Deep Learning Process Integration on Heterogeneous GPU/FPGA Embedded Platforms

Walther Carballo Hernandez

▶ **To cite this version:**

Walther Carballo Hernandez. Deep Learning Process Integration on Heterogeneous GPU/FPGA Embedded Platforms. Embedded Systems. Université Clermont Auvergne, 2022. English. NNT : 2022UCFAC087 . tel-04109891

HAL Id: tel-04109891

https://theses.hal.science/tel-04109891v1

Submitted on 30 May 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

DOCTORAL THESIS

Deep Learning Process Integration on Heterogeneous GPU/FPGA Embedded Platforms

---

Author:
Walther CARBALLO-HERNÁNDEZ

Supervisor:
Prof. François BERRY
Prof. Maxime PELCAT

Reviewers:
Prof. Philippe COUSSY
Prof. Alfredo GARDEL VICENTE

Examiners:
Prof. Miguel Octavio ARIAS ESTRADA
Prof. Ricardo CARMONA GALÁN
Prof. Patrick GIRARD
Prof. Paula LÓPEZ MARTÍNEZ

Defended in public on

November 18, 2022

A thesis submitted in fulfillment of the requirements for the degree of Docteur de l'Université Clermont Auvergne

at the

DREAM - Image, Systèmes de Percéption et Robotique (ISPR)
Institut Pascal

Numéro National de Thèse (NNT) : 2022UCFAXXXX

UNIVERSITÉ CLERMONT AUVERGNE
Ecole Doctorale des Sciences Pour l'Ingénieur
Institut Pascal

Abstract

Deep Learning Process Integration on Heterogeneous GPU/FPGA Embedded Platforms

by Walther CARBALLO-HERNÁNDEZ

Deep Learning (DL) algorithm deployment on *edge devices*, such as Convolutional Neural Network (CNN) inference, has established a high computing demand on devices with limited resources, requiring low execution time and reduced energy consumption. To meet the requirements with such constraints, hardware systems have adopted unconventional processors co-located on the same platform. This architectural *heterogeneity* introduces many challenges in how these processors interact. A well-defined software-hardware co-design environment must be carefully built to ensure a high-performance solution. For this purpose, heterogeneous hardware-awareness must be integrated in the design workflow.

To avoid hardware-agnostic low performance programming, state-of-the-art literature incorporates performance profiling models to aid with partition selection on each accelerator. Subsequently, mathematical optimization techniques benefit from these models to improve workload distribution, specifically tailored for the platform.

This thesis aims to assist the designer by studying the *modeling*, *partitioning* and *optimization* of embedded heterogeneous platforms in the context of CNN computation models. The scope of this dissertation mainly covers the coupling topologies between Graphics Processing Unit (GPU) and Field Programmable Gate Array (FPGA) accelerators in hybrid systems. The opportunities and limitations of hybrid programmable logic and Single-Instruction Multiple-Data (SIMD) architectures are analyzed and discussed.

Keywords: Heterogeneous computing, Embedded design, CNN, GPU, FPGA

UNIVERSITÉ CLERMONT AUVERGNE
Ecole Doctorale des Sciences Pour l'Ingénieur
Institut Pascal

Résumé

Intégration des Processus d'Apprentissage Profond sur des Plateformes Hétérogènes Embarquées GPU/FPGA

par Walther CARBALLO-HERNÁNDEZ

Le déploiement d'algorithmes tel que l'inférence de réseaux de neurones convolutifs, impose des temps d'exécution courts et une consommation électrique maîtrisée sur des systèmes embarqués disposant de ressources de calculs limitées. De nouvelles électroniques spécifiques composées de différentes architectures matérielles ont émergé pour répondre à ce type de demande. Ce type d'architecture *hétérogène* impose de nouveaux défis notamment au niveau de l'interaction entre les différentes composantes. Il est alors nécessaire de passer par une étape de co-design matérielle/logicielle décrivant au mieux la plateforme électronique afin d'atteindre un optimum en termes de consommation, latence et vitesse d'exécution.

Pour éviter une conception agnostique du matériel, la littérature de l'état de l'art incorpore des modèles en profilant la performance. Celle-ci, dans l'intention d'aider avec la sélection des partitions sur chaque accélérateur. En suite, des techniques d'optimisation mathématique profitent de ces résultats pour améliorer encore la distribution des tâches de travail spécifiquement adaptées à la plateforme.

Cette thèse cible d'assister le développeur en étudiant le *modelage*, *partitionnement* et *optimisation* pour les plateformes embarquées hétérogènes dans le contexte des modèles de calcul des CNNs. Le cadre de ce manuscrit couvre principalement des topologies en couplant des accélérateurs GPU et FPGA dans des systèmes hybrides. En se concentrant de cette manière, sur les opportunités et limitations de l'intégration d'architecturelle : logique programmable customisable et SIMD.

Mot clés: Calcul hétérogène, Conception embarquée, Réseau de neurones convolutif, GPU, FPGA

To my parents, my sisters, my brother and all my family

Acknowledgements

I would like to express my deepest gratitude to my supervisors: Prof. François Berry and Prof. Maxime Pelcat. Without your guidance and expertise this endeavor would not have been possible. Additionally, I will always be extremely grateful for the funding from the European Commission's Marie Skłodowska-Curie Actions and for this international opportunity. My appreciation also goes out to Prof. Ricardo Carmona Galán and the colleagues from the ACHIEVE ITN 2020 consortium who professionally shaped my career.

My recognition extends to the committee members for actively participating in my project and proof-reading my dissertation: to my reviewers Prof. Philippe Coussy and Prof. Alfredo Gardel Vicente and my examiners Prof. Patrick Girard and Prof. Paula López Martínez. I am also indebted with Prof. Miguel Octavio Arias Estrada for his encouragement and motivation in the early steps of my doctoral thesis, being such a model of excellence in research and an integral human being. Thanks also go out to Prof. Shuvra Shikhar Bhattacharyya for your collaboration on the first chapter and academic article of this manuscript.

I would also like to thank my lab friends and work colleagues from DREAM Team and the Institut Pascal: Kamel, Ny Ando, Lobna, Juan, Abiel, Seyfedinne, Anas and Ivan Luca. Special thanks to PhD. Luca Maggiani and PhD. Federico Civerchia from Sma-Rty for leading and helping me during my secondment in Italy. I would like to particularly thank PhD. Jonathan Bonnard, without your help the design of the smart camera (*X*-MERA) would have simply not seen the light of existence. Thanks for your invaluable encouragement and wise advice. Thank you all for being there and tolerating me in *that* difficult period of time during the development of my thesis.

Most importantly, my family deserves my endless gratitude. My mother, whose eyes closed one last time before the beginning of this project, but who has always being as present as the day I opened mine for the very first time. My father, who has always been my biggest supporter, constantly encouraging me to give the best of me through the wisdom of his words and his advice. To my sisters, Pamela and Yamile, for affectively reminding me how amazing I am, specially when my self doubts clouded my judgement. My brother Carlos, who backed me up on each decision in my life and being there during difficult times. To the friends that I have met on the way around the world these last years. For all these wholesome moments of success with my family and friends, from the deepest and truthful part of my heart, thank you.

Contents

1 Introduction **1**
1.1 Heterogeneous computing 1
1.2 Challenges and problem definition 4
1.3 Contributions 7
1.4 Manuscript outline 7
1.5 Publications 8
1.6 Submissions 9

2 Flydeling: Performance Models for Hardware Acceleration of CNNs through SI **11**
2.1 Chapter abstract 11
2.2 Introduction 12
2.3 Related works 13
2.4 Problem definition: Applying Flydeling to CNN KPIs 17
2.4.1 Flydel definition and properties 17
2.4.2 Using CNN properties as application activity 19
2.4.3 Device Key Performance Indicators (*KPI*) models 20
2.4.4 Automated model selection with the competitive ensemble modeling 22
2.4.5 Heterogeneous model definition 25
2.5 Flydeling: randomly-excited system identification applied to KPI estimation 26
2.5.1 System identification 29
2.5.2 Stochastic sequence excitation 30
2.5.3 Linear parametric SI 33
2.5.4 Non-linear parametric SI 34
2.5.5 Model validation 34
2.6 Flydeling evaluation and results 35
2.6.1 Dataset generation 35
2.6.2 Resulting experimental models and evaluation 43
2.7 Conclusions 49

3 Heterogeneous Partitioning Techniques **51**
3.1 Chapter abstract 51
3.2 Introduction 51
3.3 CNN partitioning on GPU-FPGA platforms 53
3.4 Related works 58
3.5 Problem definition: Heterogeneous partitions 61

3.5.1 Mobile CNN modules: partitioning and scheduling 61
3.5.2 DHM for FPGA synthesis definition 64
3.5.3 Inter-device communication modeling 65
3.5.4 CUDA microarchitecture comparison of current Nvidia embedded GPUs 66
3.6 Partition experimental methodology, evaluation and results 67
3.6.1 Measurement-based performance metrics comparison 68
3.6.2 Heterogeneous partitioning results 72
3.7 Conclusions 77

4 CNN Model Partitioning Optimization **81**
4.1 Chapter abstract 81
4.2 Introduction 82
4.3 Related Works 82
4.4 Flydels as monomials and posynomials formulations 84
4.5 Optimization Problem Formulation 85
4.5.1 GGP Formulation of the Heterogeneous CNN Layer Partitioning . 85
4.5.2 GGP Penalization by Equality Constraints Condensation 86
4.6 Experimental results 87
4.6.1 Single layer optimization 88
4.6.2 Full CNN model optimization 91
4.7 Conclusions 99

5 Conclusions and discussion **101**

A Convexity, monomials and posynomials **103**
A.1 Convexity definition 103
A.2 Convexity properties 103
A.2.1 Summation 103
A.2.2 Scaling 104
A.3 Posynomial properties 104
A.3.1 Summation 105
A.3.2 Product 106

B Heterogeneous Smart Camera Architecture Conception **107**
B.1 X-MERA: Co-processor Heterogeneity Integration for smart caMERAs . . 107
B.1.1 Middle Board (MDB) and Bottom Board (BTB) 108
B.1.2 Power measurements and PCIe inter-device communication 111
B.2 Chimera library 114

List of Figures

1.1 Generic heterogeneous architecture [Zah17]. 2
1.2 Multicore types according to [CMHM10]. 3
1.3 Some embedded heterogeneous platforms. 4
1.4 KPI modeling of a heterogeneous platform as system. 5
1.5 CNN model partitioning and mapping into an embedded heterogeneous platform with multiple accelerators. 6
1.6 Thesis content development flow. 7

2.1 Flydel models. 18
2.2 Competitive ensemble of single-variable models as weak regressors combined to build a multi-variable strong regressor for a performance model. 23
2.3 Flydeling methodology. From left to right: Stochastic sequence excitation at data- and structure-level, KPI measurement observations and dataset generation, ensemble modeling for flydel creation and K-fold Cross-Validation. 28
2.4 General SI block diagram. M is the system to be identified by model $\hat{M}$. Values of IFMs, weights (ω) and biases (b) are randomized at data-level. The CNN activity features ($x[k]$) are randomized at structural-level. 29
2.5 Stochastic excitation abstraction levels. 32
2.6 Architecture model as for the heterogeneous setup with three computing elements: A Nvidia® Jetson TX2® CPU-GPU (green) SoC and an Intel® Cyclone10GX FPGA (blue) interconnected through x4 lanes of a PCIe Gen2 link (gray). Each processing element and communication node has an IFM and a OFM associated to it required for metric performance measuring on each device. These measurements are used for the flydel data generation and estimation. 37
2.7 Average generated subsampled dataset $\mathcal{D}$ on each embedded CPU (top row), GPU (middle row) and FPGA (bottom row) platform by keeping constant $|\mathcal{X}| - 2$ features for 3D visualization. Each Subfigures row shows latency and energy KPIs on a single device. The contour curves represent the relation between a given KPI and a single feature variable. On Subfigures 2.7a, 2.7e, 2.7c and 2.7g; features $k = 11$ and $N = 512$ are kept constant. On Subfigures 2.7b, 2.7f, 2.7d and 2.7h; features $H_I = 100$, $W_I = 100$ and $C = 512$ are kept constant. On Subfigures 2.7i and 2.7k, features $k = 5$ and $N = 10$ are constants. Finally, in Subfigures 2.7j and 2.7l, $H_I = 12$, $W_I = 12$ and $C = 10$ are kept constant. 38

2.8 Example of energy consumption measurements (in mJ) over multiple experiment iterations on the ARM® CPU-cores (solid red), the Nvidia® Pascal GPU architecture (solid green) of the Jetson TX2® SoC and the Intel® Cyclone 10GX® (solid blue). Common convolution filter sizes are iterated 1000 times on a fixed size input tensor ($32 \times 32 \times 3$ CIFAR-like input image for this example [Kri09]) with a random uniform distribution for the input and a normal distribution for the kernel values. Number of filters N increases progressively over time and the resulting observations are averaged. . . . 41
2.9 Obtained parameter distribution using 50 iterations of 10-fold cross-validation. 45
2.10 Weight value distributions of the first four convolutional layers of VGG16 (2.10a-2.10d). ResNet18 number of zeros per layer 2.10e. 47
2.11 Comparison of pretrained CNN models energy performance with ImageNet against flydel estimation. On each KPI estimation, the percentage error is shown over each bar. 48

3.1 Tiling on a heterogeneous platform with a single GPU and a single FPGA with configuration: $H_I = H_G + H_F$, $W_I = W_F = W_G$ and $C_I = C_G = C_F$. The blue tensor is the IFM partition mapped on the FPGA. While the green tensor is the IFM partition mapped on the GPU. 54
3.2 Heterogeneous Depth-wise convolution example where the $k \times k$ convolution per input channel is executed on the GPU and the Conv1×1 convolution is done on the FPGA. The partition in blue represents the data and task workload on FPGA, while in green the partition on the GPU. 54
3.3 Heterogeneous Grouped convolution example where the C_I input channels and kernel filters are contiguously and heterogeneously divided on each device. Both resulting partitions are finally concatenated. The partition in blue represents the data and task workload on FPGA, while in green the partition on the GPU. 55
3.4 Heterogeneous Fused layer example where a couple or intermediate layers activity are stored in the internal FPGA RAM memory. Afterwards the output tensor is transferred to the GPU for deeper layers processing. The partition in blue represents the data and task workload on FPGA, while in green the partition on the GPU. 56
3.5 SqueezeNet's Fire module (a) graph representation and (b) scheduling. The nodes from the graph in blue are scheduled on the FPGA while the ones in green are scheduled on the GPU. 62
3.6 MobileNetv2's Bottleneck module with no spatial reduction (a) graph representation and (b) scheduling and with spatial reduction (c) graph representation and (d) scheduling. The nodes from the graph in blue are scheduled on the FPGA while the ones in green are scheduled on the GPU. 63

3.7 ShuffleNetv2's Stage module with no spatial reduction (a) graph representation and (b) scheduling and with spatial reduction (c) graph representation and (d) scheduling. The nodes from the graph in blue are scheduled on the FPGA while the ones in green are scheduled on the GPU. 64
3.8 (a) Latency and (b) Throughput in communication using PCIe Gen2 with x4 lanes over multiple transfer sizes for read, write and simultaneous between the Cyclone10GX FPGA and TegraTX2 GPU. 66
3.9 Performance comparison between the Nvidia® Jetson TX2 with Pascal microarchitecture against Nvidia® Jetson Xavier NX with Volta microarchitecture over many CNNs, as discussed in [HSKR21]. 68
3.10 Latency comparison between multiple convolution function sizes on Cyclone10GX FPGA (blue) and Jetson TX2 GPU (green) for different CNN layers on an input image of $224 \times 224 \times 3$. Blue bars represent the layers implemented on the FPGA and the green bars represent the latency on the GPU. Notice that the missing blue bars represent unfeasible tasks to be mapped on more computational intensive task given the logic or memory element constrains for the FPGA device. 69
3.11 Power dissipation measurement over multiple experiment iterations on the Nvidia Pascal GPU architecture of the Jetson TX2 SoC. Common convolution filter sizes are iterated 5000 times on a fixed size input tensor ($224 \times 224 \times N$ for this example) with a random uniform distribution for the input and the kernel values. The number of filters N increases progressively over time and the resulting observations are averaged. 70
3.12 Average power dissipation comparison between multiple convolution function sizes on Cyclone10GX FPGA (blue) and Tegra TX2 GPU (green) for different CNN layers on an input image of $224 \times 224 \times 3$. Blue bars represent the layers implemented on the FPGA and the green bars represent the power dissipation on the GPU. Notice that the missing blue bars represent unfeasible tasks to be mapped on more computational intensive task given the logic or memory element constrains for the FPGA device. 71
3.13 Energy comparison between multiple convolution function sizes on Cyclone10GX FPGA (blue) and Jetson TX2 GPU (green) for different CNN layers on an input image of $224 \times 224 \times 3$. Blue bars represent the layers implemented on the FPGA and the green bars represent the energy consumption on the GPU. The performance factor in this measure is increased result of multiplication on both power and latency metrics. 72
3.14 Throughput comparison between multiple convolution function sizes on Cyclone10GX FPGA (blue) and Jetson TX2 GPU (green) for different CNN layers on an input image of $224 \times 224 \times 3$. Blue bars represent the layers implemented on the FPGA and the green bars represent the latency on the GPU. The red dashed line represents the maximal theoretical throughput for the PCIe communication lanes. 73

3.15 Average metric performance space of the tested SqueezeNet's modules with different workloads on an homogeneous GPU-only platform (green) and the FPGA-GPU heterogeneous platform (blue). x-axis represents the average energy and y-axis the average latency. 74
3.16 Average metric performance space of the measured MobileNetv2's modules with 0.5x parameters for different workloads on an homogeneous GPU-only platform (green) and the FPGA-GPU heterogeneous platform (blue). x-axis represents the average energy and y-axis the average latency. A zoomed subfigure highlights the performance of the first four modules in a more detailed way. 75
3.17 Average metric performance space of the measured ShuffleNetv2's modules with 0.5× parameters for different workloads on an homogeneous GPU-only platform (green) and the FPGA-GPU heterogeneous platform (blue). x-axis represents the average energy and y-axis the average latency. 76

4.1 **Setup 1:** Single layer setup of a single CNN layer allocation. 88
4.2 First iterations of a relaxed GGP sequential grouped convolution partitioning of an input tensor with 16 channels ($C = 16$) with an increasing α. The problem is solved as a set of GP problems and he tightening only takes a few iterations (iteration 17 with $\alpha = 170$) to find an acceptable solution. Each step is in polynomial time and total optimisation lasts less than a couple of hundred of milliseconds. 89
4.3 Heterogeneous objective function per iteration (without penalization term) from problem in Equation 4.7. 90
4.4 Relaxed GGP sequential grouped convolution partitioning of an input tensor with 16 channels ($C = 16$) with an increasing α over a CPU-GPU-FPGA network. 91
4.5 **Setup 2:** Multi-layer setup of a full CNN model with several layers allocated. The heterogeneous architecture is theoretically simulated. 92
4.6 Communication weight function $S(x) = x^{2k}$ for different values of k. . . . 94
4.7 Approximation error function based on $L2$-Norm 95
4.8 Resulting GConv channel-wise optimized partitions for AlexNet 4.8a, VGG16 4.8b and ResNet 4.8c. 98

B.1 *X*-MERA. 108
B.2 *X*-MERA power supplies, communication interfaces and heterogeneous processors synoptic scheme. 110
B.3 Communication interfaces pinout. 112
B.4 FPGA as NPU accelerator using *Delirium*. 115
B.5 General workflow from high-level programming tools to low-level on-device deployment of CNN models with *Delirium* and "*chimera_lib*". . . . 116

List of Tables

2.1 Modeling taxonomy according to Pimentel et al. [Pim17] and García-Martin et al. [GMRRG19] 17
2.2 Flydel curvature, monotonicity and posynomial properties. 19
2.3 Embedded heterogeneous platform hardware specifications. 37
2.4 Selected single-feature models per KPI and per device using RMSE as loss function with L2 regularization term. $\#\theta^*$ is the number of total parameters chosen for the strong regressor KPI. 44
2.5 Modeling state-of-the-art comparison. 50

3.1 Heterogeneous partitioning methods characteristics. 57
3.2 Result comparison of GPU homogeneous vs FPGA-GPU heterogeneous design on ImageNet-like dataset format for several module architectures with different parameters configurations. 78
3.3 Energy and latency comparison with state-of-the-art partitioning techniques on heterogeneous FPGA-GPU against homogeneous implementations. * In [OHY+18] values were estimated for a single FPGA-GPU pair. 79

4.1 Optimization state-of-the-art comparison. 100

B.1 *X*-MERA Intel® Cyclone® 10 GX FPGA pinout table. 113

Glossary

ADC Digital-Analog Converter. 38, 69

ALM Adaptive Logic Module. 21, 42, 44, 86

ALUT Adaptive Look-Up Table. 21, 42, 44, 86

ANN Artificial Neural Network. 15, 17, 19

ASIC Application Specific Integrated Circuit. 4, 11, 15, 17, 31, 58–60, 81

BN Batch Normalization. 46, 48

BNN Binary Neural Network. 59

BRAM Block RAM. 21, 50

BSP Bulk Synchronous Programming. 59

BTB BoTtom Board. 108, 109

BW Bandwidth. 26, 50

CCR Computation-to-Communication Ratio. 26, 66, 94, 96, 97, 99, 102

CNN Convolutional Neural Network. iii, v, xiii–xvi, 1, 4–8, 11–13, 15–22, 27, 29–33, 35, 37, 39, 42–44, 46, 48–53, 55, 58–61, 63, 65, 67–77, 79, 81–88, 91–93, 95–97, 99, 101, 102, 104, 107, 114–116

CPU Central Processing Unit. xiii, xiv, xvi, 1–3, 7, 11, 12, 14, 15, 17, 30, 35–39, 41, 43, 44, 48–50, 53, 58–61, 69, 77, 81, 84, 88, 90, 91, 96, 97, 99–102, 109, 111, 114

CUDA Compute Unified Device Architecture. 15, 30, 59, 67, 107, 114

DHM Direct Hardware Mapping. 35, 39, 42, 43, 46, 48, 51, 52, 61, 65, 67, 68, 70, 73, 77, 84, 92, 101, 102, 115

DL Deep Learning. iii, 1, 4, 11, 12, 51–53, 61, 65, 67, 77, 82, 101, 107, 115

DMA Direct Memory Access. 114

DMU Decision Making Unit. 60

DNN Deep Neural Networks. 83

DP Dynamic Programming. 97, 100

DPU Deep learning Processor Unit. 16, 50

DRAM Dynamic Random Access Memory. 16, 50

DSE Design Space Exploration. 4, 7, 13–15, 49, 60, 81, 82, 99, 107

DSP Digital Signal Processor. 3, 4, 58, 65

DT Decision Tree. 16, 17, 50

DVFS Dynamic Voltage-Frequency Scaling. 60

FFT Fast-Fourier Transform. 14, 50

FM Feature Maps. 5, 30, 51, 56, 65, 75, 76, 88, 91, 93, 114

FPGA Field Programmable Gate Array. iii, v, xiii–xvii, 1, 3, 4, 7, 11, 12, 15–17, 21, 30, 31, 35–39, 42–44, 46, 48–56, 58–65, 67–79, 81, 83, 84, 86, 88–92, 96, 97, 99–102, 107–109, 111, 113–115

FPS Frames per Second. 42

GEMM GEneral Matrix-Multiplication. 14, 50, 59

GGP Generalized Geometric Programming. xvi, 8, 81, 82, 84, 86, 89–97, 99–101

GP Geometric Programming. xvi, 8, 81, 82, 84–90, 99–101, 105

GPU Graphics Processing Unit. iii, v, xiii–xvii, 1, 3, 4, 7, 11, 12, 15–17, 30, 35–39, 41–45, 48–56, 58, 59, 61–65, 67–79, 81, 83, 84, 86, 88–91, 97, 99–102, 107, 109, 111, 114

