Richard C. Wilson and William A. P. Smith, editors, *Proceedings of the British Machine Vision Conference (BMVC)*, pages 87.1–87.12. BMVA Press, September 2016. ISBN 1-901725-59-6. doi: 10.5244/C.30.87. URL https://dx.doi.org/10.5244/C.30.87.

[129] Matthew D. Zeiler and Rob Fergus. *Visualizing and Understanding Convolutional Networks*, pages 818–833. Springer International Publishing, 2014. ISBN 978-3-319-10590-1. doi: 10.1007/978-3-319-10590-1_53. URL http://dx.doi.org/10.1007/978-3-319-10590-1_53.

[130] Hongwei Zhao, Weishan Zhang, Haoyun Sun, and Bing Xue. Embedded Deep Learning for Ship Detection and Recognition. *Future Internet*, 11(2), 2019. ISSN 1999-5903. doi: 10.3390/fi11020053. URL https://www.mdpi.com/1999-5903/11/2/53.

[131] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. DeepThings: Distributed Adaptive Deep Learning Inference on Resource-Constrained IoT Edge Clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2348–2359, Nov 2018. ISSN 0278-0070. doi: 10.1109/TCAD.2018.2858384.

[132] Ivan Zyrianoff, Alexandre Heideker, Dener Silva, João Kleinschmidt, Juha-Pekka Soininen, Tullio Salmon Cinotti, and Carlos Kamienski. Architecting and Deploying IoT Smart Applications: A Performance–Oriented Approach. *Sensors*, 20(1), 2020. ISSN 1424-8220. doi: 10.3390/s20010084. URL https://www.mdpi.com/1424-8220/20/1/84.

Appendix A

METIS parameters

In this appendix, we report the METIS parameters that led to the METIS results in each experiment. We show these values for reproducibility. Table A.1 shows these parameters for the METIS inference rate maximization experiments in Table 5.3 and Table A.2 shows these parameters for the METIS communication reduction experiments in Figure 5.5. *Cut* stands for the *edgecut* minimization objective function and *vol* stands for the total communication volume minimization objective function. Table A.3 shows the METIS parameters that led to the respective results in Figure 6.5 and Table A.4 shows the METIS parameters that led to the respective results in Figure 6.6.

Table A.1: METIS parameters used to achieve the METIS results in Table 5.3.

Number of devices allowed to be used in the experiments	**Number of partitionings *(ncuts)***	**Number of iterations in the refinements *(niter)***	**Objective function *(objtype)***	**Maximum allowed imbalance among the partitions for the memory *(ubvec[0])***	**Maximum allowed imbalance among the partitions for the computation *(ubvec[1])***
free input LeNet 2:1					
2	16	5	vol	1.01	1.1
4	16	1	cut	1.025	1.01
11	64	5	cut	1.01	1.5
56	N/a*	N/a	N/a	N/a	
63	N/a	N/a	N/a	N/a	
locked input LeNet 2:1					
2	2	5	vol	1.25	2.0
4	8	5	cut	1.01	1.5
11	4	1	cut	1.01	2.0
56	N/a	N/a	N/a	N/a	
63	N/a	N/a	N/a	N/a	
free input LeNet 1:1					
2	8	10	vol	1.025	1.1
4	32	1	vol	1.015	1.1
11	128	50	cut	1.015	1.1
56	N/a	N/a	N/a	N/a	
63	N/a	N/a	N/a	N/a	
locked input LeNet 1:1					
2	128	1	vol	1.25	1.5
4	64	5	vol	1.025	1.5
11	32	5	cut	1.025	2.0
56	N/a	N/a	N/a	N/a	
63	N/a	N/a	N/a	N/a	

* Not applicable.

Table A.2: METIS parameters used to achieve the METIS results in Figure 5.5.

Number of devices allowed to be used in the experiments	**Number of partitionings *(ncuts)***	**Number of iterations in the refinements *(niter)***	**Objective function *(objtype)***	**Maximum allowed imbalance among the partitions for the memory *(ufactor)***
free input LeNet 2:1				
2	1	1	cut	15
4	1	1	cut	25
11	16	1	cut	100
56	N/a*	N/a	N/a	N/a
63	N/a	N/a	N/a	N/a
locked input LeNet 2:1				
2	1	1	vol	50
4	64	10	vol	50
11	2	5	vol	1
56	N/a	N/a	N/a	N/a
63	N/a	N/a	N/a	N/a
free input LeNet 1:1				
2	8	10	vol	500
4	32	1	vol	25
11	128	50	cut	20
56	N/a	N/a	N/a	N/a
63	N/a	N/a	N/a	N/a
locked input LeNet 1:1				
2	1	5	cut	20
4	4	5	vol	100
11	16	10	cut	250
56	N/a	N/a	N/a	N/a
63	N/a	N/a	N/a	N/a

* Not applicable.

Table A.3: METIS parameters used to achieve the METIS results in Figure 6.5.

Number of devices allowed to be used in the experiments	**Number of partitionings *(ncuts)***	**Number of iterations in the refinements *(niter)***	**Objective function *(objtype)***	**Maximum allowed imbalance among the partitions for the memory *(ufactor)***
2	4/8/16/32	5/10/25/50/100	cut	20
4	8	10	cut	67
8	4/8	10/25	vol	15/20/67/125/250/500
16	16	25	cut	67
40	N/a*	N/a	N/a	N/a

* Not applicable.

Table A.4: METIS parameters used to achieve the METIS results in Figure 6.6.

Number of devices allowed to be used in the experiments	**Number of partitionings *(ncuts)***	**Number of iterations in the refinements *(niter)***	**Objective function *(objtype)***	**Maximum allowed imbalance among the partitions for the memory *(ubvec[0])***	**Maximum allowed imbalance among the partitions for the computation *(ubvec[1])***
2	32/64/128	5	cut	1.15	1.005
4	16	5	vol	1.20	1.001
8	16	7	cut	1.17	1.001
16	32	10	cut	1.17	1.001
40	N/a*	N/a	N/a	N/a	N/a

* Not applicable.