HDL Hardware Description Language. 115

HLS High-Level Synthesis. 15

HOG Histogram Oriented Gradient. 59

HPC High Performance Computing. 4, 52, 53, 59

IC Integrated Circuit. 38, 69

IFM Input Feature Map. xiii, xiv, 5, 16, 19–21, 24–27, 29, 31–33, 37, 39, 42, 46, 48, 53–55, 59, 65, 69, 70, 72, 73, 75, 76, 79, 88, 89, 93, 114, 115

ILP Integer Linear Programming. 15, 17, 50, 60, 83, 93, 94, 97, 100

IoT Internet of Things. 51, 83, 100

KPI Key Performance Indicator. xiii, xiv, xvii, 5–7, 11, 14–18, 20–22, 24, 25, 28, 29, 31–33, 35–38, 40, 42–45, 48–50, 68, 72, 74, 84, 101

LAB Logic Array Block. 21, 42, 44, 86

LFSR Linear Feedback Shift-Register. 31, 39

LMA Levenberg-Marquardt Algorithm. 22, 24, 34, 36, 42

LP Linear Programming. 15, 17, 50, 97, 100

LSE Least Squares Error. 12, 30, 32, 34, 50

LSM Least Squares Minimization. 34

LUT Look-Up Table. 59, 65

MAC Multiply and ACcumulate. 15–17, 30, 50, 64, 99

MDB MiDdle Board. 108, 109

MISO Multiple Input Single Output. 34

ML Machine Learning. 15, 34, 35

MoA Model of Architecture. 32

MoC Module-on-Chip. 37, 69

MVTU Matrix-Vector-Threshold Unit. 59

NAS Neural Architecture Search. 60, 102

NCS2 Neural Compute Stick 2. 4

NNLS Non-Negative Least Squares. 15, 50

NPU Neural Processing Unit. xvi, 115

NRMSE Normalized RMSE. 43–45

OFM Output Feature Map. xiii, 5, 19–21, 25, 32, 37, 55, 62, 65, 72, 93, 114, 115

OLS Ordinary Least Squares. 15, 50

PCIe Peripheral Component Interconnect Express. xiii, xv, 35, 37, 59, 61, 65, 69, 72–74, 108, 109, 111, 114

PDF Probability Density Function. 35

PMC Performance Monitoring Counter. 14, 15, 17, 50

PRBS Pseudo-Random Binary Sequence. 31

RF Random Forest. 15–17, 50

RFE Response From Excitation. 30

RL Reinforcement Learning. 102

RMSE Root Mean Square Error. xvii, 23, 24, 43, 44

RTL Register Transfer Level. 58

SDF Synchronous Data Flow. 60

SI System Identification. xiii, 8, 11–13, 16, 17, 22, 24, 26, 27, 29–31, 34, 35, 37, 46, 49, 101

SIMD Single-Instruction Multiple-Data. iii, v, 4, 39, 52, 64, 65, 76, 101

SISO Single Input Single Output. 17, 33

SoC System-on-Chip. xiii–xv, 36, 37, 41, 69, 70, 77, 84, 102, 111

SoM System-on-Module. 108, 111

SVM Support Vector Machine. 15, 17, 34, 59

TPU Tensor Processing Unit. 4, 58, 102

U-Core Unconventional Core. 2–4

VHDL Very High Speed Integrated Circuit (VHSIC) Hardware Description Language. 115

VPU Vision Processing Unit. 58

Chapter 1

Introduction

This first Chapter introduces the main context and ideas of the presented thesis. First, the concept of heterogeneous computing is discussed in the domain of embedded design and the motivation behind it. A plethora of hardware architectures arise from this definition of heterogeneous platforms. A similar phenomenon can be inferred from the software implementation of Deep Learning (DL) models. Thus, this thesis focuses on Central Processing Unit (CPU)-Graphics Processing Unit (GPU)-Field Programmable Gate Array (FPGA) as emerging embedded technologies for DL applications, specifically for Convolutional Neural Network (CNN) models. As a second stage, the challenges evoked from the embedded architecture and algorithmic integration are the central topic of this manuscript. These challenges delimit the scope of this dissertation and establish the problem definition and hypothesis. The contributions are the result of this problem definition, proposing the general and specific objectives to further develop a global methodology. Finally, the manuscript outline is presented to address each individual specific objective. This resulting outline serve as a guide for the thorough content development of the following Chapters.

1.1 Heterogeneous computing

The demand of computing power has not slowed down since the dawn of the computing term itself. As a matter of fact, this demand was already present way before the birth of electronics and even mechanical computers. One can argue, that current processing platforms are only catching up on the most feasible problems that mathematicians, physicians and scientists have already formalized since centuries. Human innovation through technology has allowed to profit from this never-ending progress in a more tangible manner. As a consequence, this pragmatism has led to an environment of problem solving devices adapted not only for high-end computing centers, but also for our daily life. Perpetuating this way, the requirements for more capable processing devices. Nevertheless, as it will be discovered by the reader through this dissertation, this processing scaling is not trivial, nor inconsequential. After the first CPU conception, the most evident step for scaling was to increase the number of cores, giving birth to the concept of *multiprocessor*.

The parallel capabilities and limits of multiple processors against single core processors was first presented by Gene Amdahl and its well-known Amdahl's law [Amd67]. This

work awakened skepticism in the scientific community and extensive discussions between many experts since the early stage of the multiprocessors era [Gus88]. However, a common recurrent topic in these debates is the fraction of parallelism and serial execution found in a given application to benefit from speed-up. The presence of parallelism is a key component for *multiprocessor acceleration* or speed-up on hardware [DRPDDP15]. An *accelerator* is a processing device capable of instruction execution to perform some computation. This software parallelism and hardware acceleration co-dependency must be well described to achieve efficient algorithmic porting into the accelerator architectures. In this manuscript, this is referred as *integration*. To attain this level of efficiency, it is essential to describe the platform in terms of its hardware architecture nature. The first multiprocessor accelerators were considered as *homogeneous*. This is because platforms mostly replicated the same core or processing element with the same architecture capabilities and executing the same instructions. As more and more developers and users made the transition between sequential to parallel programming, newer paradigms were adopted.

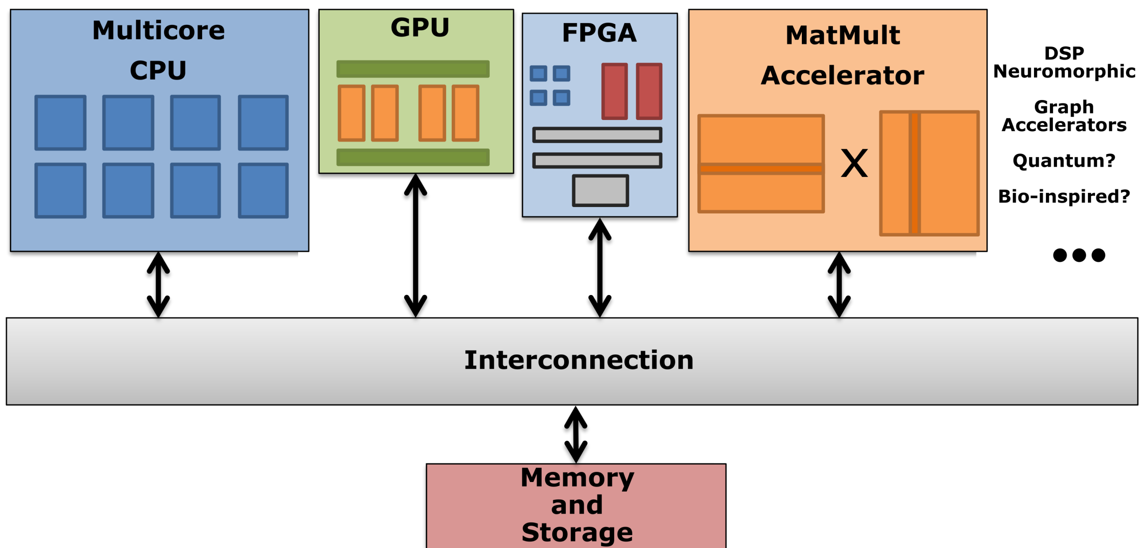
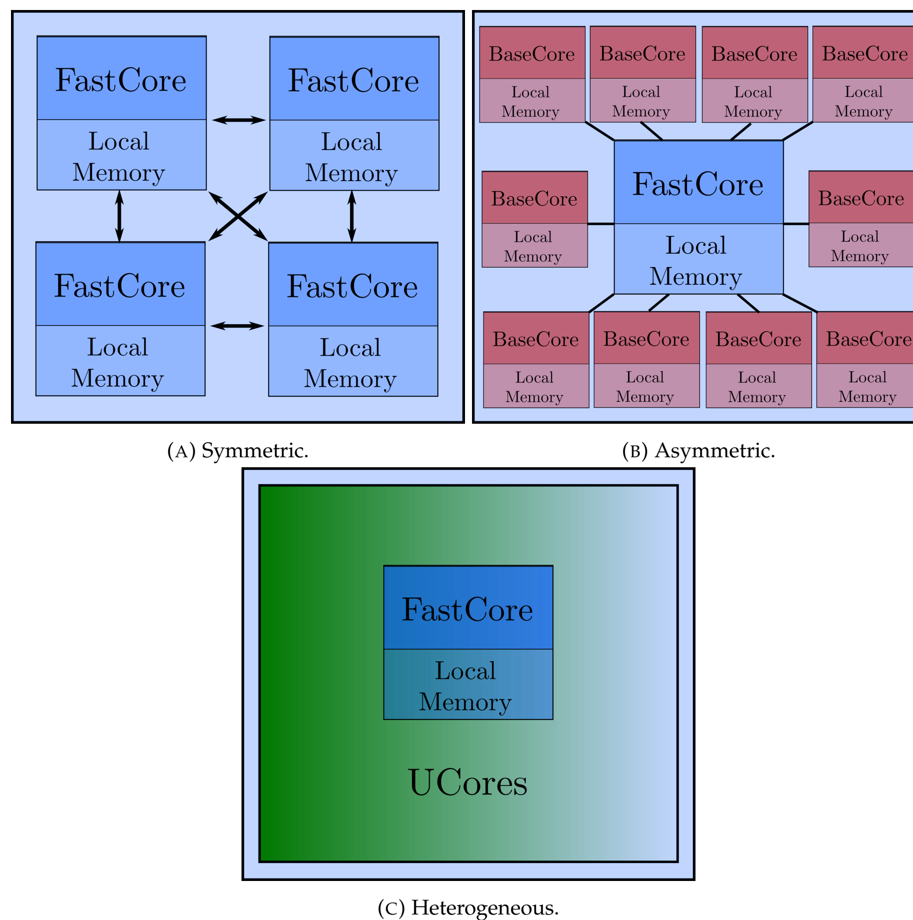


FIGURE 1.1: Generic heterogeneous architecture [Zah17].

Heterogeneous computing was introduced to address the *heterogeneity* present in core capabilities or specific ways to handle instructions [Zah17]. However, the term heterogeneity has not a formal nor strict definition. Different levels of heterogeneity can be achieved. Homogeneous multiprocessor architectures can fetch and execute different instructions; like execution synchronization, while the others are fully dedicated to computing. Therefore specializing some cores to an specific task. Another level of heterogeneity could be the type of hardware architectures deployed in our hardware. Heterogeneous architectures can be composed from three main accelerator groups: symmetric, asymmetric and Unconventional Core (U-Core) [CMHM10]. Symmetric cores were the first based on identical core replication, like the first multicore CPUs. Asymmetric focused on specializing some cores to specific tasks or capabilities. Finally, U-Cores cover a wide spectrum of programmable custom-logic devices, highly specialized processors or emerging technologies and how they interact to each other. In this thesis, we focus on the latest one. Figure 1.1

shows a generic heterogeneous architecture incorporating many symmetric or asymmetric accelerators, like: CPUs to U-Cores, like: GPUs, FPGAs, Digital Signal Processors (DSPs), matrix-multiplication accelerators and emerging technologies. However, some heterogeneous architecture include more complex memory models, where heterogeneity is also present in the way these accelerators communicate. For instance, the hybrid accelerator architectures from Figure 1.2 lacks the interconnection from Figure 1.1. Thus, the communication is defined by the neighborhood topology and/or visible memory components on each core. Figure 1.2 shows different multicore accelerator types from [CMHM10]. The arrows represent a dedicated communication link between cores. Fast cores are usually single-core processors to execute sequential instructions with their own resources, like local memory and caches. While the base-core are equivalent to execute parallel portion of instruction execution. Some multicore topologies, like in 1.2a, include a broadcast communication style, where all processor can communicate with each other. While some others have a main sequential processor to orchestrate task execution of similar base cores, as depicted in 1.2b. Finally, in Subfigure 1.2c, a plethora of Ucores are used to deploy parallel execution and communicate directly with the main core. These Ucores may vary in capabilities from each other and include different types of communication links.



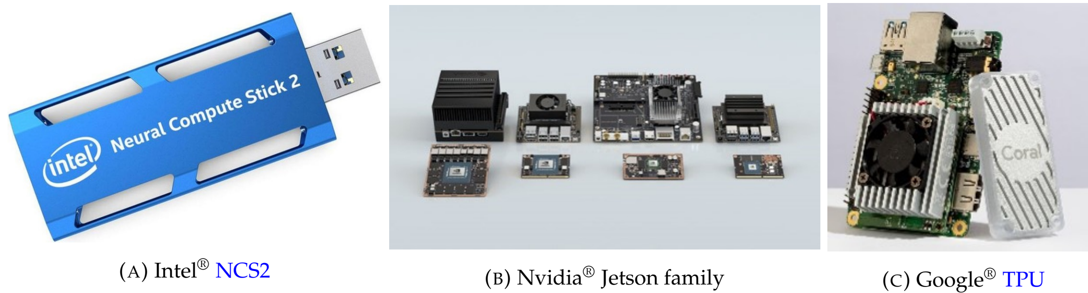
(A) Symmetric. (B) Asymmetric.

(C) Heterogeneous.

FIGURE 1.2: Multicore types according to [CMHM10].

The scope of this dissertation is delimited by the type of U-Cores adopted on the heterogeneous platform architecture. Two individually well-studied accelerators are

explored. On one hand, the GPU is based on core replication with hundreds (embedded), thousands (personal computers) or even tenths or hundred of thousands (data centers) of processing elements. These cores operate in a Single-Instruction Multiple-Data (SIMD) fashion and each core can have a different workload attributed by a scheduler. This is, that they execute the same operation over different data chunks or batches, known as *warps* or *wavefronts*, depending on the vendor. At the other hand, the FPGA takes advantage of its programmable configuration to create custom designs at expense of limited logic and memory resources.



(A) Intel® NCS2 (B) Nvidia® Jetson family (C) Google® TPU

FIGURE 1.3: Some embedded heterogeneous platforms.

These heterogeneous platforms, as depicted in Figure 1.1, are a well-established environment paradigm in High Performance Computing (HPC) data centers since the last decades. This quick adoption has been widely accelerated because of the demand of current DL algorithms [TAI+20]. In these specific cases, considerations such as energy consumption, resource utilization or memory transfers; play a lesser role to achieve lower latency and higher throughput. This is however, not the case for *embedded* platforms, where usually a fragile trade-off between system performance, energy efficiency and resource constraints is crucial [HSKR21]. Figure 1.3 shows some embedded heterogeneous platforms with U-Cores. Subfigure 1.3a shows the Intel® Neural Compute Stick 2 (NCS2) [Mov19] with aDSP-GPU microarchitecture. Subfigure 1.3b shows the Nvidia® Jetson family of embedded GPUs, which some of them include a dedicated Application Specific Integrated Circuit (ASIC) for DL, named tensor cores. Subfigure 1.3c shows the Google® Tensor Processing Unit (TPU) [JYP+17]. With the rise of this new generation of heterogeneous platforms *on the edge*, a well-defined development ecosystem must be carefully organized to find adequate solutions, this is known as Design Space Exploration (DSE). In the following subsection, the deployment implications of embedded heterogeneous computing for CNN model inference are discussed as motivation behind this manuscript.

1.2 Challenges and problem definition

In this thesis, the inference process of CNN models for image classification tasks is studied. In this context, as seen in the upper part of Figure 1.4, a pre-trained CNN model takes as an input an image and it outputs a class label. Once a CNN model is implemented in a hardware platform, also known as the *system*, several computational operations are performed to generate this class label. The required computation has a cost in terms of

time, energy and resources. Such metrics are useful to evaluate the efficiency of a given system, these metrics are commonly known as Key Performance Indicators (KPIs). A KPI is a quantifiable measurement that illustrates the effectiveness of the system towards a specific goal. These KPIs depend on several variables, like the dimensions of the CNN layers, the topology of the network, the internal architecture of the system or the image patterns. Since it is an overwhelming task to control all these variables, it is desired to estimate a *model* from KPI observation from multiple configurations of these CNN layers.

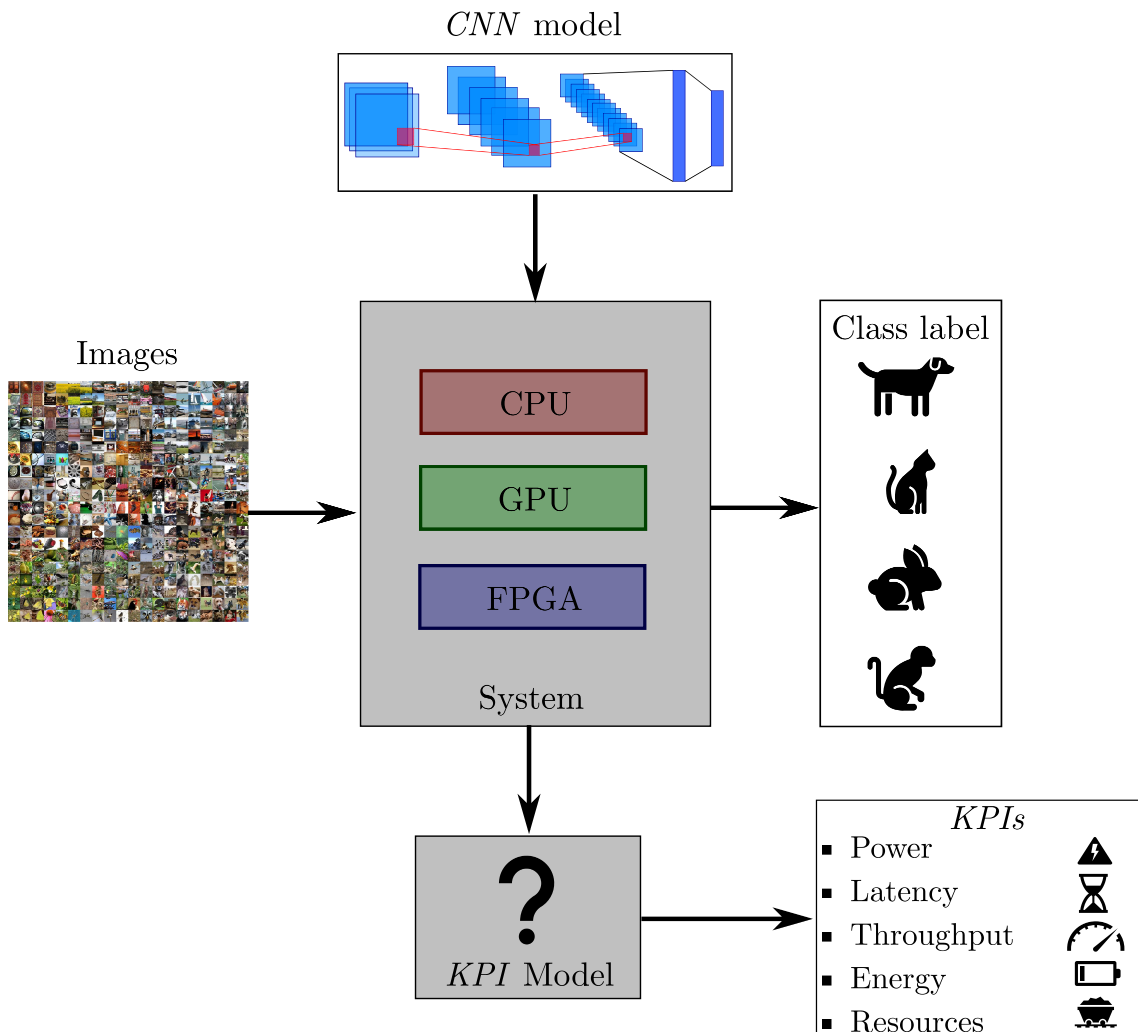


FIGURE 1.4: KPI modeling of a heterogeneous platform as system.

More generally, as it will be slowly introduced during the development of the dissertation, the CNN inference is a computational model that operates over different data workloads called Feature Mapss (FMs), not only images. Each processing block operates over an Input Feature Map (IFM) and produces an Output Feature Map (OFM). The details of these operations are described extensively in the following chapters. Momentarily, it is relevant how these memory transfers are handled and mapped. This is because, from a hardware perspective, memory hierarchy is one of the main challenges and bottlenecks in heterogeneous platforms. Several levels in a memory model like on-chip memory, external memories, local or global memories, shared or private resources; have a direct impact on

system performance. The accelerator topology and its interconnection is equally important [Zah17]. Similarly, selecting the distributed workload on a heterogeneous platform is not a simple task. Scheduling relies on data dependency and how each individual accelerator is synchronized. Additionally, some partitions may be well-suited for an specific platform, but quite inefficient on another. This creates a generalization limitation dependant on the specific hardware capabilities. All these challenges become more complex considering the typical embedded design constrains [HBNY19]. Figure 1.5 illustrates the partitioning of a CNN model of computation. Each resulting partition is mapped to an individual accelerator on a custom heterogeneous architecture.

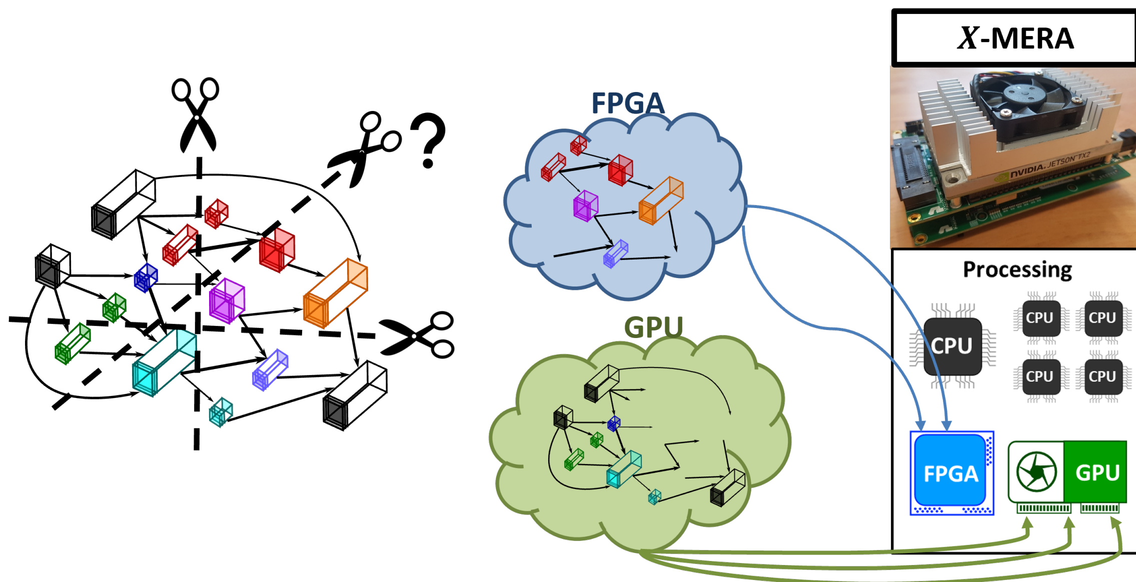


FIGURE 1.5: CNN model partitioning and mapping into an embedded heterogeneous platform with multiple accelerators.

Summarizing, the main challenges covered in this manuscript are listed below:

- Heterogeneous hardware platform performance characterization and evaluation.
- CNN model partitioning, mapping and scheduling.
- Inter-device memory transfers handling.
- Selecting appropriate design solutions considering limited embedded resources.

The following questions are derived from the challenges to guide this thesis and set the contribution basis:

- How to efficiently deploy a CNN topology on a hybrid platform?
- How can a hardware-aware design solution be evaluated?
- Which KPI metrics should be chosen for evaluation?
- Which CNN configuration features should be selected to characterize the heterogeneous platform?

- How to split the CNN model for a better efficiency?
- How to guarantee optimal solutions and time efficiency?

From these questions, a more formal **hypothesis** definition is proposed:

For pretrained CNN model inference, an optimal set of mapped partitions can be obtained from an optimization problem formulation and efficiently deployed on embedded heterogeneous platforms, where each accelerator is analytically characterized and a set of partitioning rules are chosen.

1.3 Contributions

The general objective inferred from the hypothesis is *the obtainment and mapping of these optimal accelerated partitions result of the optimization problem*. However, to achieve this goal, three specific tasks must be individually resolved: the *modeling*, the *partitioning* and the *optimization*. The main content development of this thesis is depicted in Figure 1.6.

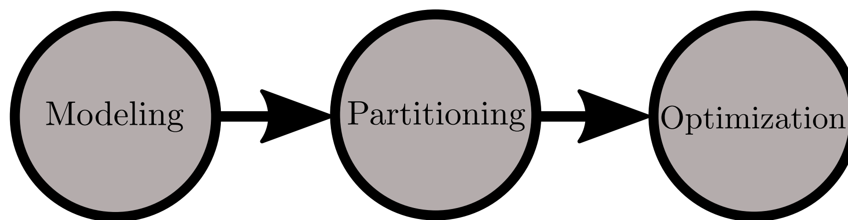


FIGURE 1.6: Thesis content development flow.

The contribution outcome of the manuscript is enumerated below:

1. A methodology to obtain a set of analytical models that describe the hardware KPI efficiency of a heterogeneous platform for CNN workloads.
2. Adapting state-of-the-art partitioning, mapping and scheduling techniques to heterogeneous platforms.
3. An optimization problem formulation with an optimal solution that can be efficiently solved in terms of time and computational complexity for DSE.
4. A software-hardware co-design environment for CNN partition acceleration with high-level abstraction.
5. A smart camera prototype that benefits from the capabilities of a CPU-GPU-FPGA embedded heterogeneous accelerators as test platform. This smart camera is named *X-MERA*.

1.4 Manuscript outline

This first chapter, served as introduction for the main motivation, challenges and contribution of this manuscript. The basis for the content of each chapter has also been established.

The following chapters cover the main body and contributions described above. For the next three chapters, an introduction and state-of-the-art related works are presented that set the context and motivation behind each chapter. Additionally, each chapter conclusion sets a transition between the results and their impact for the subsequent chapters. The three main body chapters are briefly described as:

- Chapter 2: This chapter focuses on the heterogeneous hardware architecture characterization through *performance modeling* with analytical mathematical models for CNN layer configurations. The resulting models are the outcome of a methodology inspired from a black-box System Identification (SI) approach with random sequence excitation. The chapter addresses the contribution 1.
- Chapter 3: This chapter covers the mapping and scheduling adapting *heterogeneous partitioning techniques* to the already described hardware models for mobile CNN models from Chapter 2. The experimental results compare the heterogeneous partitioning against homogeneous solutions at a module-level partitioning. The chapter addresses contribution 2.
- Chapter 4: This chapters formalizes the mathematical *CNN model partitioning optimization* problem. This problem formulation is elaborated in a context for Geometric Programming (GP) and Generalized Geometric Programming (GGP) optimization. This optimization requires some strict specific mathematical formulation, but it offers some convexity and optimal properties. The chapter address contribution 3.

The Chapter 5 further discusses the obtained results of the thesis. Furthermore, an analysis of the possible research exploration paths for this dissertation is studied. These opportunities offer new research question and a new spectrum of hypotheses for embedded heterogeneous platform design. Finally, the Appendix A lists a set of mathematical properties useful for the GGP problem formulation, such as convexity and posynomial preserving operations. This is specially useful for 4. Appendix B describes the hybrid software development environment and the heterogeneous smart camera prototype *X*-MERA used in Chapters 2, 3 and 4. This appendix addresses contributions 4 and 5.

1.5 Publications

- Walther Carballo-Hernández, François Berry, Maxime Pelcat, and Miguel Arias-Estrada. Towards embedded heterogeneous FPGA-GPU smart camera architectures for CNN inference. In *Proceedings of the 13th International Conference on Distributed Smart Cameras*. ACM, sep 2019
- Walther Carballo-Hernández, Maxime Pelcat, and François Berry. Why is FPGA-GPU heterogeneity the best option for embedded deep neural networks? *Presented at DATE Friday Workshop on System-level Design Methods for Deep Learning on Heterogeneous Architectures (SLOHA 2021)*, February 2021

1.6 Submissions

- Walther Carballo-Hernández, Maxime Pelcat, Shuvra S. Bhattacharyya, Ricardo Carmona Galán, and François Berry. Flydeling: Streamlined performance models for hardware acceleration of CNNs through system identification. Under review for ACM Transactions on Modeling and Performance Evaluation of Computing Systems (ToMPECS), 2022
- Walther Carballo-Hernández, Maxime Pelcat, Ricardo Carmona Galán, and François Berry. A module-level CNN partitioning method for FPGA-GPU heterogeneous acceleration. 2022

Funding acknowledgements

This project has received funding from the European Union's Horizon 2020 research and innovation program under the Marie Skłodowska-Curie grant agreement No 765866.

Chapter 2

Flydeling: Performance Models for Hardware Acceleration of CNNs through SI

2.1 Chapter abstract

As covered in Chapter 1, DL algorithms, such as CNNs in many near-sensor systems, opens new challenges in terms of energy efficiency and hardware performance. An emerging solution to address these challenges is to use tailored heterogeneous hardware accelerators combining processing elements of different architectural natures such as CPU, GPU, FPGA or ASIC. In order to progress towards heterogeneity, a great asset would be an automated design space exploration tool that chooses, for each accelerated partition of a CNN, the most appropriate architecture. To feed such a design space exploration process, models are required that provide very fast yet precise evaluations of alternative architectures or alternative forms of CNNs. Quick configuration estimation could be achieved with few parameters from representative input sequences. This chapter studies a solution called *flydeling* (as a contraction of flyweight modeling) for obtaining these models by inspiring from the black-box System Identification (SI) domain. We refer to models derived using the proposed approach as *fly*weight mo*dels* (*flydels*).

In this chapter, a methodology is proposed to generate these *flydels*, using CNN properties as predictor features together with SI techniques with a stochastic excitation input at a feature map dimensions level. For an embedded CPU-FPGA-GPU heterogeneous platform, it is demonstrated that it is possible to learn these KPIs *flydels* at an early design stage and from high-level application features. For latency, energy and resource utilization, *flydels* obtain estimation errors varying between 5% and 10% with less model parameters compared to state-of-the-art solutions, and are built automatically from platform measurements. A special form of these models (monomial and posynomial forms) are employed in Section 4, in combination with partitioning techniques from Section 3, to characterize heterogeneous platforms.

2.2 Introduction

DL models are increasingly integrated into edge devices for a vast set of applications such as computer vision and speech recognition. To support the strong computing requirements of DL, heterogeneous architectures with application specific acceleration are the dominating solution for edge DL [VGG+20, TST+19, OHY+18], taking over single-processor systems. However, when using traditional programming and hardware description languages, designers are required to build an advanced prototype of their heterogeneous system before obtaining reliable processing performance estimates of their CNN implementation in terms of energy, execution time and resource utilization, given the complexity of such hybrid systems. This constraint leads to a late system performance evaluation and to relatively blind design choices. This drawback is intensified by the demand of efficient and quick comparison between different CNN architectures on constrained platforms.

In order to introduce hardware-awareness to the heterogeneous system partitioning of DL algorithms, many models have been proposed that give an insight on application performance [VGG+20, TST+19, OHY+18]. Nevertheless, these models require low-level features which are non-trivial to obtain by the developer, like micro-operations performance or per-instruction performance. Furthermore, these architecture-specific models are only used to analytically describe single-architecture devices. Thus, making it non-obvious how these can be combined to extend on heterogeneous platforms with hybrid architectural specifications. A heterogeneous architecture platform is composed of two or more devices. Where a device, is defined as any processor capable of computing an specific workload with its own resources; such as memory and its own computing elements. These local device-specific resources are also commonly known as on-chip resources. Most modeling techniques are trained on datasets that do not consider variations of internal memory transfers between these devices. Because modern compilers tend to optimize memory usage by keeping static variables in device memory, they carefully handle memory transfers and this optimization creates biased datasets that do not perform well over applications outside the assumption of keeping static variables in the on-chip memory. As a solution, a *black-box* identification technique that builds analytical behavioural models is proposed, inspired from the domain of System Identification (SI) of control systems, and adapt it to CNN modeling. To avoid modeling over only memory-optimized applications, a methodology is proposed to excite the system and extend to a broad application range. This procedure allows generalization without relying on over-confident models trained on biased datasets. It is hypothesized that CNN hyper-parameters are sufficient as a representation of an application activity. Therefore, using these CNN processing block properties as an application-specific set of features, a measurement-based dataset is obtained for energy, latency and resource utilization on an embedded CPU-GPU-FPGA platform.

The main contribution of this section consists of a methodological derivation of an ensemble of lightweight behavioural models (called *flydels*) from a randomly excited dataset. The ensemble is composed of single-variable linear and non-linear (logarithmic, exponential and polynomial) models using Least Squares Error (LSE) minimization for

each individual computing device modeling. A multi-variable parametric model is derived. Finally, a method to validate the obtained model precision metrics is proposed using K-fold cross-validation by demonstrating that their numerical variations are the result of a random process.

In this chapter, three major assumptions are made: ① It has been demonstrated that, not only is the convolutional layer the most common operation in CNN models, but also the most time consuming and parameter intensive operation [CX14]. For instance, AlexNet convolutional layers with 256 images require around 90.7% of the total processing time [CX14], while in more recent models, such as ResNet, require up to more than 99% of processing [HZRS15]. As modeled by Amdahl's law, enhancing the biggest fraction of the process is the way to proceed when minimizing the latency of a given task [Amd67]. Efforts are thus focused on obtaining an analytical behavioural model for convolutional layers. ② Many heterogeneous acceleration solutions currently aim at speeding up a chosen *partition* of a whole CNN pipeline [MCVS20, GKBS20, WXH21]. Therefore, modeling a variety of these accelerated partitions is a critical task. In the context of this work, an *accelerated partition* is defined as a connected graph subset result of a disjoint decomposition of the original CNN. ③ The Design Space Exploration (DSE) problem consists in determining the appropriate partition and its configuration selection. Thus, the efforts are concentrated on each sub-network performance analysis that constitute the DSE. Moreover, a heterogeneous platform can be analytically represented by a combination of individual processing and communication models [SSEM19, ZWTD19, VGG+20]. As a consequence, a composite heterogeneous profiling is affected by the reliability of each model, which errors shall be confined within application-related acceptable bounds.

This manuscript is organized as follows: Section 2.3 summarizes related works. Section 2.5 gives an overview of the main concepts of linear and non-linear parametric System Identification (SI). Section 2.4 defines the problem at hand, the CNN operations, and the ensemble methodology. It also introduces important concepts for single-variable up to multi-variable profiling using ensembles. The main challenges of system modeling estimation are highlighted. The experimental methodology description is given in Section 2.6, where also the dataset generation is explained and the experimental results are evaluated and validated. Finally, Section 2.7 concludes the chapter with a discussion of main results and contributions.

2.3 Related works

The DSE on embedded systems requires, in most cases, an evaluation of a vast collection of design points, which are impossible to explore by brute force [Pim17]. Additionally, the evaluation of such designs points may be complex to realize and therefore quite time consuming. In [Pim17], Pimentel categorizes design point evaluation methods into 1) implementation measurements, 2) simulation, and 3) analytical models. Each evaluation method offers a trade-off between estimation speed and precision, the measurements-based method being the most precise but slowest of solutions; while analytical models offer

highest evaluation speed at the cost of a precision loss given the reduced number of degrees of freedom. Analytical modeling itself is divided into three groups: mechanistic (white-box) modeling that builds on advanced knowledge of the system structure, empirical (black-box) modeling that builds on simplified assumptions on the system structure and hybrid (gray-box) modeling, where only some few parameters of the system are known [Eec10]. Depending on the *a-priori* knowledge of the internal mechanics of the system, different techniques have been proposed to estimate performances. For taking early design decisions, performance evaluation speed is prioritized over evaluation precision. Indeed, most system features are subject to further modifications so extreme precision is in general useless. This work aims at automatically deriving analytical models from system simulation and implementation measurements. As heterogeneous systems are getting more architecturally complex over time and many elements are confidential, a black-box modeling method is proposed to fit learning processing systems performances.

Predictive modeling has been deeply explored as an essential stage for efficient DSE, and this on multiple devices and KPIs [OB18]. As an example, energy modeling on embedded heterogeneous processors has been studied by Seewald, et. al in [SSEM19] by aggregating sets of system measurements. Authors' system performance modeling solution is based on an averaging approximation over multiple real-time obtained traces. As a result of such studies, heterogeneous system design can take advantage of latency and energy modeling with three different approaches, as proposed by Pimentel : mechanistic (white-box), empirical (black-box) and hybrid (gray-box) [Pim17]. In [GMRRG19], multiple works over a large number of processors are evaluated with analytical models and regression techniques. The proposed taxonomy is helpful to understand what is being modeled and which features are being used to predict performance. The use of the authors' three modeling levels is chosen:

Mechanistic (white-box) modeling: White-box models use low-level features to approximate the processor performance. The commonly selected features are extracted from Performance Monitoring Counters (PMCs) events associated with each (architecture-specific) instruction. Since the specifications of the hardware and the instructions required to execute a task are known, it is simpler to determine the performance of the system. Traditional performance estimation from an architectural perspective uses a mixture of linear and non-linear regression models with identified features such as main memory accesses, hierarchical cache events and processing latency [LB06]. These methods reach extremely high prediction precision but are not meant for generalizing to other instruction sets, and require millions of instruction-level traces for predicting KPIs over a full application. Architectural diversity between processing accelerators and large-scale application thus limit applicability of such prediction methods. Moreover, the feature selection among many PMC events remains a challenging issue to feed the proposed models. Some works on CPUs instruction-level modeling focus on selecting predominant PMCs features for energy and power performance [SFML19] or resource mapping per micro-instruction for several tasks, such as Fast-Fourier Transform (FFT) or GEneral Matrix-Multiplication

(GEMM) [DGB+20]. Leveraging on these models, tests have been proposed for model selection over a variety of features and KPIs with simple models using Linear Programming (LP) or Integer Linear Programming (ILP). Focusing more on parallel workloads, works have studied power estimation modeling and simulation on GPUs [BIM16] [NMN+10] using PMCs on Compute Unified Device Architecture (CUDA) kernel applications with simple linear regression, and reaching 4.7% of error on processing power estimation. Other studies discuss the use of traditional regularized linear regression techniques such as Lasso and non-linear regression techniques such as Random Forest (RF) [OBSK17]. In [OBSK17], authors select the most important features from a set of PMCs on different workloads, minimizing Ordinary Least Squares (OLS) and Non-Negative Least Squares (NNLS).

Empirical (black-box) modeling: Studies at an empirical level rely only on application measurements to obtain modeling features and thus model the implementation decisions (architecture design, mapping, scheduling, timing, memory management) as part of the "black-box". They thus tend to have a reduced capacity of generalization but models tend to be extremely compact. In [VMFBCGRV20], the performance of most popular CNNs topologies is modeled when executed on embedded CPU devices, using the CNNs layer hyperparametes as features with linear regression. If multiple models are built for multiple design choices, the performance of heterogeneous systems can be evaluated (taking some strong hypothesis on time and energy additivity). In [JZK+20], a procedure is proposed to estimate CNN inference time from analytical models for multiple architectures, using CNN hyperparameters as features. Authors' solution is based on the Roofline model [WWP09]. By determining the Multiply and ACcumulate (MAC) and data transfers requirements on each CNN layer, authors estimate latency with 88% and 98% precision on AlexNet [KSH12] and LeNet [LHBB99] respectively. This analysis is restricted to layers for two popular CNN architectures. In this chapter, it is aimed to further extend the generic CNN performance modeling by proving that it is possible to obtain automatically some analytical models for general CNN properties sizes, for different devices and different KPIs.

Hybrid (gray-box) modeling: If a partial number of architectural resources are known or can be approximated, they can be used as a feature for performance estimation. This is especially useful for custom logic development on FPGAs or ASICs where KPIs are more affected by architecture choices than by the algorithm-related transistors activity. Machine Learning (ML) methods like Artificial Neural Networks (ANNs), Support Vector Machines (SVMs) and RF applied to modeling resource utilization and throughput estimation of High-Level Synthesis (HLS) generated hardware have proven to reach 95% accuracy in favorable cases [MFS+19]. More complex application-specific modeling techniques using e.g. reinforcement learning have also reached high accuracy for FPGA performance and resource prediction [WXH21]. While these models reach high precision, they employ complex models, non suitable for large DSEs for which great numbers of decision combinations must be evaluated. When considering the specific case of CNN accelerators characterization, Ma, et. al [MCVS20] relate the latency and throughput of hierarchical

memory accesses of a CNNs accelerator to the size of their IFM. This study focuses on the memory transfers between an external Dynamic Random Access Memory (DRAM) and the on-chip memory before and after processing with less than 3% error difference. In [GKBS20], Goel, et. al create a run-time profiler to estimate CNN time execution on Xilinx® Deep learning Processor Unit (DPU) using regression techniques, and MACs operations as application-specific features. This model obtains 7% of errors on time execution estimation with Decision Tree (DT), RF and polynomial models. In [YCS17], the number of MACs operations is shown not to be sufficient for an efficient energy approximation. It is discussed how data transfers must also be taken into consideration into the modeling features. While performance prediction methods mentioned above can be executed fast, they do not automate the procedure for obtaining KPI estimates, and each study targets one or a set of specific use cases.

Table 2.1 summarizes the methods of fore-mentioned taxonomy. Although the previously analyzed state-of-the-art studies cover multiple devices, features and input datasets, each of these methods focuses on the study of a particular use case. System heterogeneity has been demonstrated to bring time and energy efficiency to CNN systems. Optimized architectures combine e.g. FPGA and GPU architectures [VGG+20, TST+19] that exploit respectively pipeline-level parallelism and data-level parallelism. However, having a white-box model for a heterogeneous platform with all hardware specifications for all CNN applications is extremely difficult. Additionally, constantly adding up novel architectures make use-case specific models fast obsolete and call for automated modeling procedures, which is one of the biggest drawbacks of white-box modeling. In this work, a hardware-generic black-box and KPI-generic method for black-box modeling are proposed, and effort are focused on assessing the confidence level of the obtained model. The method is generic to the three previously defined levels of modeling. Using the proposed method, one can derive black-box multi-variable lightweight analytical models using SI, and build them for multiple KPIs with a small number of parameters. The method builds on ensemble regression to reach this objective [MMSJS12]. The result of such analytical models, can be further deployed for gray-box modeling. For instance, extrapolating the energy consumption from resource utilization on a CNN design.

Approach	Features	Works	Models	Platform	KPIs
Mechanistic (White-box)	PMCs	[LB06]	Linear Non-linear	CPU	Latency
		[SFML19]	Linear		Energy Power
		[DGB+20]	LP ILP		Throughput
	Instructions	[BIM16] [NMN+10]	Linear	GPU	Power
		[OBSK17]	Linear Non-linear/RF		Latency
Empirical (Black-box)	CNN hyperparemeters	[VMFBCGRV20]	Linear	CPU	Latency Energy
		[JZK+20]	Roofline	ASIC	Latency
		[WWP09]	Roofline	CPU	Latency
Hybrid (Gray-box)	Resources	[MFS+19]	ANNs/SVMs/RF	FPGA	Throughput
		[WXH21]	Reinforcement learning		Throughput
	Memory accesses	[MCVS20]	Non-linear (Analytical)		Latency Throughput
	MACs	[GKBS20]	DT/RF Polynomial		Latency
		[YCS17]	Analytical		Energy

TABLE 2.1: Modeling taxonomy according to Pimentel et al. [Pim17] and García-Martin et al. [GMRRG19]

2.4 Problem definition: Applying Flydeling to CNN KPIs

This section discusses the definition of flydel and the application of flydeling SI to CNN KPI estimation. The modeled KPIs are power, latency, energy, throughput and hardware resources. Finally, we describe the ensemble of proposed models and how they are selected for a specific metric from single-variable to multi-variable regressors for FPGA, CPU and GPU.

2.4.1 Flydel definition and properties

We define *flydel* as a combination of many single-variable mathematical model to find an analytical relation between a KPI and an *activity feature*. First a single-variable model $\hat{m}(x_i)$ maps all the input activity features to a real-value, $\hat{m}\colon \mathbb{N} \to \mathbb{R}$. Where x_i is the single-variable activity feature. For this dissertation, each activity feature is a natural number, $x_i \in \mathbb{N}$ since each CNN property can only take positive integer values. While, $m \in \mathbb{R}$ to represent the physical values of the KPIs. The following Single Input Single Output (SISO) analytical models to obtain a flydel are used:

- Polynomial (n-order, $n+1$ parameters): $\hat{m}(x_i) = \sum_{j=0}^{n} a_j x_i^j$

- Logarithmic (2 parameters): $\hat{m}(x_i) = a_1 ln(x_i) + a_0$
- Exponential (3 parameters): $\hat{m}(x_i) = a_2 a_1^{x_i} + a_0$
- Reciprocal (2 parameters): $\hat{m}(x_i) = a_1 \frac{1}{x} + a_0$

where a_i is the parameter for feature x_i. This function set can easily be extended in case KPI fitting is not efficient. The n-order polynomial covers the linear and quadratic models. Using a cost function with the regularization term, the ensemble chooses less complex (lower order) models rather than over-parameterized models. As demonstrated in Section 2.6, the logarithmic and exponential models help to discover better non-linear relations over some KPIs, such as power and throughput.

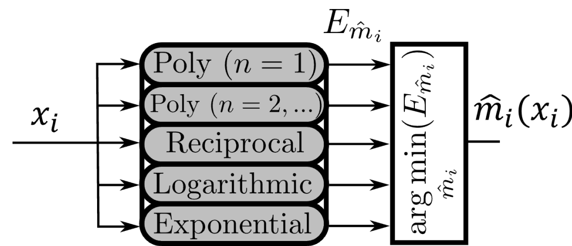


FIGURE 2.1: Flydel models.

From Sections A.2 and A.3 from Appendix A, it is possible to analyze the curvature (convex, convex or affine), monotonicity (non-decreasing, non-increasing or monotone) properties and potential posynomials form (depending if coefficients are all positive) for *flydels*. Figure 2.1 the single-variable building-block model selection for *flydeling* in Section 2.5. Table 2.2 shows the curvature, monotonicity and if the selected models can be formulated as posynomials. For more complex aggregation functions, such as *function composition* ($f(x) = h \circ g = h(g(x))$), these properties are extremely helpful to determine the properties of the resulting function. As for instance, according to monotonicity, function composition preserves curvature following the rules listed below [BV04, BKVH07]:

- $f(x)$ is convex if $h(x)$ is convex and non-decreasing and $g(x)$ is convex.
- $f(x)$ is convex if $h(x)$ is convex and non-increasing and $g(x)$ is concave.
- $f(x)$ is concave if $h(x)$ is concave and non-decreasing and $g(x)$ is concave.
- $f(x)$ is concave if $h(x)$ is concave and non-increasing and $g(x)$ is convex.

Since the aggregation function $h(x)$ from previous Subsection is an affine function, which is both concave and convex with non-decreasing curvature, the resulting function $f(x)$ will depend on the curvature of $g(x)$. This means that all functions from flydels must be convex, so the aggregation function can also be convex.

In the following subsection, we discuss the concept of *feature* in the context of CNNs inference. We further define the covered KPIs.Finally, we explain how we combine different analytical models to obtain a flydel through a methodology named *flydeling*.

[1]Using Taylor expansion approximation.

Model (Single variable)	Potential Posynomial	Curvature	Monotonicity
Polynomial (n=1)	Yes	Affine	Non-decreasing
Polynomial (n=2,...)	Yes	Pseudo Convex (over trained data)	Pseudo Non-decreasing (over trained data)
Reciprocal	Yes	Convex	Non-increasing
Logarithmic	No	Concave	Non-decreasing
Exponential	Yes[1]	Convex	Monotonous

TABLE 2.2: Flydel curvature, monotonicity and posynomial properties.

2.4.2 Using CNN properties as application activity

One of the main motivations of the deployment of heterogeneous platforms for CNN models is the presence of heterogeneity in both computation and communication CNN tasks. In [LHBB99], the modern concept of CNN is introduced using a gradient-based learning and parameter sharing over previously defined ANN, inspiring from the biological visual cortex. In this subsection, we briefly describe the most common computing operations of CNN models:

- **Convolutional layer (Conv):** A convolutional layer receives as an input a multidimensional tensor (Input Feature Map (IFM)) I of size $H_I \times W_I \times C_I$ from a previous layer $l-1$. This tensor is multiplied and accumulated with a sliding window of a kernel tensor K of size $k_h \times k_w \times C_I \times N$. Typically, the kernel matrices are square so $k_h = k_w$. The resulting OFM O on the current layer l is computed as:

$$O(x,y,n) = \sum_{c=1}^{C_I}\sum_{i=1}^{k_h}\sum_{j=1}^{k_w} K_n(i,j,c)\cdot I(x+i\cdot s_h, y+j\cdot s_w, c) \tag{2.1}$$

  while the OFM dimension are computed as:

$$\left\lfloor \frac{H_I - k_h + 2p_h}{s_h} + 1 \right\rfloor \times \left\lfloor \frac{W_I - k_w + 2p_w}{s_w} + 1 \right\rfloor \times N \tag{2.2}$$

  where s_h and s_w are the stride in each direction of the 3D IFM [GlLJ18]. Strides perform downsampling and make the size of the tensors reduce when progressing into the network. They are particularly used in classification purpose CNNs and in the first layers of auto-encoders. In many image classification CNNs $s_h = s_w = 1$. p_h and p_w are the padding on both dimensions used to preserve the border pixels of the image, so that $W_O = W_I$ and $H_I = H_O$, when $p_h = p_w = \lfloor \frac{k}{2} \rfloor$. When increasing the stride, the size of the output tensor is reduced.

- **Fully Connected Layer (FC):** The CNN networks presented in this manuscript include in the last layers a classification ANN where its kernel filter interconnects all

the previous layer's neurons to the output layer. The IFM is squeezed in a vector of size $1 \times H_I W_I C_I$ multiplied, accumulated and mapped by the matrix kernel of size $H_I W_I C_I \times N$. Therefore the resulting output vector is obtained with the following expression:

$$O(n) = \sum_{x=1}^{H_I W_I C_I} K_n(x,n) \cdot I(x) + b(n) \tag{2.3}$$

- **Pooling layer:** The pooling layer serves as a spatial invariant and feature compression between layers. Pooling layers employ a variety of data downsampling methods such as max or average pooling windows of size $k_w = k_h$ with a stride $s_w = s_h$ of the size of the pooling filter. Similar to the stride in a convolutional layer, the IFM is reduced from $H_I \times W_I \times C_I$ to an OFM of $\left\lfloor \frac{H_I - k_w}{s_w} + 1 \right\rfloor \times \left\lfloor \frac{W_I - k_h}{s_h} + 1 \right\rfloor \times N$. As a side effect, inter-device intercommunication is lowered if feature map communication happens after pooling layers.

- **Activation (ReLu and TanH):** introduce non-linearities into CNNs. CNNs mostly use three different activation functions: sigmoid $f(x) = \frac{1}{1+e^{-x}}$, hyperbolic tangent $f(x) = tanh(x)$ and Rectified Linear Units (ReLU) [KSH12] $f(x) = max(x)$.

One of the main challenges for system identification is the selection of input features. It has been demonstrated in state-of-the-art that the computational and communication complexity of CNN is strongly linked to the size of IFMs and OFMs [ZWTD19]. As shown in Equation 2.1 in Section 2.5, these tensor maps depend on the properties of the convolutional layer. Thus, these properties as application activity features are selected:

- $H_I \times W_I$: Input tensor width and height on the IFM.

- C_I: Input channel depth of the IFM.

- k: Size of convolution filter $k_h \times k_w$ with $k = k_h = k_w$. This variable modifies the size of the OFM.

- N: Number of convolution filters. This variable modifies the size of the OFM.

The following subsection describes how these application activity features are used to establish a relation with the KPIs. Expressed differently, for this use case study, the elements used for the set $\mathcal{X}$ in the tuple $\mathcal{S}$ are $\mathcal{X} = \{H_I, W_I, C_I, k, N\}$.

2.4.3 Device Key Performance Indicators (*KPI*) models

A KPI is a quantifiable measurement that illustrates the effectiveness of the system towards a specific goal. The objective is to demonstrate the capability of flydeling by finding the analytical performance models $\hat{M}(H_I, W_I, C_I, k, N)$ of a CNN computation block given a set of configurations and numerical values. These models estimate a given KPI. This must be done for each individual device;$\hat{M}_{CPU}$, $\hat{M}_{GPU}$ and $\hat{M}_{FPGA}$, so as to characterize each processing component of the heterogeneous platform. The estimated models $\hat{M}$, to be trained from platform measurements, comprise the following KPIs:

- **Power** $\hat{P}(H_I, W_I, C_I, k, N)$: Average power dissipation of a CNN layer in W.

- **Latency** $L\hat{A}T(H_I, W_I, C_I, k, N)$: Average latency or time execution in ms for the processing of one IFM.

- **Energy** $\hat{E}(H_I, W_I, C_I, k, N)$: Average energy consumption in J of the processing of one IFM. The energy is highly correlated to power consumption and latency. As a matter of fact, if both the instant power consumption and time execution of a task can be efficiently modeled, then the energy can be calculated by the integral of instant power in a time interval (from t_i to t_f):

$$\hat{E}(t) = \int_{t_i}^{t_f} \hat{P}(t)dt \tag{2.4}$$

  Moreover, the energy can be approximated by employing the average power consumption:

$$\hat{E}(t) = \hat{P}(t) \cdot L\hat{A}T(t) \tag{2.5}$$

  This is useful when the variations of power consumption are due to measurement noise and the evaluated task remains the same (same configuration and same numeric values of operations).

- **Throughput** $\hat{T}(H_I, W_I, C_I, k, N)$: Throughput of a task in GB/s. Similar to the energy model, the throughput can be inferred from the latency model. For instance, for fully pipelined implementations of n CNN tasks, the throughput can be obtained by computing the reciprocal of the total execution time:

$$\hat{T}(H_I, W_I, C_I, k, N) = \frac{1}{\sum_n L\hat{A}T_i(H_I, W_I, C_I, k, N)} \tag{2.6}$$

  On the other hand, for a parallel execution, the throughput depends on the slowest task allocated on an accelerator:

$$\hat{T}(H_I, W_I, C_I, k, N) = \frac{1}{\max(\hat{M}_{CPU}, \hat{M}_{GPU}, \hat{M}_{FPGA})} \tag{2.7}$$

- **Resources** $\hat{R}(H_I, W_I, C_I, k, N)$: Resources mapped on the device accelerator. In the case of the FPGA, it can describe the number of logic elements like Adaptive Look-Up Tables (ALUTs), Adaptive Logic Modules (ALMs), Logic Array Blocks (LABs) or memory components like registers, M20K blocks, memory LABs or Block RAMs (BRAMs).

- **Communication latency** $L\hat{A}T_{Comm}$: Inter-device communication latency between processing elements platform. This KPI is directly proportional to the bit width precision and number of elements in the OFM tensor to be transferred. The total size of the data transfers can be computed using Equation 2.2 for each layer.

Although KPIs are user-defined, they can be dictated by the system requirements. Depending on the objectives of the system, some KPIs can be obtained as a composition of more basic KPIs. As for instance, the energy consumption can be calculated by the element-product of latency and power average, or $\hat{E} = L\hat{A}T \cdot \hat{P}$. Similarly, in a non pipelined implementation with time execution dependency, the throughput is simply the inverse of the latency, or $\hat{T} = \frac{1}{L\hat{A}T}$. In this Chapter, set $\mathcal{M}$ in the tuple $\mathcal{S}$ is defined with these KPIs or $\hat{M} = \{\hat{P}, L\hat{A}T, \hat{E}, \hat{T}, \hat{R}\}$ with $\hat{M} \in \mathcal{M}$. The following subsection describes how these single-device models are obtained and built together to estimate a multi-variable model for the different KPI using an ensemble technique.

2.4.4 Automated model selection with the competitive ensemble modeling

In previous subsections, it has been stated the objective of predicting KPIs from CNN hyperparameters used as application activity features, and proposed to use regression methods to estimate model parameters 2.4. However, the objective is for the designer not to guess the mathematical relationship between activity and KPIs but rather to let flydeling find the best function. This multi-variable or multi-predictor model assumption can be difficult in practice but several heuristic-based techniques exist to obtain a suitable model. The proposition is to use a set of single-feature models, also known as an *ensemble*, to find the relation between each activity feature and each KPI individually. For this purpose, the dimension of $\hat{M}\colon \mathbb{N}^5 \to \mathbb{R}$ is reduced to $\hat{m}(x_i)\colon \mathbb{N} \to \mathbb{R}$. In this dimensionally-reduced model $\hat{m}(x_i)$, x_i can take the value of any activity feature $x_i \in \mathcal{X}$ with $\mathcal{X} = \{H_I, W_I, C_I, k, N\}$ while $\hat{m}$ is an element of $\hat{M}$ ($\hat{m} \subset \hat{M}$) targeting any KPI $\hat{m} \in \{\hat{P}, L\hat{A}T, \hat{E}, \hat{T}, \hat{R}\}$.

In Section 2.3, ensemble learning is motivated by the combination of diverse models to improve a final prediction. In this chapter, it is proposed to use the single-variable models defined in the previous subsection as weak learners or weak regressors. As discussed in [FRW02], for competitive ensembles, low error correlation and model diversity are desired attributes for the base models. To achieve these purposes, techniques such as dataset subsampling and multiple regression methods, are used to get a better generalization [MMSJS12]. To avoid error correlation, each set of weak-regressors is trained with a different dataset based only on a single feature variable as input, keeping constant the other features. This implies, for example, that some models will be trained to learn the relation between the power performance P and the number of kernel filters N, while others will fit the relation between throughput T and the convolution filter size k, etc.

Competitive ensemble modeling for regression, as many other competitive ensemble techniques, requires the definition of two stages: selection and aggregation [FRW02]. The first stage, illustrated in Figure 2.2, usually known as *selection*, is where lies the core motivation of competitive ensembles. In this first step, a deterministic or stochastic criterion is defined to choose the best performing models. The extraction of a single model per activity feature x_i is chosen, by taking the model $\hat{m}_i$ with the smallest cost function $\mathcal{C}_{\hat{m}_i}$, after parameter estimation using both methods from Section 2.5: linear SI for the linear model and Levenberg-Marquardt Algorithm (LMA) for all the other non-linear models.

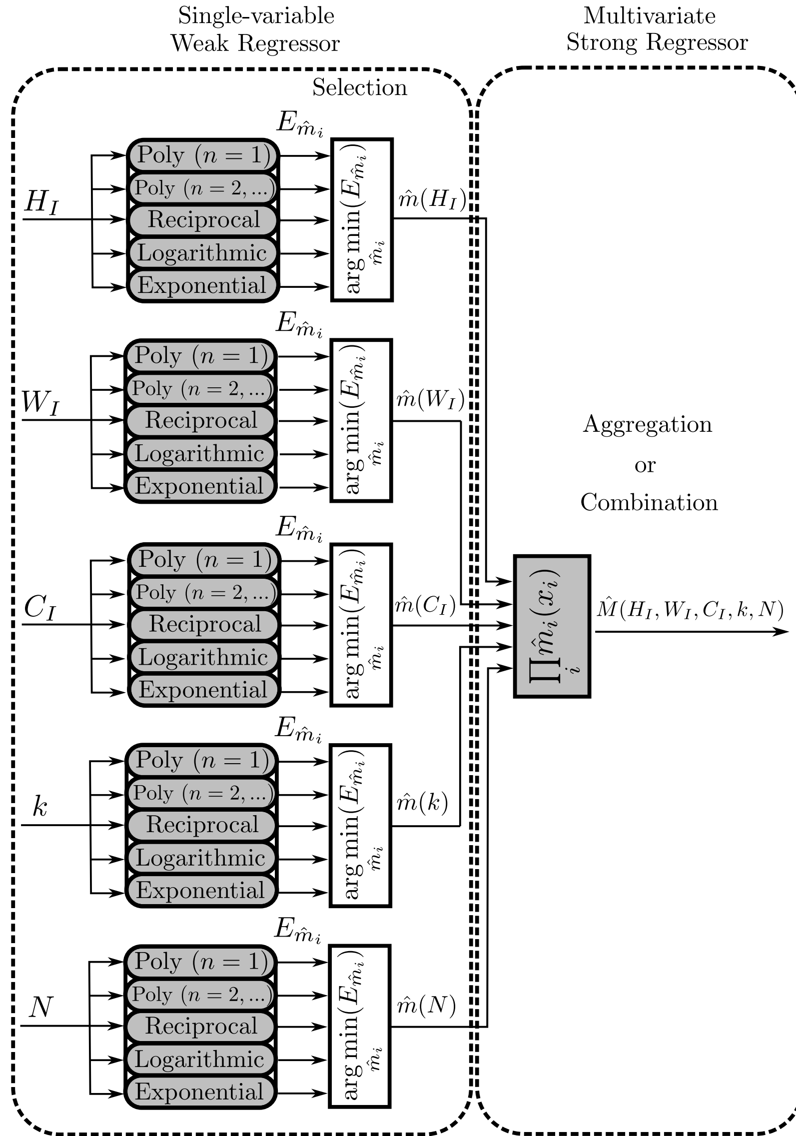


FIGURE 2.2: Competitive ensemble of single-variable models as weak regressors combined to build a multi-variable strong regressor for a performance model.

$$\hat{m}(x_i) = \arg\min_{\hat{m}_i}(\mathcal{C}_{\hat{m}_i}) \tag{2.8}$$

The cost function is defined as a combination of two terms: a loss function and a regularization term to avoid overfitting, as shown in Equation 2.9. In this work, a loss function based on the residual Root Mean Square Error (RMSE) is selected. Because it is more appropriate for dealing with a small dataset of few thousands of samples, as discussed in Section 2.6.1, regularization is important for reducing overfitting. In this work a L2 regularization term is used.

$$\mathcal{C}_{\hat{m}_i} = \mathcal{L}^{RMSE}_{\hat{m}_i} + \lambda||\theta_{\hat{m}_i}||^2 \tag{2.9}$$

The hyper-parameter λ penalizes models with a large number of parameters or with large parameters. Adjusting this value modifies the criterion of selection stage on the

ensemble. The selected loss function can be computed with the following equation:

$$\mathcal{L}_{\hat{m}_i}^{RMSE} = \sqrt{\frac{||E_{\hat{m}_i}||^2}{K}} \tag{2.10}$$

The second stage shown in Figure 2.2 is the *aggregation* or *combination*. In this step, the multi-variable model is obtained from a non-linear combination as strong regressor.

$$\hat{M}(H_I, W_I, C_I, k, N) = \prod_i \hat{m}_i(x_i) \tag{2.11}$$

The resulting model is a non-linear expression, representing the relations with the different feature variables. Product function in Equation 2.11, was selected arbitrarily to reflect the multiplicative nature of the resource utilization. It is theorized, that since the number of hardware resources increases multiplicative with respect of each IFM dimension, then also do the execution time and energy consumption. Some more traditional additive aggregation functions in ensemble learning, like averaging and weighted sum [MMSJS12], were tested with poor estimation results. Furthermore, since the previously estimated parameters do not consider the influence of the other inputs because of each feature decorrelation during training, a posterior parameter estimation must be done again at this stage. On this second identification stage, the entire dataset is used with no sub-sampling. This re-training is done at the strong-regressor SI stage. The average parameter values of the single-variable models from selection stage, is used as starting point. Algorithm 1 shows the pseudo-code for model estimation using the proposed ensemble.

Algorithm 1: Methodology for dataset generation and SI

Result: Tuple $\mathcal{S} = \langle \mathcal{M}, \mathcal{X}, \Theta, \mathcal{D} \rangle$
Input: KPI models $\mathcal{M}$ and features $\mathcal{X}$
Output: Estimated parameter Θ and dataset $\mathcal{D}$ matrices
Dataset subsampling d for weak-regressor SI;
for x_i ***in*** $\mathcal{X}$ **do** # Keep feature x_i variable and all others features z as constants
    $x_i \leftarrow$variable;
    $z \leftarrow \mathcal{X} - x_i$;
    $d \leftarrow \mathcal{D}(x_i, z, P, LAT, E, T)$;
end
Weak-regressor SI with LMA and cost function computing;
for *kpi*,x_i,m_i ***in*** *KPI*,$\mathcal{X}$,$\mathcal{M}$ **do** # For every KPI, feature and model
    $\theta \leftarrow$LMA(d,x_i,m_i);
    $RMSE_{m_i} \leftarrow$RMSE(kpi,x_i,m_i,θ)+L2(θ);
end
Strong-regressor SI with competitive ensemble selection and full dataset training;
$m \leftarrow \arg\min_{m_i}(RMSE_{m_i})$;
$M \leftarrow$prod($a_i m$);
$\mathcal{M} \leftarrow M$;

2.4.5 Heterogeneous model definition

In previous subsections, it has been extensively explained the ensemble-based methodology to derive several multi-variable device models from single-variable modeling. In addition, it has been proposed a similar approach to have communication overhead profiling. In this subsection, the discussion is further extended on how these individual device behavioural models are combined together to estimate the performance of a heterogeneous system model $\hat{M}_{Het}$ from $\hat{M}_{CPU}$, $\hat{M}_{FPGA}$ and $\hat{M}_{GPU}$.

Heterogeneous KPIs are highly dependant on the selection of the partitioning techniques. The resulting set of task partitions must be offloaded to each accelerator by a scheduling mechanism. In [OHY+18], a number of heterogeneous models have been proposed to estimate the total execution time, energy consumption and communication time on contiguous partitions. In this section, this definition is extended to complement the previously-defined single-variable models. In [OHY+18], the latency is defined as the time execution sum of each task on both devices, given by a partition l_{part} from a finite number of partitions L. Furthermore, the communication latency for the IFM and OFM must be considered and added to the heterogeneous model.

$$L\hat{A}T_{Het}(X, \theta_{Het}, l_{part}) = \sum_{l=l_{part}} L\hat{A}T^{l}_{Device} + \sum_{L} L\hat{A}T^{l}_{Comm} \tag{2.12}$$

Where X is the set of all predictor variables and θ_{Het} is the resulting superset from the union of the parameters from the processing devices and communication models. Using this definition of latency, the total energy consumption is estimated considering idle power consumption using the following expression:

$$\hat{E}_{Het}(X, \theta_{Het}, l_{part}) = \sum_{l=l_{part}} L\hat{A}T^{l}_{Device} \cdot \hat{P}^{l}_{Device} + \left(L\hat{A}T_{Het} - \sum_{l=l_{part}} L\hat{A}T^{l}_{Device} \right) \cdot \hat{P}^{idle}_{Device} + \sum_{l=l_{part}} L\hat{A}T^{l}_{Comm} \cdot \hat{P}^{l}_{Comm} + \left(L\hat{A}T_{Het} - \sum_{l=l_{part}} L\hat{A}T^{l}_{Comm} \right) \cdot \hat{P}^{idle}_{Comm} \tag{2.13}$$

Where $\hat{P}^{l}_{Device}$ is the active power consumption of a give partition on the device. While $\hat{P}^{idle}_{Device}$ is the idle power consumption when the device is not doing any processing. During this idle time, the device may be transferring feature maps to the another device. Thus, it is important to consider the total system estimation with the communication latency, $L\hat{A}T^{l}_{Comm}$, and also the idle energy consumption of each communication link, This is done by multiplying by its respective idle active power consumption $\hat{P}^{idle}_{Comm}$.

2.5 Flydeling: randomly-excited system identification applied to KPI estimation

This section describes the proposed flydeling three-step methodology. A flyweight model (*flydel*) is characterized by a very low number of parameters, while its computational operations may require complex operations. This section covers the concepts of system stochastic excitation and validation. Furthermore, two techniques for parametric linear and non-linear SI algorithms are discussed, to be used in Section 2.4 for building performance models from ensembles.

Most modern compilers are optimized to avoid unnecessary memory transfers so as to avoid inter-device time overheads [CWV+14]. If some variable values remain unmodified over time, the communication latency is lowered and less energy is required to execute computation. This is the case of Nvidia® CuDNN tool, where static variables, such as weights of IFMs remain in device memory cache. Therefore, models trained over measurements on these optimized systems may perform well on static applications, where all weight kernel values are already loaded in memory. However, they are not able to generalize to non-static use-cases where all transfers do occur, for instance in applications with high memory transfers and low Computation-to-Communication Ratio (CCR) and low Bandwidth (BW). A low CCR means that the application spends more execution time in memory transfers than in realizing actual computing. This is the ratio of the data to compute divided by the data to transfer. This is usually a consequence of reduced capacity of data transfers in a band, known as BW. Such a bias toward static use-cases leads to derived models whose predictions generally become over-optimistic. To avoid bias, in SI, black-box systems are excited with diverse input sequences that aim at capturing all the modes of the system. Moreover, the model itself, and its input features, must be representative enough of the problem to fit this type of phenomena. The frequency response, in the domain of linear dynamical systems, defines the behaviour space of the output with respect to different temporal stimuli. Inspired from this principle, a three-step methodology is presented for modeling embedded processing performance, which goes from a random system excitation to a cross-validation test for model evaluation. By randomly-exciting sequence, it is referred to a stream of input-training data that is used to excite a system model so that stable and accurate estimates can be derived from the response to the excitation. The flydeling methodology from Figure 2.3 is decomposed as follows:

1. Dataset generation with a random input sequence that serves as a stochastic excitation input for training model parameters at a structural-level, and IFMs tensors, weight and bias values are initialized with a random input sequences at a data level.

2. *Flydel* System Identification (SI): Model parameters are estimated, using ensemble regression with both linear and non-linear models.

3. *Flydel* test using K-fold cross-validation for robustness evaluation, where K is a parameter of the methodology. For this validation step, $K = 10$.

The first step of the methodology consists in applying a stochastic excitation to the system for dataset generation at two different abstraction levels: at a data level and at a structural level. The data level excitation sequences cover the values of the IFMs, CNN weights and biases while the structural level samples stochastically the CNN hyperparameters such as the size of the IFMs, the number of kernel filters and the kernel filter sizes. In the rest of this section, SI procedures are discussed. They will be used for model validation in Section 2.4.

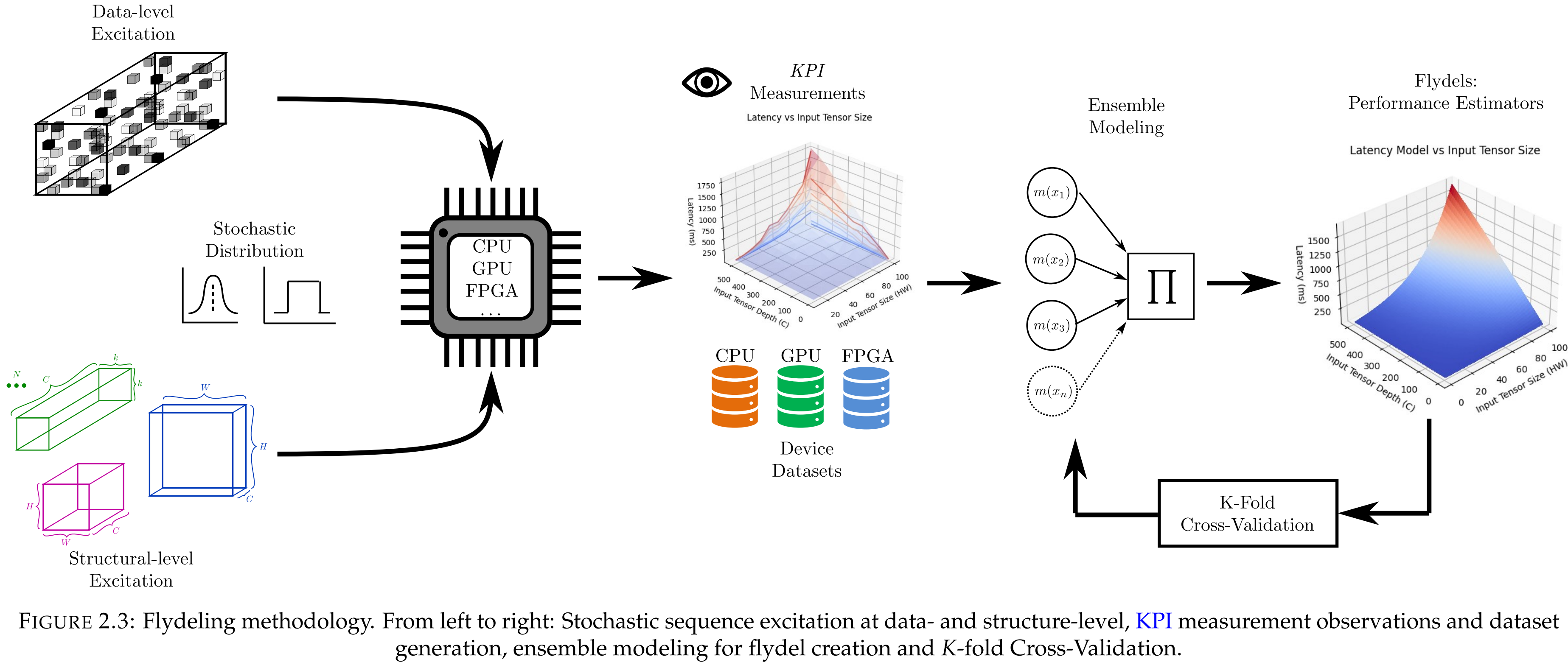


FIGURE 2.3: Flydeling methodology. From left to right: Stochastic sequence excitation at data- and structure-level, KPI measurement observations and dataset generation, ensemble modeling for flydel creation and K-fold Cross-Validation.

2.5.1 System identification

Following the dataset generation from a randomly excited system, the second step of the model estimation is executed. SI is a technique that was originally used in control theory to estimate the parameters of a proposed mathematical expression. These mathematical expressions are known as models [JP70]. The models are built from measurement-based dataset samples, as the solutions to an inverse problem. The SI method applied to KPI prediction can be defined as a 4-tuple, $\mathcal{S} = \langle \mathcal{M}, \mathcal{X}, \Theta, \mathcal{D} \rangle$. Where $\mathcal{M}$ is a set of multiple system performance models, each adapted to fit a chosen KPI. Each model, $M \in \mathcal{M}$, is a function that maps input features to the given KPI $M : \mathbb{R}^{|X|} \rightarrow \mathbb{R}$. Models of KPIs are chosen to be well-defined smooth differentiable continuous functions. The application activity set $\mathcal{X}$ contains the independent variables, $\mathcal{X} = \{H_I, W_I, C_I, k, N\}$, resulting from the random sequence defined in the previous subsection at a structural-level. For this purpose, application-specific sets of descriptor variables from the different CNN operation configurations are used. The set Θ is composed of all coefficients or *parameters* trained using the identification techniques described in this section. Depending on the model, these parameters can be obtained in one-shot, analytically or iteratively by numerical approximation. The previously defined dataset $\mathcal{D}$ contains the measured outputs of the system to be identified, stochastically-excited at a data level. Section 2.6.1 describes how this dataset is empirically generated using the excitation methodology from previous subsection.

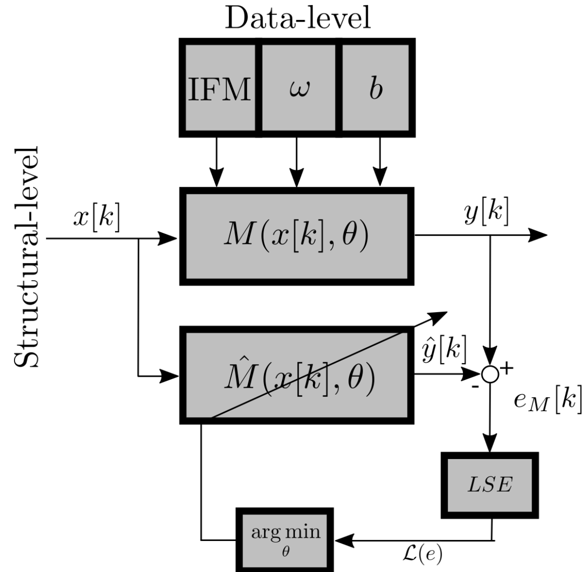


FIGURE 2.4: General SI block diagram. M is the system to be identified by model $\hat{M}$. Values of IFMs, weights (ω) and biases (b) are randomized at data-level. The CNN activity features ($x[k]$) are randomized at structural-level.

Figure 2.4 shows a general block diagram for system identification. The system M and estimated model $\hat{M} \in \mathcal{M}$ are fed with the input $x[k] \in \mathcal{X}$, for the discrete case, with $k = 1, 2, 3, ..., K$ samples from previous section. As a response, both the system and model generate a set of K samples $y[k] \in \mathcal{D}$ and $\hat{y}[k] \in \mathcal{D}$, respectively. The information is

extracted in the form of the parameters estimated or updated after comparing the system against the model. This comparison is done usually by computing the difference or errors from each instance, or $e_M[k] = y[k] - \hat{y}[k]$. These error residuals $e_M[k]$ are used to estimate how precise the model is from a LSE perspective.

Models are categorized into linear and non-linear. Depending on whether the number of parameters is fixed or can vary during optimization, models are also divided into parametric and non-parametric, respectively. Depending on the chosen optimization solution, models can be trained using analytical or numerical methods [JP70]. Analytical methods focus on obtaining an exact solution on a well-defined mathematical expression or problem. However, in most cases, these solutions are time consuming or non-trivial to find. In those cases, numerical methods can approach to an acceptable solution by evaluating a set of relatively simple operations. In this work, linear and non-linear parametric system identification are used with the objective to train models from small training sets and from maintaining the number of parameters as low as possible, as explained in Section 2.6. Both solutions are also described later in this Section.

2.5.2 Stochastic sequence excitation

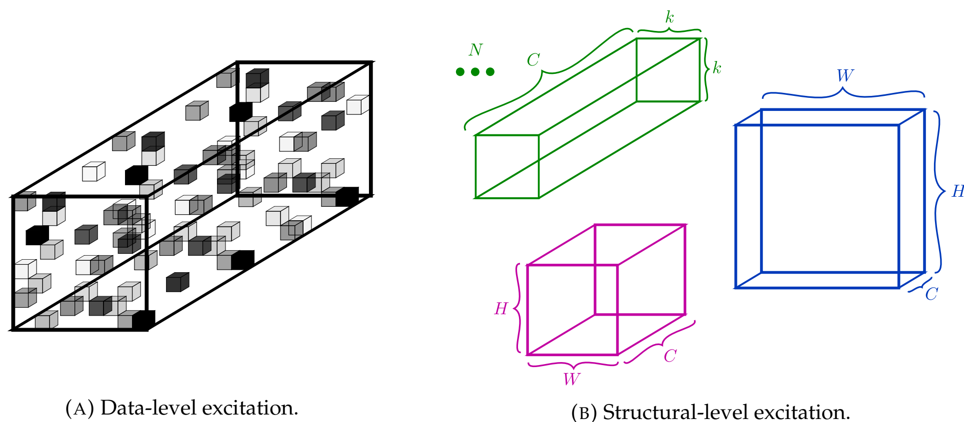
Response From Excitation (RFE) measurements are required before proceeding with SI. The excitation input sequences are split into two abstraction levels: *structural-level* (coarse-grained) and *data-level* (fine-grained). The structural-level focuses in the configuration of the CNN FMs shapes as introduced in Section 2.4. Each layer on a CNN have a different structural configuration with a number of MAC operations and memory accesses, that directly impact the deployment performance. Similarly, at a data-level, the individual element values of the FMs influence the efficiency of both computation and communication. This is because the compiler optimize for data reuse that directly depend on the intrinsic patterns of memory accesses and reduced computation. In this Subsection, both abstraction levels are discussed.

When building black-box system models for design space exploration, a designer aims at interpolating and extrapolating measured system configurations to unknown system configurations. Hence, a system must have a rich excitation input, representative of a wide variety of configurations for a given application. In SI, this is achieved by using a random sequence, that follows a probability distribution representative of real data in order to extract the output response [JP70]. A true random sequence cannot be generated by a deterministic system, but a designer can create a pseudo-random sequence with wide enough periodicity for exciting the application. Pseudo-random sequence generation can be efficiently implemented in software by calling random number generators on either CPU or GPU processing elements to excite the data-level values. Thus, in the case of a CNN excitation, and without taking hypothesis on the distribution of data and weights, it is possible to define a random distribution input tensor using random number generation software tools using, e.g., *torch.rand* function on PyTorch [PGC+17] with CUDA support for GPU programming. However, the FPGA case is more complex. Indeed, random generation tools such as the VHDL *math_real* package are not synthesizable, making

them usable for functionality testbenches but not for empirical measurements. For this reason, a hardware implementation of a Pseudo-Random Binary Sequence (PRBS) is needed to excite the system and build the desired models. Many techniques have been proposed to generate on FPGA randomly-distributed sequences [BGCO18]. In [BGCO18], Linear Feedback Shift-Registers (LFSRs) implementations on FPGAs are studied as normal random sequence generators. From an implementation perspective, one of the simplest techniques to achieve these sequences is to instantiate a cascade of delays and select their outputs through a bit mask. The input of the cascade is fed back using a user-defined logical function, combining the selected bits from the bit mask. The output of the logical function is fed back as input of the first delay block. In the Z-transform domain, the z^{-1} blocks represent the discrete delay of a single sample with a sampling frequency f_s. The length of this delay cascade dictates the period of the LFSR by $(2^N - 1) \cdot T_s$, where N is the length of the cascade and T_s is the sampling period. Depending on the application and the number of $k = 1, 2, 3, ..., K$ samples required: N, the bit mask and the combinational logic function can be chosen in a deterministic way [BGCO18]. As an example, if a designer needs to create a random sequence with 1K elements to generate an input tensor with dimensions $32 \times 32 \times 1$, a delay length $N = 11$ is selected to ensure a Pseudo-Random Binary Sequence (PRBS) with the desired periodicity. The periodicity of the sequence depends on the *AND* bit mask. For this work, the $9th$ and $11th$ delay elements are chosen and a logic function of the *XOR* [BGCO18]. Therefore, the input of the first delay would be $x = z^{-9}x \oplus z^{-11}x$, being $z^{-11}x$ the output of the cascade and the sequence used to excite the systems. Depending on the initial conditions of the N delayed samples of x, it is possible to add a random seed for variety and reproducibility purposes. The creation of a randomized input allows us to obtain a measure dataset of the output of each device and further study their response. This is especially important on custom logic devices like FPGAs and ASICs. Where, from an electronics perspective of the hardware, the dynamic power modeling highly depends on the dynamic switching frequency of the transistors and on their hardware technology [NLPH21]. Figure 2.5 shows the two abstraction levels at a different granularity. Subfigure 2.5a depicts the internal values of the IFM values, weight and biases values, while Subfigure 2.5b shows the tensor configuration shapes. Following subsections discusses both these excitation levels.

2.5.2.1 Structural-level stochastic excitation:

The set $\mathcal{X}$ is defined as the set that contains all independent variables or *descriptor variables* that feed the models as input, also commonly known as *features*. The structural configuration of the CNN layer defines these features. Features, also known as predictor variables, are referred as *application activity* [PMD+17]. Application activity is an abstraction of the pressure put by an application on a platform. It can take many forms but aims to be the minimal application-side information required to compute, with fidelity [JIP10] or accuracy, the value of a KPI, or of a set of KPIs. The right selection of the descriptor variables is one of the most critical tasks of the SI problem at a coarse-grain. Depending on the nature of the system, these variables can be easily chosen, while on other cases,



(A) Data-level excitation.

(B) Structural-level excitation.

FIGURE 2.5: Stochastic excitation abstraction levels.

a deeper correlation understanding is required between dependent and independent variables. In this section, application-specific sets of descriptor variables from the different CNN operation configurations are used. These CNN structural properties are shown in literature to be well related to KPIs due to the embarrassingly concurrent nature of the applications [VMFBCGRV20], and to the large parallelism of hardware substrates that make strongly non-smooth behaviors such as timing anomalies [CHO12] non dominating in practice.

The flydeling procedure includes a model validation step to determine whether the application activity contains features that are sufficiently representative for the required modeling task. In particular, the features should be suitable to model the application workload and to compute target KPIs for a given target platform (the hardware together with its compilation and synthesis setup) through continuous functions. In this chapter, each hardware architecture is modeled in isolation. Cross-architecture modeling that would generalize a unique cross-platform model is kept for future work. It would require additional features, representing architecture e.g. with a Model of Architecture (MoA) [PMD+17]. The chosen CNN activity features are the dimension of the IFM, $H_I \times W_I \times C_I$; and the dimension of the kernel filters, $k \times N$. This choice is covered in details in Section 2.4. The configuration space is randomly sampled by stochastically selecting from each feature space.

2.5.2.2 Data-level stochastic excitation:

At a finer-grain, a dataset $\mathcal{D}$ from the random structural configuration samples that contains the sampled outputs of each KPI with each input variable from $\mathcal{X}$ is built. The dataset $\mathcal{D}$ comprises information on the system response. In other words, it contains the values of the IFMs, OFMs, weights and the obtained KPI measurements. Nevertheless, since these samples are used to train, test a validate the proposed models from a Least Squares Error (LSE) perspective, it is required that these dataset inputs are randomly excited at data-level values. Explained differently, the input tensor values, weights and biases shall be normally randomized to excite the processing elements for the model to

be generalizable. As previously explained in this Section, this is especially important for custom logic design.

Algorithm 2 shows the steps for the stochastic input generation on a device platform at both structure- and data-level. The algorithm take as inputs the desired KPIs to be modelled with a given set of features ($\mathcal{D}$). A stochastic distribution is generated to excite the system at two levels: The first for-loop, the structural-level, modifies the shape or dimensions of the IFM tensor by changing the features of set $\mathcal{X}$ which is uniformly-distributed over the CNN configurations. The second for-loop, the data-level, varies the values of the IFM elements, weights and biases for K samples with a normally-distributed sequence. Most trained CNN models follow a normal distribution for kernel weight and bias values, as shown in Subsection 2.6.2. The innermost loop takes a measurement of each KPI and averages it by the total number of K samples.

Algorithm 2: Random excitation methodology for dataset generation

```
Result: Dataset 𝒟
Input: Features 𝒳 and Key Performance Indicator (KPI)
Output: Dataset 𝒟 matrix from uniformly-distributed sample on 𝒳
# 𝒟 Dataset generation;
# Structural-level stochastic excitation;
𝒳←uniform(H_I, W_I, C, k, N);
for x_i in 𝒳 do # For every feature
    for i ← 1 to K do # For K samples
        # Data-level stochastic excitation;
        IFM←uniform(H_I, W_I, C);
        ω ←normal(k, C, N);
        b←normal(N);
        for kpi in KPI do # For every KPI
            average_kpi←average_kpi+k_measure;
        end
        average_kpi←average_kpi/K;
        𝒟[i] ←tuple(x_i,average_kpi);
    end
end
```

2.5.3 Linear parametric SI

In control theory systems, the inputs of the system must have a rich frequency response to ensure an output response representative of the underlying system. Usually, this is done by supplying a stochastic input signal or a random sequence. SISO linear parametric model of 0-order (no delay) can be solved by analytically finding optimal parameter vector θ^* from $\theta^* = U^+V.V$ is the output vector of K samples of the linear model and U is the input matrix of the inputs. These values are obtained from the residual vector $E = V - U\theta$.

U^+ denotes the Moore-Penrose pseudo-inverse defined as $U^+ = (U^TU)^{-1}U^T$, where the number of (linearly independent) samples is greater than the order of the linear model. In other words, the model must not be over-parameterized, keeping K greater than the size

of the vector θ, which is a common practice for SI and regression applications. For higher order linear models with temporal dependencies, the U matrix not only includes current values of the input, but also previous or delayed information of the output samples of $y[k]$. In this case, the maximum delay of $y[k]$ sample determines the *order* of the SI model. Furthermore, if SI is extended to the Multiple Input Single Output (MISO) case, then the U matrix also includes the samples of the different inputs $x_i[k]$. Where $x_i[k]$ is the k-th sample of the i-th input. For the use case of this work, SI is applied on a MISO setup without temporal dependency (Section 2.4). Nevertheless, in the more general case, dynamical dependencies may exist and flydeling may require the use of feature sampling over time. A time-dependency test for a given application before proceeding with model proposal for the ensemble model is recommended [Kee11].

2.5.4 Non-linear parametric SI

In contrast to the previous method, Levenberg-Marquardt Algorithm (LMA) is a numerical solution for the Least Squares Minimization (LSM) problem that can be efficiently applied on continuous non-linear differentiable estimation models [Lev44, Mar63]. The algorithm convergence is sensitive to the number of parameters, normally suitable to models with only tenths of parameters. As demonstrated in Section 2.6.2, the use-case is ideal for the utilization of this algorithm. LMA is often described as a generalization combining both gradient descent and Gauss-Newton. The $i+1$ iteration of traditional gradient descent follows the path towards the direction of minimum error $\lambda \nabla f(y, x, \theta_i)$. λ regulates the step size of each iteration. The function f is to be minimized given by the LSE criterion, usually L2 norm $f(E) = \frac{1}{2}||E||^2$.

Levenberg [Lev44] proposed a combination of both methods by approximating the curvature using the Hessian H into gradient descent method. However, when λ is larger than the Hessian, the curvature of the residual function is not taken into consideration. Marquadt [Mar63] modified the update rule to consider large steps in low curvature and small steps in steep curvature $\theta_{i+1} = \theta_i - (H + \lambda diag(H))^{-1} \nabla f(y, x, \theta_i)$. This is known as the update rule for LMA. This is one of the most used heuristic-based algorithms for ML, since it has been demonstrated to perform well on practice. Furthermore, it is a suitable method for small parametric models with few parameters and samples [Ran04] compared to other model approximation techniques like kernel-based techniques, such as SVMs. SVMs for regression may require long training time on big measurements dataset and may be difficult to tune while choosing a right kernel function [WH05]. Despite the fact that these models usually perform better for more complex models, it is shown that simple lightweight model selection can be suitable for some problems. All these observations fit accordingly to the case of study as described in the following sections.

2.5.5 Model validation

The third and last step of the methodology is the model precision evaluation using cross-validation. The main objective of this validation is to test that the estimated set of

model parameters (Θ^*) and an error-based cost function ($\mathcal{C}$), treated as random-variables, follow a probability distribution with relatively small variance, as a result of a random evaluation process. To obtain the random-variable distribution, the estimated model must be evaluated over different subsets of the data set $\mathcal{D}$. In ML context, this is usually done by splitting the dataset in folds and by training over some fold, while evaluating over the remaining folds. This procedure is known as *k-fold cross-validation*. This evaluation method avoids over-fitting over the whole dataset. Furthermore, the resulting parameters distribution can then be analyzed as the required random-variable. This distribution can be visually identified by analyzing multiple graphical illustrations of the obtained Probability Density Function (PDF) or histogram. A vast number of techniques like: histogram plotting, box plots, violin plots, probability plots, Q-Q plots, P-P plots, just to mention some, offer insight into the nature of the random variable distribution. However, statistical tests can give us a reliable information on the parameter Gaussian nature of the obtained estimates in some cases these plots are evidence enough [GZ12]. It is demonstrated in Section 2.6.2, that this is the case for flydeling, as applied to given use cases. It is thus proven that indeed the parameter variations are result of a Gaussian process. Finally, to determine if the parameters values of obtained models are suitable, an acceptance criterion or *OK/KO* test is defined. In the validation stage, it is proposed an acceptance criterion based on the statistical second central moment, the variance σ^2. A threshold inferior to 10% of the normalized standard deviation is selected for each parameter distribution from the k-fold cross-validation. Algorithm 3 sums up the integration of these concepts from ML, statistics and automation for a validation stage.

2.6 Flydeling evaluation and results

In this section, the experimental setup is described for testing flydeling on real CNNs and platforms, from the dataset generation $\mathcal{D}$ to estimation of the target KPIs. In Section 2.4, it is covered and shown examples of the individual performance for a particular device. At the end, in Subsection 2.6.2, the results of the homogeneous platform measurements for different hardware modules are discussed.

2.6.1 Dataset generation

Dataset $\mathcal{D}$ is composed from a set of measurements over each KPI, each activity feature on $\mathcal{X}$ and for each device d, or equivalently $\mathbb{R}^{|\mathcal{M}|\times|\mathcal{X}|\times K\times|d|}$, with K samples and where $|\cdot|$ is the operator cardinality for a given set. In the given use-case, the heterogeneous platform has 3 devices; the CPU, GPU and FPGA. Therefore, $|d| = 3$. These data points are used by the SI regressors to obtain the CPU, GPU and FPGA device KPIs models describing the CNN processing block. As depicted in Figure 2.6, for the experimental setup, a custom heterogeneous platform is employed incorporating an Nvidia® Jetson TX2® embedded CPU-GPU (green), and an Intel® Cyclone 10 GX FPGA (blue) using a dataflow Direct Hardware Mapping (DHM) technique [APS+17]. Interconnection is established by a Peripheral Component Interconnect Express (PCIe) Gen2 x4 (5 GT/s)

Algorithm 3: Methodology for model validation

```
Result: Parameter distributions and validation
Input: KPI models 𝓜 and dataset 𝓓
Output: Estimated parameter Θ, metric distributions and validation decision val
# 10-fold Cross-validation with shuffled dataset;
for i ← 1 to iterations do # Several iterations
    𝓓←shuffle(𝓓);
    for n ← 1 to 10 do # Split dataset in training and validation folds with LMA
        𝓓_train, 𝓓_val←split(𝓓);
        Θ_n ←LMA(𝓓_val,𝓧,𝓜);
        RMSE_𝓜 ←RMSE(kpi,𝓧,M,Θ_n)+L2(Θ_n);
    end
    dist_Θ ←PDF(Θ);
    σ_val ←NORM_STD(dist_Θ);
end
# Parameters distributions on test dataset;
for dev in Devices do # For every device
    for kpi in KPI do # For every KPI model
        dist_RMSE_𝓜 ←PDF(RMSE_𝓜);
        for θ in Θ do # For every parameter
            dist_θ ←PDF(θ);
            σ_θ ←NORM_STD(dist_θ);
            # Acceptance criterion (OK/KO Test);
            if σ_θ ≤ 0.1σ_val then # Acceptance OK criterion
                # Accept parameter estimation;
                val ←OK(θ);
            else
                # Reject parameter estimation;
                val ←KO(θ);
            end
        end
    end
end
```

communication channel (gray). The communication link between CPU-GPU lie on the same System-on-Chip (SoC) die, therefore they share a common external memory. More details on the prototype experimental setup are described in Appendix B.

Table 2.3 shows the specifications of this particular system. Since this platform incorporates a processing element with embedded custom logic, it is important to be specially careful with the limited design resources. The logic and memory elements utilization will be inferred by the selected models on the FPGA side. Since this platform incorporates a processing element with embedded custom logic, it is important to be especially careful with the limited design resources. The logic elements and memory utilization will be inferred by the selected models on the FPGA side. The aim is to study in future work how this flydeling can be help design partitioning at application-level for heterogeneous platforms.

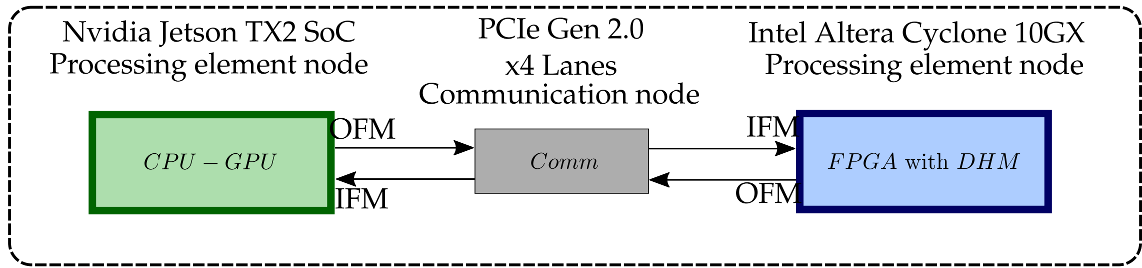


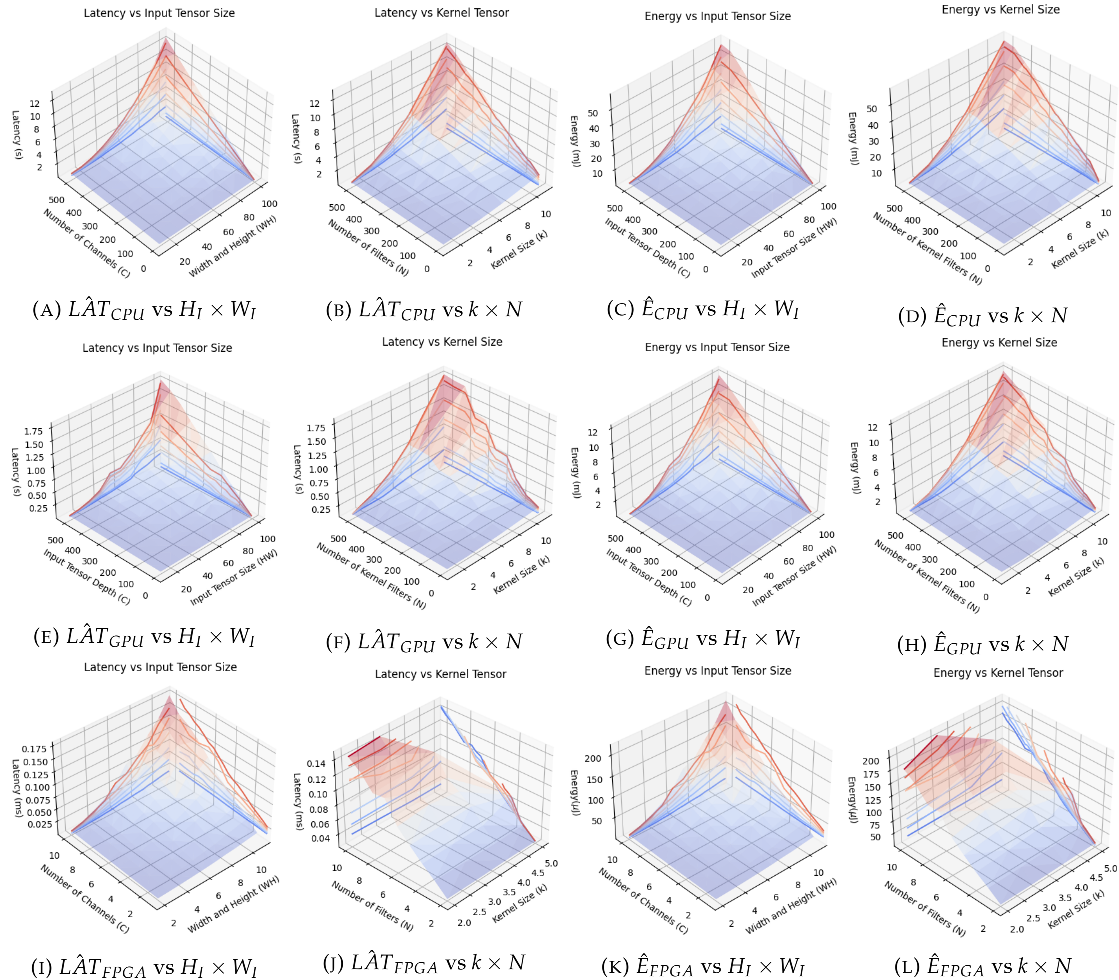
FIGURE 2.6: Architecture model as for the heterogeneous setup with three computing elements: A Nvidia® Jetson TX2® CPU-GPU (green) SoC and an Intel® Cyclone10GX FPGA (blue) interconnected through x4 lanes of a PCIe Gen2 link (gray). Each processing element and communication node has an IFM and a OFM associated to it required for metric performance measuring on each device. These measurements are used for the flydel data generation and estimation.

Device	Frequency	Memory		Logic Resources	
Nvidia® Jetson TX2 ® CPU (Denver + Arm® A57®)	2 GHz	LPDDR4	8 GB	Denver cores	2
		eMMC	32 GB	A57® cores	4
Nvidia® Jetson TX2® GPU	1300 MHz	Registers	2 MB	SMs	2
		L1	96 KB	SMPs	8
		L2	512 KB	CUDA cores	256
		LPDDR4	8 GB	Shaders	256
Intel® Cyclone 10 GX® FPGA	50 MHz	ALM Registers	321 K	LEs	220 K
		M20K	11740 Kb	ALMs	80 K
		MLAB	1690 Kb	DSPs	192
		DDR3	1 Gb	Mult	384

TABLE 2.3: Embedded heterogeneous platform hardware specifications.

2.6.1.1 Power

The first KPI measurements obtained for the dataset is the power efficiency in Watts. On the Jetson TX2® Module-on-Chip (MoC), a Tegra X2® SoC is incorporated, which integrates a multi-channel power monitor (INA3221) used to obtain the measurements of power dissipation on the multi-core ARM® CPU and the Pascal-architecture GPU. The 3-channel power monitor is configured with a 64-sampling average filter. As described in Section 2.5, SI requires an varied application activity to obtain a system response that will be used by the ensemble to derive a behavioural model describing the energy KPI. A random uniformly-distributed sequence is generated for the samples, but for the weight and bias values a normal distribution is selected. This also helps to preserve application generalization that the compiler may infer by varying the values. This is the case for some tools, like Nvidia® CuDNN toolbox, where the compiler avoid redundant data transfers, but its optimized for a specific data distribution, which may not be exploitable on other CNN applications. This excitation technique from dynamic SI is incorporated for the proposed use case. In Figure 2.8, the embedded GPU average power KPI response in mW is used to estimate the energy by keeping the input image size constant and



(A) $\hat{LAT}_{CPU}$ vs $H_I \times W_I$ (B) $\hat{LAT}_{CPU}$ vs $k \times N$ (C) $\hat{E}_{CPU}$ vs $H_I \times W_I$ (D) $\hat{E}_{CPU}$ vs $k \times N$

(E) $\hat{LAT}_{GPU}$ vs $H_I \times W_I$ (F) $\hat{LAT}_{GPU}$ vs $k \times N$ (G) $\hat{E}_{GPU}$ vs $H_I \times W_I$ (H) $\hat{E}_{GPU}$ vs $k \times N$

(I) $\hat{LAT}_{FPGA}$ vs $H_I \times W_I$ (J) $\hat{LAT}_{FPGA}$ vs $k \times N$ (K) $\hat{E}_{FPGA}$ vs $H_I \times W_I$ (L) $\hat{E}_{FPGA}$ vs $k \times N$

FIGURE 2.7: Average generated subsampled dataset $\mathcal{D}$ on each embedded CPU (top row), GPU (middle row) and FPGA (bottom row) platform by keeping constant $|\mathcal{X}| - 2$ features for 3D visualization. Each Subfigures row shows latency and energy KPIs on a single device. The contour curves represent the relation between a given KPI and a single feature variable. On Subfigures 2.7a, 2.7e, 2.7c and 2.7g; features $k = 11$ and $N = 512$ are kept constant. On Subfigures 2.7b, 2.7f, 2.7d and 2.7h; features $H_I = 100$, $W_I = 100$ and $C = 512$ are kept constant. On Subfigures 2.7i and 2.7k, features $k = 5$ and $N = 10$ are constants. Finally, in Subfigures 2.7j and 2.7l, $H_I = 12$, $W_I = 12$ and $C = 10$ are kept constant.

sweeping over different operations and number of input channels. This average filtering in time domain works as a low-pass filter on frequency domain, that is the reason the measurements show a rounded shape between experiment iterations. The Sigma-Delta Digital-Analog Converter (ADC) used on the Integrated Circuit (IC) uses a 500KHz sampling rate following the specifications from manufacturer (Texas Instruments®). For the measurements, data is retrieved by passing through the OS which introduces an overhead on the number of samples taken per experiment iteration, however this is the reason multiple iterations are executed on a single computing task, Conv1 $\times$ 1 for example, is averaged over 5000 samples. One program thread is executed on a dedicated CPU core to read from the GPU power rail. Since shallow layers have less filter kernels, they take less time to be executed in this example. Therefore, high frequency sampling variations are taken into consideration for the average accumulator. The Jetson TX2® includes multiple

performance modes that may vary the CPU core utilization and the operating frequency of the embedded GPU to save energy. The power mode is set to the Max-N with a GPU frequency of 1.3GHz and allowing the full use of the multi-core processor for the CPU dataset generation. Note that multiple models,one per mode, could be generated to cover the different cases. The software tools used to generate the CNN on the GPU was PyTorch [PGC+17] version 1.0 with CUDA 8.0® support for the Jetson TX2®. While no state-of-the-art CNN architecture is selected for the training, the activity features take common values found on CNN applications from popular model zoos. Thus the obtained models are likely to extrapolate to newer practical cases of CNN architectures. On the CPU-GPU side, the IFM, weights and bias of the CNN are generated using the PyTorch function *rand*, which generates a random number from a uniform distribution and initializes the tensors, while *randn* generates a random normal distribution for kernel weights.

On the FPGA side, the Power Estimation tool® from Intel Quartus Pro Edition® version 17.1 is used targeting multiple convolutional task operations on the Intel® Cyclone 10 GX FPGA. The FPGA CNN synthesis design is based on the DHM technique [APS+17]. DHM maps directly the function on hardware, therefore its power varies rapidly with the number of processing elements mapped on the device. Using Quartus Pro Edition® it is also possible to generate a dataset based on the resulting resource utilization after synthesis considering the LFSR implementation of Section 2.5. In previous work, it has been demonstrated that DHM technique can outperform SIMD execution [CHPB21] on the GPU at the cost of a high resource utilization on the FPGA. This effect increases with the number of kernel filters on a fixed IFM. Nonetheless, this is only true as long as the design can fit into the FPGA device. Being the Cyclone10GX an embedded FPGA, this limitation is quickly met for a fixed kernel filter size and feature input map, for example, at 64 filters for Conv7 $\times$ 7 with a IFM of size 64$\times$64. This only allows the mapping of small functions for deeper modules or layers in a typical CNN application. This highlights the importance and complexity of mobile CNNs deployment on embedded processors.

2.6.1.2 Latency

Using a similar methodology as in the power measurements, a timestamp was included on the execution of every iteration for all convolution functions in Figure 2.8. This way, it was possible to track the execution time of a specific task. Similarly, on the FPGA side the Timer Analyzer tool® on the Quartus Pro Edition® tool is used for FPGA latency estimation. The latency is dictated by the critical path on the pipelined execution on the DHM of the last datum to be processed [APS+17]. In some cases, not only the FPGA clearly outperforms the GPU, but the latency remains almost constant at the expense of more resources. This is because the compiler tries to reduce the critical path latency for synthesis, preserving the given operational frequency of around 50MHz for the proposed platform. This means, that the more intensive the computational workload is, bigger is also the difference in latency performance between FPGA and GPU devices. Again, this claim is true as long as the design can be mapped on the device and the critical path is constant.

2.6.1.3 Energy

For this case, energy in Joules is calculated by integrating power over the processing window of one image $E(\cdot) = P(\cdot) \times LAT(\cdot)$. Indeed, the current setup does not pipeline computation over multiple images. The energy efficiency between different substrates can be increased either by increasing the power efficiency or by reducing latency. The energy efficiency of a multi-device setup will also depend on communication latency and communication power, since idling power consumption may be considerable for some use cases. In Figure 2.8, the solid lines represents the measurements of the element product of both power and latency KPIs to determine the energy consumption.

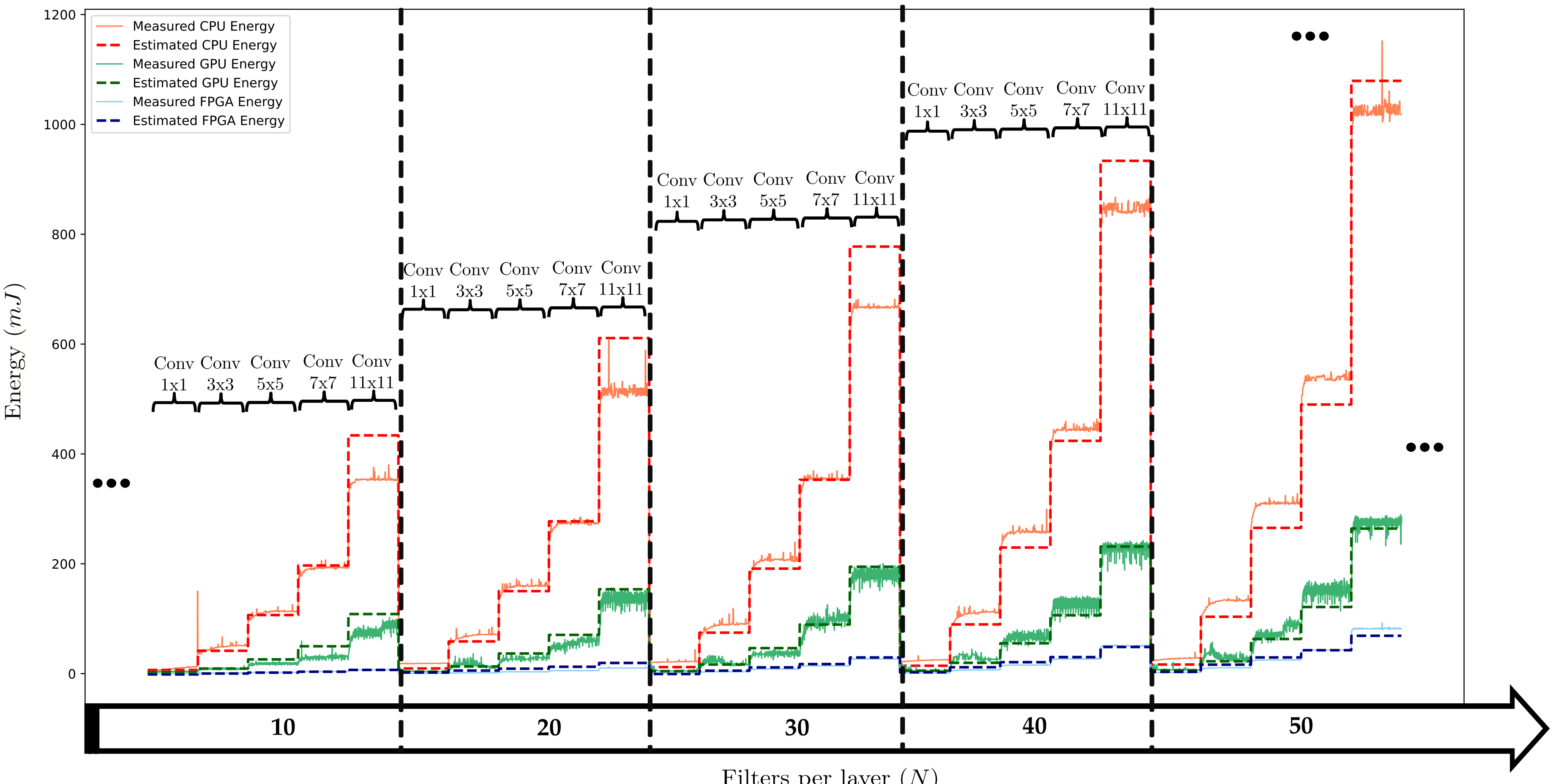


FIGURE 2.8: Example of energy consumption measurements (in mJ) over multiple experiment iterations on the ARM® CPU-cores (solid red), the Nvidia® Pascal GPU architecture (solid green) of the Jetson TX2® SoC and the Intel® Cyclone 10GX® (solid blue). Common convolution filter sizes are iterated 1000 times on a fixed size input tensor ($32 \times 32 \times 3$ CIFAR-like input image for this example [Kri09]) with a random uniform distribution for the input and a normal distribution for the kernel values. Number of filters N increases progressively over time and the resulting observations are averaged.

2.6.1.4 Throughput

The last generic metric considered in this work for all the devices is the throughput in GB/s. Minimum throughput is often the main characteristic put forward for an accelerator. It can also be expressed as Frames per Second (FPS) in computer vision applications. Many factors affect the throughput on a GPUs, for example from a hardware perspective, the number of cores that can be simultaneously launched, or the memory utilization on the device. In addition, from a computation perspective, the data transfers and redundancy in memory transfers can also impact the throughput. As more data accesses and operations are scheduled for the GPU, the throughput is reduced since the communication to host remains constant while the latency increases. On the FPGA device, the throughput is also correlated to the latency and the maximal operation frequency dictated by the critical path. When using DHM, the throughput also depends on the number of data flows in the last layer of the fused layer. Therefore, the throughput can be obtained as $T = H_I W_I C_I f_{max} \times N_{flows}$; where f_{max} is the maximum operating frequency and N_{flows} the number of data buses at the last layer on the FPGA. For this application implementation, N_{flows} is the same as kernel filters (N). At the other hand, in the best scenario, f_{max} is the operating frequency of the FPGA, 50 MHz in this case. As presented in [APS+17], each dataflow can process one pixel with all its channels for the IFM. However, since both the critical path and f_{max} are obtained after synthesis, modeling them can help preemptively determine them at design stage. This complexity of KPI inference is a strong motivation for flydeling, as CNN code generation is now very advanced, making it possible to prototype and measure KPIs quite easily, but performance is in general difficult to predict for a given hardware setup.

2.6.1.5 Resources

Custom logic design requires a careful handling of resource utilization. Because of this resource utilization is highly dependent on the algorithmic implementation, it is helpful for the designer to estimate the required logic elements for a particular solution. Resource estimation is specially crucial for embedded design using DHM technique [APS+17]. Since every weight and operation is directly mapped on hardware in a dataflow fashion, the available resources can be easily exhausted as more computation is required on the FPGA platform. Although different resource KPIs can be chosen for different applications, in this study, it has been identified four relevant KPIs on the Intel® Cyclone 10 GX FPGA platform: the ALUTs, ALMs, LABs and M20K memory blocks. Similarly to previous KPIs in this subsection, it is included in dataset $\mathcal{D}$ the number of required elements per configuration sample from the stochastic excitation on Section 2.5. The proposed models $\hat{R}$ are defined in the real domain, $\hat{R} \in \mathcal{M}$ with $\mathcal{M} : \mathbb{R}^4 \rightarrow \mathbb{R}$, so they are smooth and differentiable to use LMA. Nevertheless, the number of logic and memory elements are positive integers, thus the greater closest integer is taken; changing the codomain from $\hat{R} : \mathbb{R}^4 \rightarrow \mathbb{R}$ to $\hat{R} : \mathbb{R}^4 \rightarrow \mathbb{N}$.

To alleviate graphic visualization for the multiple KPIs, the number of varying activity features is reduced. This subsampled dataset can visually help to identify some hidden relationships between features and KPIs. Figure 2.7 shows a randomly chosen subset of convolutional operations features on the CPU-GPU-FPGA multi-device target. The Subfigures 2.7a, 2.7e and 2.7i show the latency; while 2.7c, 2.7g, and 2.7k show the energy on the CPU, GPU and FPGA, respectively; by keeping the kernel filter size (k) and the number of filters (N) fixed. Similarly, Subfigures 2.7b, 2.7b and 2.7j show the latency; while 2.7d, 2.7h and 2.7l show the energy consumption by complementary fixing the input tensor size ($H_I \times W_I \times C$). This reduced 3D visualization, shows the variations of KPIs with respect to the two two features, the input tensor size ($H \times W$) and the number of input tensor channels (C). It can be observed that these relationship do not show strong discontinuities. As an example, it is possible to visually identify a linear relation between the latency and the number of channel by analyzing the projections of contour curves in Subfigure 2.7a for the CPU device. Similarly, a quadratic relation is likely (and logical) using the input tensor size. While this gives us some insight for explainability and may motivate a manual selection for these models, it is desired that the ensemble could automatically select the model with statistically support of the aggregation criterion from Section 2.4.

2.6.2 Resulting experimental models and evaluation

In this subsection, it is presented the single-feature models, or weak regressors, selected from the competitive ensemble for every studied KPI on each computation node. In Table 2.4, information on the chosen models that were used on the multiple-feature model $\hat{M}(H_I, W_I, C_I, k, N)$ is shown. As discussed, the RMSE metric is chosen for the given cost function to estimate the models performance. However, the RMSE depends on the scale or the units of each KPI. Therefore, Normalized RMSEs (NRMSEs) are compared, dividing RMSEs by the range of the full dataset.

$$\mathcal{L}_{\hat{M}}^{NRMSE} = \frac{\mathcal{L}_{\hat{M}}^{RMSE}}{y_{max} - y_{min}} \tag{2.14}$$

Finally, these individual behavioural models are trained on a subsampled dataset by keeping constant the other application activity features on each weak regressor. This sub-sampled training dataset is a slice of the full dataset. Therefore, they need to be retrained with the full training dataset to adjust the parameters of the strong regressor. This is the case for each KPI on each processing device.

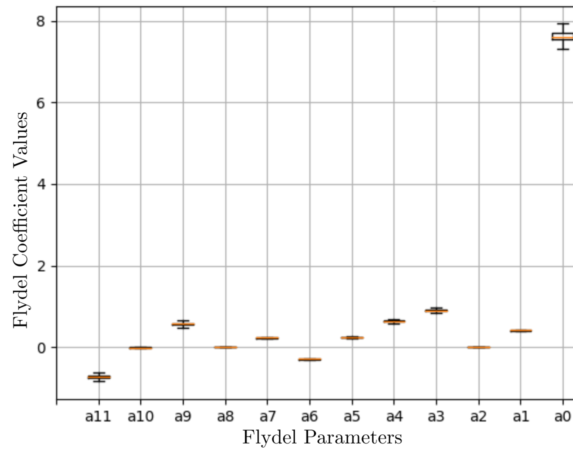
In Figure 2.8, the light-colored solid curves show real-time energy measurements for different CNN layer configurations on the considered GPU. These KPI measurements are directly obtained from a stochastic excitation of the platform which generates relatively small variations in the instrumental readings. The test set measurements are used to evaluate the obtained analytical models from the ensemble in real-time. The red dashed curve is the estimated energy consumption obtained from the model $\hat{E}_{CPU}$, the green one for $\hat{E}_{GPU}$ and the blue one for $\hat{E}_{FPGA}$. While the FPGA solution with DHM is the

			Features					
Node	KPIs		$H_I \times W_I$	C_I	k	N	$\#\theta^*$	NRMSE
CPU	$\hat{P}$		Quad	Log	Poly	Log	11	12.5%
	$\hat{LAT}$		Poly	Poly	Poly	Lin	14	7.0%
	$\hat{E}$		Quad	Lin	Poly	Quad	12	7.7%
	$\hat{T}$		Poly	Reci	Reci	Poly	12	3.9%
GPU	$\hat{P}$		Poly	Log	Quad	Quad	13	12.2%
	$\hat{LAT}$		Quad	Lin	Quad	Quad	11	6.6%
	$\hat{E}$		Quad	Lin	Quad	Lin	10	7.2%
	$\hat{T}$		Poly	Reci	Reci	Poly	12	7.1%
FPGA	$\hat{P}$		Poly	Quad	Poly	Lin	13	7.1%
	$\hat{LAT}$		Poly	Lin	Poly	Lin	10	8.1%
	$\hat{E}$		Quad	Quad	Lin	Quad	11	7.9%
	$\hat{T}$		Quad	Reci	Poly	Lin	10	3.3%
	$\hat{R}$	ALMs	Log	Poly	Quad	Poly	13	9.4%
		ALUTs	Poly	Lin	Poly	Lin	12	11%
		LABs	Log	Poly	Quad	Lin	11	11.8%
		M20Ks	Poly	Lin	Quad	Quad	12	10.1%

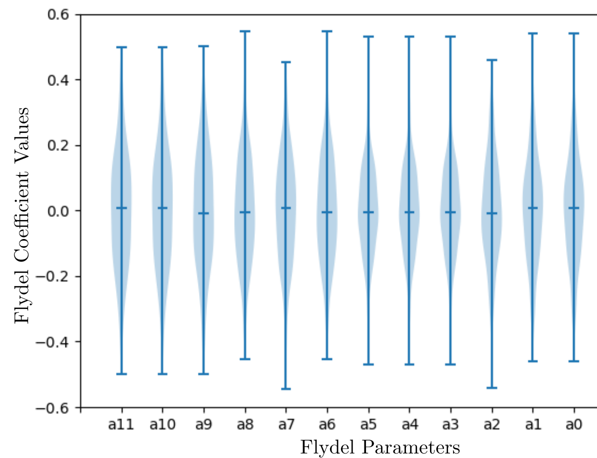
TABLE 2.4: Selected single-feature models per KPI and per device using RMSE as loss function with L2 regularization term. $\#\theta^*$ is the number of total parameters chosen for the strong regressor KPI.

most appealing in terms of energy efficiency, many implementations exhaust quickly the number of resource elements on the device. Generally, the models are able to interpolate relatively well on the range of the configuration space where they were trained. However, as the measurements extend beyond the training dataset the precision degrades quickly. To address this extrapolation problem, more training measurements shall be added to the training data. These training samples must be representative enough of the application.

The weak regressor selection highly depends on the subsampled dataset selected per feature. Therefore, the choice of a representative enough set of samples dictates the precision of the single-feature selected model, since the relation of the KPI is generalized from these data points. It has been chosen a random subsampling solution by keeping some activity features constants, as illustrated in Figure 2.7. However, depending on the nature of the dataset, it is possible that these relations are not preserved from different subsamples, which may lead to a bad model selection that may not fully represent the full dataset. Because of the monotonous growth or reduction of the models in the given dataset, it is assumed that dataset complies with this relationship preservation. Another aspect to take into consideration is the interpolation and extrapolation capabilities of the obtained models. Since the dataset was obtained from typical feature ranges found in common CNN architectures, the k-fold cross-validation allows a good approximation to test whether over-fitting occurs and therefore, a good estimation for interpolation. However, the ensemble may still select a model that overfits training points and prevent extrapolation. For instance, the latency evaluation of the CPU model ($\hat{LAT}_{CPU}$) chooses a polynomial model for feature C_I where a linear model may be more suitable. Even considering the regularization term on the cost function, this model was obtained from



(A) Absolute parameter values boxplots on an example of Latency KPIs on the GPU ($L\hat{A}T_{GPU}$).



(B) Centered and scaled parameter values violin plots on an example of Latency KPIs on the GPU ($L\hat{A}T_{GPU}$).

FIGURE 2.9: Obtained parameter distribution using 50 iterations of 10-fold cross-validation.

the aggregation stage. Although the NRMSE is small for the evaluated data points, the cost function error may start to degrade over extrapolation data points. To alleviate this problem, the training set may be expanded if possible, but it depends on the availability of platforms to test the considered cases.

Testing over a validation dataset, the parameter values must have a low relative standard deviation. It is tested with Gaussian forms on the distributions to give confidence in the standard deviation and test the white-noise nature of input parameters on each model, per KPI. As a validation example of a single KPI on a device, in Figure 2.9, it is shown the respective obtained parameter distribution using K-fold cross-validation (with $K = 10$) over the dataset as presented in Section 2.5. It is iterated over 50 times to extend the obtained parameter and metric validation sample distributions by shuffling the dataset on each iteration. Subfigure 2.9a, shows the absolute parameter box plot values over the strong regressor model for latency KPI on the GPU. It can be observed that the

variance of the model parameters over different fold is relatively low per parameter. On the other hand, Subfigure 2.9b shows the centered and scaled parameter distributions for the latency modeling on the same device. It is visually confirmed that the parameters follow a Gaussian distribution resulting from a random-process. Thus, it can be effectively confirmed that the defined models are robust to the samples where the models have not been trained with. However, this measurements samples must have the same properties and nature that the ones used to generate the original dataset. Nevertheless, because of the randomly-excited methodology for SI, it is assumed that the generated dataset is representative enough to cover a wide range of application range. For this purpose, the excitation input must also cover statistical representative sample of the range of the system operation.

Up to this point, the flydel performance has only been validated on the proposed obtained randomly-excited measurement dataset subset. However, in this Section it is further shown that the performance of flydels can be extended to state-of-the-art CNN layer models. This is mainly because common CNN architectures use Batch Normalization (BN) layers that bound the IFM values to a range of a normal distribution with approximately zero mean and unit variance ($P(IFM) \approx \mathcal{N}[0, 1]$) [SS15]. Similarly, for the case of many pre-trained CNNs architectures, a considerable amount of kernel weight values are centered around 0 with a given variance ($P(\Theta) \approx \mathcal{N}[0, \sigma^2]$). The quantization step increases even further the number of zeros. This has a considerable impact for hardware implementation of quantized CNN layers on FPGAs using dataflow techniques, such as DHM [APS+17]. Since weights are directly mapped in hardware, increasing the sparsity and number of zeros also reduces the quantity of computing components and registers used for the dataflow execution. As an example, Subfigures 2.10a, 2.10b, 2.10c and 2.10d from Figure 2.10 show the kernel pre-trained weight distributions of the first 4 layers of VGG16 CNN [SZ14]. Subfigure 2.10e shows the number of potentially saved multipliers per layer using an 8b quantization over ResNet18 [HZRS15]. Since deeper layers in ResNet18 also include a higher count of zero-valued weights, these layers save the most the number of multipliers when directly mapping convolutional layers on hardware for FPGAs. The pre-trained models were obtained using the *torchvision* model zoo with PyTorch [PGC+17] on ImageNet dataset [DDS+09].

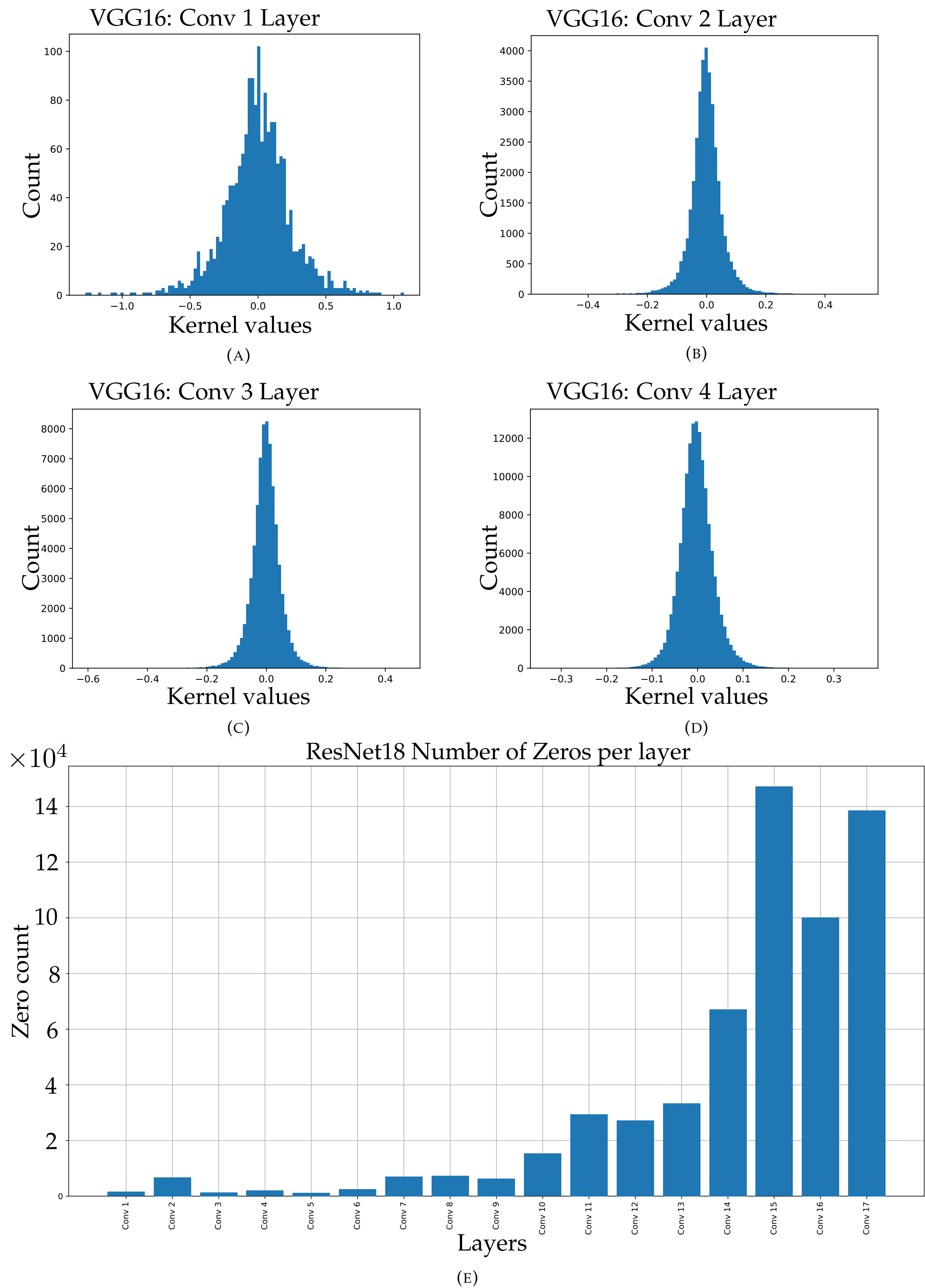


FIGURE 2.10: Weight value distributions of the first four convolutional layers of VGG16 (2.10a-2.10d). ResNet18 number of zeros per layer 2.10e.

Finally, Figure 2.11 shows the precision error of the performance estimation using the obtained flydels on the CPU (red), GPU (green) and FPGA (blue). Solid bars represent the actual performance measurements of the processing device, while dashed bars represent the flydel estimation. The flydels (dashed bars) are tested on the first layer of VGG16 ($k = 3$ and $N = 64$) [SZ14], InceptionV3 ($k = 3$ and $N = 32$) [SVI+15] and SqueezeNet (0.5x) ($k = 3$ and $N = 64$) [IHM+16] as accelerated partitions. Similarly, these CNN models were pre-trained with ImageNet dataset using BN. Thus, the weight value distributions are normally distributed, as shown in Subfigures 2.10a, 2.10b, 2.10c and 2.10d. Samples from ImageNet dataset ($3 \times 224 \times 224$), were randomly selected to feed-forward the IFMs of each input layer. For InceptionV3, these sample images were adjusted to a size of $3 \times 299 \times 299$, to match model configuration. For the FPGA, a worst-case scenario is also shown, where there are no zero nor one values on the kernel weights. This way, the compiler is forced to map each weight employing DHM. Since the flydels were trained using a dataset based on normal distribution excitation, validating over a biased worst-case scenario gives the maximum possible error between estimation and (from 18% to 22%). On normal cases for he FPGA, the error ranges between 6.2% and 6.4%. For the CPU, the error oscillates between 6% to 8%. On the GPU, the error is around 5%. These values are not significantly different to those presented in previous validation on Table 2.4. Thus, validating the efficiency of flydeling on pretrained CNN models on ImageNet.

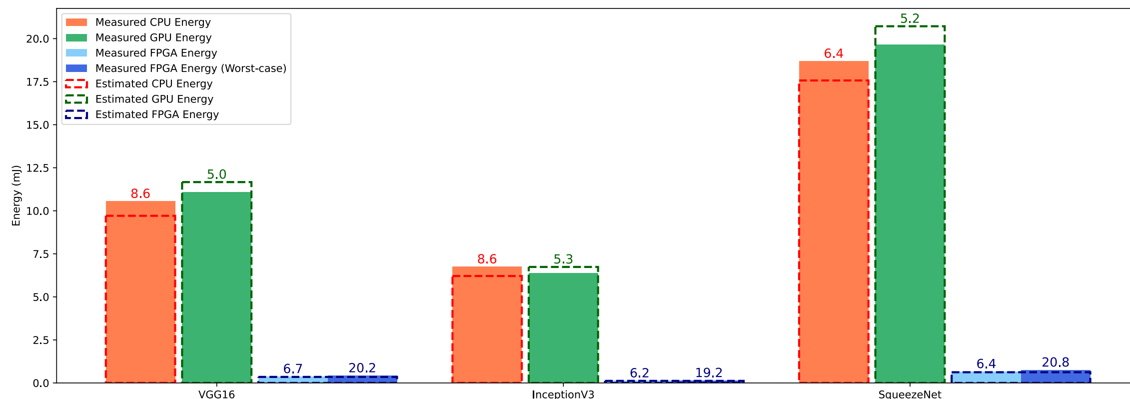


FIGURE 2.11: Comparison of pretrained CNN models energy performance with ImageNet against flydel estimation. On each KPI estimation, the percentage error is shown over each bar.

2.7 Conclusions

In this Section, it has been proposed a method called *flydeling* to generate very lightweight CNN performance models by applying random system excitation and using black-box System Identification (SI) from CNN hyperparameters. For an embedded CPU-FPGA-GPU heterogeneous accelerator platform, it has been demonstrated that it is possible to compute streamlined models of KPIs, called flyweight models (abbreviated as *flydels*), obtaining good KPI modeling accuracy, at an early-stage in development, from high-level application activity features. For latency, energy and resource utilization, error values oscillating between 5% and 10% were obtained with less parameters compared to state-of-the-art solutions. These homogeneous models are especially useful for acceleration partitioning, scheduling, and Design Space Exploration (DSE) of CNNs at application-level on heterogeneous platforms. Following Chapters 3 and 4 include the exploitation of *flydels* for design automation of heterogeneous systems.

Work	App	Model Selection	Features	Platform	KPIs	Models	Reg	Parameter Number	Error %
Shahid et al. [SFML19]	FFT GEMM ...	Error Minimization	PMCs	CPU	Energy	Linear	-	6	14.3∼68.5
Derumigny et al. [DGB+20]	General Purpose	ILP Maximization	μOps Instructions Cycles	CPU	Latency Throughput	K-means Graph LP	-	>10K	$\times 10^{-5}$
O'Neal et al. [OBSK17]	Computer Graphics	OLS and NNLS Minimization	PMCs	GPU	Power Energy	Linear	Lasso	40	7.45
						Linear	Lasso	30	7.47
						RF	-	577	10.28
						Linear	Lasso	23	13.87
						Linear	Lasso	83	13.68
						Linear	Lasso	59	8.91
Ma et al. [MCVS20]	CNN	Performance Maximization	Tile Size MAC DRAM BW Core frequency	FPGA	Latency Throughput	Non Linear	-	*	3
Goel et al. [GKBS20]	CNN	Error Minimization	MAC BRAM	FPGA	Latency	RF	-	*	6.6
						DT	-	*	7.4
						Linear	Lasso	4**	7.7
						Quadratic	-	7**	8
This	CNN	LSE Minimization	Application Specific Properties	CPU	Latency	Polynomial	L2	14	7
					Energy	Polynomial		12	7.7
					Throughput	Non-linear		12	10.7
				GPU	Latency	Polynomial	L2	11	6.6
					Energy	Polynomial		10	7.2
					Throughput	Non-linear		12	10.3
				FPGA	Latency	Polynomial	L2	10	8.1
					Energy	Polynomial		11	7.9
					Throughput	Polynomial		10	3.3
					Resources	Polynomial		11∼13	9.4∼14.1

(*) Not specified
(**) Estimated for a single DPU

TABLE 2.5: Modeling state-of-the-art comparison.

Chapter 3

Heterogeneous Partitioning Techniques

3.1 Chapter abstract

As discussed in Chapters 1 and 2, embedded heterogeneous accelerators are rapidly gaining momentum as a dominating solution for DL on Internet of Things (IoT) devices. GPUs and FPGAs are common architectures found in these platforms. Time constrained execution and energy efficiency remain a challenge, given inter-device communication overheads. In this Chapter, the heterogeneous platform characterization is extended with a module-level partitioning that exhibits a substantial performance gain over a fully-homogeneous GPU deployment using a Direct Hardware Mapping (DHM) technique on FPGA exploiting model irregularities. These irregularities are exhibited as diverse FMs dimension shapes through different data paths in the CNN model. The FMs are the result of several different operation configurations of convolutional layers, for instance; different kernel sizes, number of kernel filters or FM merging like concatenation or addition.

Despite the high-performing design obtainable with DHM, the resource requirement using this technique is considerable, preventing a full FPGA CNN DHM implementation. Multiple CNN hybrid partition parameters are evaluated in this chapter. To estimate platform performance, a heterogeneous analytical model derivation is proposed from individual computation and communication models. A heterogeneous FPGA-GPU acceleration is achieved for image classification inference task with respect to a fully GPU implementation over SqueezeNet (21%-28% energy reduction, same latency), MobileNetv2 (12%-30% energy reduction, 4%-26% latency reduction) and ShuffleNetv2 (25% energy reduction, 21% latency reduction). It is then demonstrated, that the presented partitioning techniques effectively scale over complex state-of-the-art mobile CNN models with similar results reported in state-of-the-art solutions. Adapted heterogeneous partitioning methods will dictate the nature of the optimization problem formulation from Chapter 4.

3.2 Introduction

IoT and the emerging adoption of heterogeneous architectures in edge devices has led to suitable technologies for DL applications. Furthermore, these applications require a

well-matured programming methods exploiting parallel distributed systems and parallel programming techniques [SEP+09, HTL10]. These distributed server approaches have widely opted for the deployment of multiple hardware architectures, like monolithic SIMD architectures like GPUs, or custom logic like FPGAs. However, for embedded design, different considerations on specific constrains are required, i.e. hardware resources, energy consumption, throughput and latency. Additionally, deploying inference computation on pre-trained CNN models for edge devices have faced multiple challenges during this turbulent and hectic phase of DL research. The current state-of-the-art on CNN models for image classification is still divided on deciding when one solution outperforms the other and in which cases. Therefore, as a proposed solution, embedded platforms have widely adopted an heterogeneous architecture approach to keep up with an ever-growing computing demand. Nevertheless, increasing the complexity of the system's architecture rises also the intricacy of application mapping. Because of the latency and energy performance constraints of each individual node, a hardware-aware NN mapping methodology must be well defined to increase efficiency.

In this Chapter, the evaluation of a predefined module-wise partitioning based on common building-blocks tasks on DL processing performance estimation is proposed. In this case study, the aim is to evaluate the inference deployment of mobile CNN models, like MobileNetV2 [HZC+17, SHZ+18], ShuffleNetV2 [ZZLS17, MZZS18] and SqueezeNet [IHM+16], on an embedded FPGA-GPU heterogeneous platform. Although both hardware architectures have been well studied and evaluated on HPC centers, their specific capabilities are still to be exploited on embedded design. It is shown that some task and data partitions are more suitable to be assigned and scheduled on an specific device than others. In this dissertation, estimation is proposed of energy, latency and throughput from multiple modules used in these CNN models. Furthermore, if a specific layer can not be fit or may not be efficient to be executed on a device, an heterogeneous version of grouped or depth-wise convolution partitioning for layer-fusing are proposed, when possible.

The contributions of this Section consist of:

1. adapting module-level partitioning methods of the most popular embedded deep networks for image classification to heterogeneous platforms.

2. demonstrating that heterogeneous energy and latency models using DHM [APS+17] can be derived from individual/homogeneous light-weight device computation and communication modeling.

3. demonstrating that a combination of GPU and FPGA at a DL module-level effectively outperforms homogeneous solutions, even when inter-systems communication overheads are considered.

This Chapter is organized with the following structure: Section 3.3 gives an overview of the main concepts, operations and terms found in the state-of-the-art CNN mobile models and parallel layer optimization for heterogeneous platforms. Section 3.4 summarizes related works. In section 3.5, not only the main difficulties and limitations for

heterogeneous solutions adoption are highlighted, but also the motivation behind its benefits. Section 3.5 describes important concepts for the experimental methodology description of Section 3.6 where also, the experimental results are evaluated and validated. Finally, Section 3.7 concludes with a discussion on the benefits and limitations of CNN heterogeneous processing.

3.3 CNN partitioning on GPU-FPGA platforms

In Chapter 2, the main operations of CNN layers were introduced. In this Section, it is further extended to the definition to more specific CNNs models. While the background of the CNN DL has been well-studied in the literature, it is important to be familiarized with the most common and important computation done on state-of-the-art models. Multiple acceleration techniques have been proposed in multiple domains even before the conception of the modern term of DL. The algorithmic parallel optimization have allowed their implementation on hardware in different ways. In [MV19] an exhaustive list of techniques has been covered for GPUs. Similarly, [Mit18] expands those techniques to reduced precision techniques for FPGAs. In [MV15] the challenges of CPU/GPU heterogeneous platforms have been mentioned and explored on HPC servers. In this subsection, we formalize three partitioning techniques used on the CNN models as a combination of basic techniques, such as: loop reordering, loop unrolling, tiling, batching, pipelining and/or data compression. We also describe the common computing operations and how the proposed partitioning methods are adapted to GPU-FPGA partitioning for the main building blocks with the mobile CNN models:

Layer-wise Tiling: Layer-wise tiling is a popular technique to parallelize a CNN that splits an IFM and creates two or more tensors to be processed individually. Usually, these split partitions are equally distributed to balance the workload. However, on heterogeneous platforms, workload balancing is not as easy. When tiling, the features H_I and W_I are split and mapped to each device. Figure 3.1 illustrates a partition that splits the feature H_I into two contiguous tensors: One for the GPU (H_G) and one for the FPGA (H_F). Therefore, $H_I = H_G + H_F$. Notice that all other features remain constant; $W_I = W_F = W_G$ and $C_I = C_G = C_F$. Additionally, the N kernel filters are also not split. They are all manipulated in both architectures.

Depth-Wise separable Convolution (DWConv): This technique were first described in [Cho17] and fully utilized in [HZC+17, SHZ+18]. The main concept relies upon factorization of a traditional convolutional layer. The first of the resulting operations is a $k \times k$ convolution over every single input channel followed by a 1×1 convolution. Despite the fact that a DWConv produces the same output, the number of multiplications operations is reduced by a factor of $\frac{1}{k^2} + \frac{1}{N}$, with k being the size of the kernel filter and N the number of kernel filters. This factor is multiplied by the number of multiplications on a traditional convolutional layer. In Figure 3.2, the first operation is done on the GPU and the second one is done on the FPGA, since 1×1 has less parameters to be directly mapped. Additionally, the required memory for weight storage can be also separated

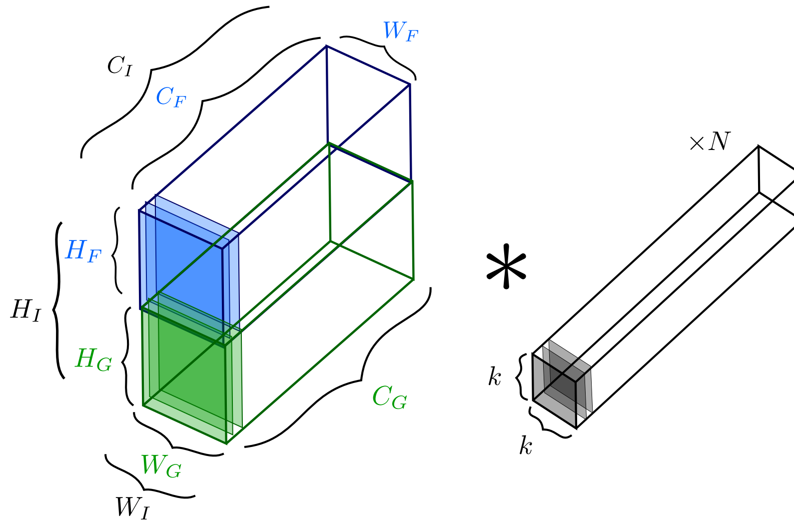


FIGURE 3.1: Tiling on a heterogeneous platform with a single GPU and a single FPGA with configuration: $H_I = H_G + H_F$, $W_I = W_F = W_G$ and $C_I = C_G = C_F$. The blue tensor is the IFM partition mapped on the FPGA. While the green tensor is the IFM partition mapped on the GPU.

as indicated in 3.2. For this operation, the device with less storage or with high-latency memory transfers can handle the 1×1 convolution layer. For this use-case, because of the reduced number of on-chip registers, this task is off-loaded on the FPGA. The limitation of FPGA resources and further details are discussed on Subsection 3.5.2.

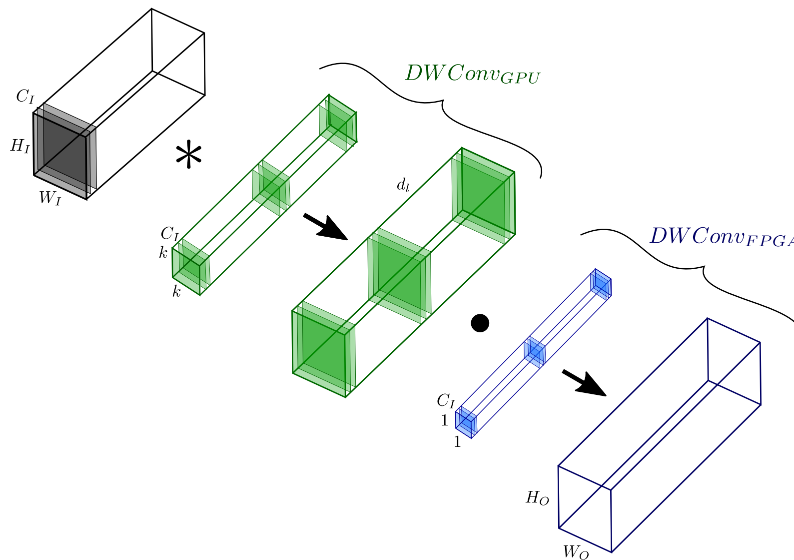


FIGURE 3.2: Heterogeneous Depth-wise convolution example where the $k \times k$ convolution per input channel is executed on the GPU and the Conv1 $\times$ 1 convolution is done on the FPGA. The partition in blue represents the data and task workload on FPGA, while in green the partition on the GPU.

Grouped Convolution (GConv): A similar approach was first implemented in [KSH12] in a homogeneous form, since their hardware architecture was composed of two Nvidia GTX GPU devices of the same type. This partitioning method divides the computational load in one or more workflows that can be executed in parallel. However, the main difference from [KSH12] consists in how the IFM is partitioned, since in GConv there is

no IFM duplication. In Figure 3.3 both devices execute simultaneously their respective convolution operation over different IFM and different weights and at the end of the workflow, both output tensors are concatenated together.

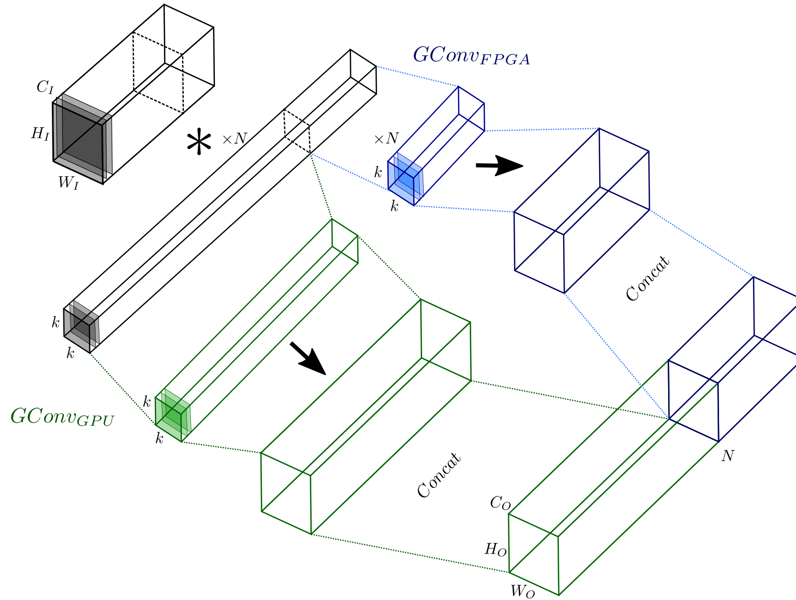


FIGURE 3.3: Heterogeneous Grouped convolution example where the C_I input channels and kernel filters are contiguously and heterogeneously divided on each device. Both resulting partitions are finally concatenated. The partition in blue represents the data and task workload on FPGA, while in green the partition on the GPU.

Similar to tiling, the channel-wise loop unrolling also usually found in the grouped convolution technique splits the IFM and kernels filters along the channel depth. However, in contrast to tiling, the kernel weights are also split, yielding to a memory requirement reduction. This is a desired consequence for devices with a limited memory footprint. Figure 3.3 shows a grouped convolution partition with two contiguous tensors and $N \times 2$ kernel filter partitions. For this instance, $C_I = C_G + C_F$. Notice that all other features remain constant; $H_I = H_G = H_F$ and $W_I = W_F = W_G$.

Fused-Layer: This partitioning was first introduced in [AFM16] as a method to store intermediate weights and neuron activity in cache from adjacent layers in depth. This approach handles one of the most common challenges in CNN models, the data transfer burden. Having a faster memory closer to the processing elements reduces the latency, avoiding off-chip data transfers. However, this method results in a trade-off on the use of memory resources inside the accelerator device, scarce resources on embedded design. Depending on the accelerator capacity, multiple containerized partitions can fit on a device. In Figure 3.4, layers are internally stored on the FPGA to be executed in a pipe-lined fashion [ZWTD19]. The OFM of the last layer in the partition is then transferred to the GPU. Notice that this opens a benefit in inter-device communication, since in most model architectures, the deeper the layer, less feature map elements require to be moved.

Table 3.1 summarizes the heterogeneous partitioning methods characteristics. Each method keeps some structural features static while executing the partitioning over the

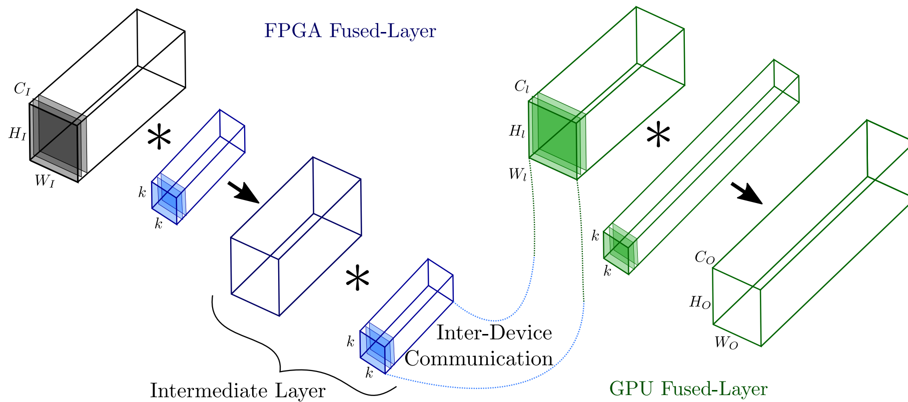


FIGURE 3.4: Heterogeneous Fused layer example where a couple or intermediate layers activity are stored in the internal FPGA RAM memory. Afterwards the output tensor is transferred to the GPU for deeper layers processing. The partition in blue represents the data and task workload on FPGA, while in green the partition on the GPU.

remaining features. Methods, such as tiling and grouped convolution, require further operations for synchronization, like merging intermediate results or concatenating resulting feature maps. As a consequence, one accelerator must be in charge of this synchronization. It is also important to notice, that some methods are more computational-efficient than communication-efficient. For instance, the depth-wise separable convolution reduces the number of multiplications making it more computation-efficient. While, fused-layer reduces the FM transfers between accelerators, offering a better communication efficiency. To define a clear winner between these partitioning methods is a difficult task since, what may be a good choice for one heterogeneous platform, might be a poor choice for another one. Specifications and limitations such as: internal memory size, communication bottlenecks and a reduced count of processing elements; play a big role for the choice of the adequate partitioning method.

Partitioning Method	Partitioned Features	Fixed Features	Computation trade	Memory trade	Communication trade	Execution
Layer-wise Tiling	H_I, W_I, C_I	k, N	$\frac{k^2 \times C_I \times H_O \times W_O \times N}{f_T}$ [1]	$\frac{H_I \times W_I \times C_I}{f_T}$	H_O, W_O, N	Parallel
Depth-wise separable Convolution	k, N	H_I, W_I, C_I	$\frac{1}{k^2} + \frac{1}{N}$	$\frac{1}{k^2} + \frac{1}{N}$	-	Pipelined
Grouped Convolution	C_I, N	H_I, W_I, k	$\frac{k^2 \times C_I \times H_O \times W_O \times N}{f_G}$ [2]	$\frac{C_I}{f_G}, \frac{C_I}{f_G}$	$\frac{H_I \times W_I \times C_I}{f} + \frac{H_O \times W_O \times N}{f_G}$	Parallel
Fused-layer	-	-	-	H_I, W_I, C_I, k, N [3]	H_O, W_O, N	Pipelined

TABLE 3.1: Heterogeneous partitioning methods characteristics.

[1] Where f_T is the partitioning factor for the tile size depending on H_I,W_I,C_I.
[2] Where f_G is the partitioning factor for GConv.
[3] Increased memory requirements.

3.4 Related works

Single Chip-based accelerators: A wide variety of architectures are built on with the purpose of CNN inference deployment on edge devices. ASIC based designs, like DaDianNao style [CLL+14], usually offer a good balance between flexibility and performance with considerate power dissipation. The small size of an ASIC node opens the possibility of good scalability on a single die with a small footprint. However, what it gains in efficiency and scalability, it loses it in reconfigurability and flexibility to quickly test designs. Following a similar approach as DaDianNao, Stripes [JAH+16] accelerates computation extending the capabilities of DaDianNao by exploiting a dynamic quantization of input features. Mixed precision training and inference [Cas18, DMM+18] outperform traditional full precision methods both in training and inference time for general purpose hardware. Therefore, many works show success exiting from traditional floating point representations, i.e. 32 bits and 16 bits. A 8 bit floating point hardware engine in [WCB+18] claims to have a 2×-4× better efficiency with intermediate accumulators of FP16 than whole FP16 computations, given that multipliers are smaller and smaller accumulator bit width. Even lower bit representations like ternary weight representation [LZL16], are mapped using Register Transfer Level (RTL) on both FPGA and ASIC, achieving lower memory requirements and more efficiency in energy consumption [ALPP17]. Similarly, in this Chapter, a quantization to reduce logic elements utilization from the FPGA side is used.

Multinational companies have also created their own specialized hardware for convolution acceleration. Google introduces an ASIC based on systolic array techniques with integer quantized computation for their. The core of TPU [JYP+17] takes a flow of 256 Byte inputs simultaneously and the partial sums are streamed through deeper computations to the bottom of the matrix multiply unit. Intel® introduced in 2019 the Intel® Neural Compute Stick 2 [Mov19], bringing the Vision Processing Unit (VPU) to the edge. Based on a Movidius Myriad X VPU with 16 programmable cores, it is optimized to execute inference for vision and imaging tasks on dedicated DSP.

GPU and FPGA based accelerators: Heterogeneous computing has been a trend that has highly appealed to the interest of hardware accelerators since the last two decades. These computing nodes have a diversity of capabilities, different ways to execute instructions or different operation handling [Zah17]. In cases where there is enough parallelism the cores can take advantage of these scheduling, FPGAs and GPUs can offer a significant performance improvement on energy efficiency compared to homogeneous platforms [CMHM10]. In [CMHM10], it is also discussed in which cases it may be better to allocate applications on custom logic or on FPGAs and GPUs.

A debate has been raised to show when FPGA solutions were able to achieve a similar or even better performance than other processing nodes. In [SDV+14] traditional image processing tasks were evaluated in different platforms to compare results in implementations. However, the developing time and integrateability of the FPGA resulted less efficient and non-evident. Further work feed the discussion on the subject of embedded vision applications comparing an ARM57 CPU, a TX2 GPU and a ZCU102 FPGA [QDL+19].

The FPGA has shown a reduction of 1.2×-22.3× in energy/frame compared to the other two platforms for vision tasks like background segmentation, colour segmentation, Harris corner tracking and Stereo block matching. Some works have taken advantage of the benefits of both worlds creating a hybrid platform for HPC. These platforms are today built on a host/guest structure and on the Bulk Synchronous Programming (BSP) model of computation. The host processing system, built from high-end general purpose cores, drives data movements and requests processing from a guest processing device, retrieving output data when ready. In current systems, hosts are multi-ARM out-of-order processors and guests can be either a GPU, and FPGA or both. In [SBD+13] a heterogeneous platform consisting of two programmable logic devices (Virtex 6 LX240 FPGA) and an Nvidia® Tesla C2050 are interconnected to be tested on image processing on Histogram Oriented Gradient (HOG) and machine learning technique SVM using the proprietary tool CUDA. While some speed-up are achieved, the communication in PCIe bandwidth presented a reduction of the speed up resulting in a bottleneck given a fine-granularity partitioning. Therefore, some works [BRF14, TDMP15] try to reduce communication bottleneck issues by bypassing or skipping data allocation at host memory. In [TDMP15] three setups with different capabilities and PCIe generations are evaluated. Their throughput performance is measured with different data sizes to mitigate the communication overhead. This is critical in order to increase number of layers or parameters to be mapped on the accelerators and be sent back to another one, as presented in this Chapter. A related work [HBNY19] uses a platform based on a multicore CPU/GPU Nvidia Jetson TX1 with a Xilinx Zynq MPSoC (ZCU102). The proposed platform was tested on image processing algorithms like histogram, dense matrix-vector multiplication and sparse matrix-vector multiplication. Finally, three works are identified with similar results and architectures using layer-wise partitioning [OHY+18], IFM as batch partitioning [VGG+20] and two-stage partitioning for feature extraction and classification [TST+19]. Batch partitioning does not split single IFMs and it is only used for training, it achieves parallelism by distributing non-duplicate data from different IFM samples. This partitioning technique is similar in computation to Grouped convolution. Although the results from these works are evaluated with small CNN models, they demonstrate that performance gain is also feasible in the case of more computationally complex models for image classification on embedded devices.

In [NSS+16] the use of Binary Neural Network (BNN) [CHS+16] on FPGAs and ASICs offer in some cases a better performance and performance per Watt in comparison to CPU and GPU given the simplicity of the processing elements substituting multiplier units with XNOR logical gates and adders as popcount with Look-Up Tables (LUTs), an efficient low-consuming resource on FPGAs and ASICs. Authors tested multiple layers in an Arria 10 FPGA with 64 and 1024 processing elements and ASICs outperforming with this hardware-friendly model a GEMM based kernel on an Nvidia Jetson TX1 embedded GPU. In [UFG+17], FINN is presented. Based on heterogeneous streaming architecture and binary CNNs, FINN is capable to execute different sized compute arrays with a pipelined stream. The main core of FINN is the Matrix-Vector-Threshold Unit (MVTU), where the batch normalization layer is treated as a threshold and the data is handled as a data

stream by folding Matrix-Vector products, similar to the elemental unit in Stripes. Authors show results using a Xilinx Zynq-7000 All programmable SoC ZC706, addressing typical challenges from embedded design. This heterogeneous platform consists of an ARM CPU and a FPGA. In [AHMSNY18] the work from FINN is extended using a Decision Making Unit (DMU) based on a Softmax function layer that decides to deploy a high-accuracy network or keep high throughput depending on the classification error. This frame-dependant multi-precision CNN preserves accuracy for embedded heterogeneous platforms.

Partitioning, scheduling and architecture search or exploration: Scheduling and task partitioning between heterogeneous multiprocessors has been extensively addressed as one of the main application of constraint programming. In [LSGE11] a two-stage optimization heuristic is proposed using an ILP technique on a static Synchronous Data Flow (SDF) to generate a Pareto front for latency and processor cost over a series of iterations for real-time streaming. For design and synthesis challenges on mobile or embedded heterogeneous processors, [WPM18] highlights the importance of the software support to facilitate design exploration. A good compile time strategy can assist the designer to allocate, on early stage, to an adequate processor a specific kernel. In this work, it is also brought to the attention of the reader, that techniques of parameter scaling, such as Dynamic Voltage-Frequency Scaling (DVFS) must also balance the over-working of a single processing core. As a conclusion, authors add that DVFS can benefit the stability and load-balance of the system. NasNet [ZL17] started a tendency for more sophisticated optimization algorithms based on the model architecture known as Neural Architecture Search (NAS). In traditional NAS, not only the weight parameters are learnt from a given CNN architecture, but also the topology of the network itself is updated. Based on this original work, other techniques take into consideration the hardware architecture and performance, like ChamNet [DZW+18] MNasNet [TCP+19] or FBNet [WDZ+19]. Using these heuristic techniques for design space search, some works like [KLKC19] have exploited it with gains in speed-up or for a smaller area and energy on ASIC designs [YYL+20].

As presented in this Section, while many state-of-the-art partitioning methods for DSE are hardware-aware, they barely consider the capabilities of heterogeneous systems. This is because aiming for these hybrid platforms requires a partitioning selection with complex algorithms, such as reinforcement learning. Furthermore, this becomes even more challenging if the reconfigurable nature of some accelerators like FPGA is considered. Thus, to improve the motivation of a particular partitioning choice, partitioning methods must be well-defined aiming towards a hybrid hardware. Most works focus therefore on dedicated single-chip solutions or solutions where the reconfigurable part is fixed or pre-programmed and resources are already allocated. Another open issue that divides the scientific community is the granularity of the partitions. Some works, have demonstrated an accepted performance with a coarser granularity, preserving partitions with relatively big data chunks. This coarse-grained partitioning like layer-wise partitions benefit from memory coherency. On the other hand, fine-grained partitions allow to better handle data

redundancy and optimize data retrieval. Both partitioning methods have proven to be efficient for different accelerators.

3.5 Problem definition: Heterogeneous partitions

Performance on a given accelerator is application-, or even, task-dependant. The granularity and heterogeneity of the task profiling play a relevant role for the module deployment performance. In heterogeneous systems, it is not evident in which cases one platform outperforms the others, since different architectures exhibit hardly comparable features and drawbacks. Furthermore, in some cases the OS scheduling and optimization hides the performance under certain conditions. Decomposition into tasks is, nevertheless, necessary since modern CNN architectures use a combination of elementary building-blocks or functions. To address the case study of porting a CNN on a FPGA-GPU heterogeneous platform, we prove that the incorporation of an embedded FPGA with DHM in combination with a embedded GPU can be a relevant approach to accelerate DL CNN models at a module-level granularity.

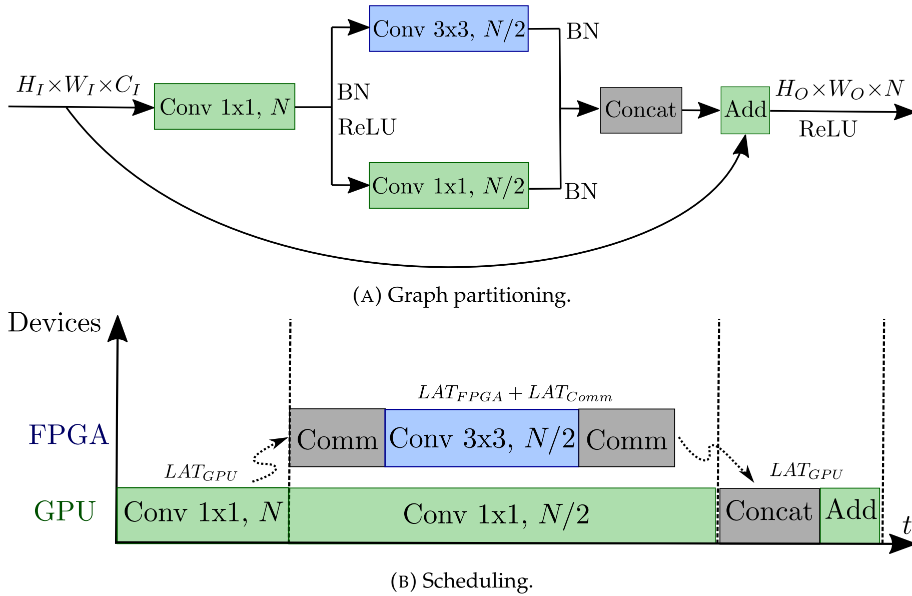
This section describes the CNN modules and scheduling to be considered for a set of popular CNN models. In addition, some important concepts for the experimental methodology in Section 3.6 are covered. Also, in Section 3.6, we focus focus our efforts on partitioning for the hardware architecture of X-MERA (Appendix B). The architecture is composed of a single CPU-GPU with a communication bus of 4-lanes Gen2 PCIe (5 GT/s or 2.5 GB/s) with a FPGA. The objective is to reduce execution time latency by adapting the CNN module partitioning on the given heterogeneous platform. The technical details of memory hierarchy and transfer speeds can be found in Appendix B.

3.5.1 Mobile CNN modules: partitioning and scheduling

While early CNNs were regular with only layer-shaped elements and systematic processing, novel methods are composed from modules. In this subsection, a module-level representation as a combination of the computation and communications operations for three popular CNN models is proposed. The three modules data-flows use the operations described in section 3.3. Data-flows are a well-explored abstraction paradigm to describe CNN applications as directed graphs, to represent data paths and to characterize computation nodes. This construct allows us to understand and identify data dependency and a temporal hierarchy execution. Additionally, the inputs and outputs of a node are observable which is fundamental for communication estimation. The output tensor of a neuron is then sent to the adjacent neuron through directed edges representing data communication. For this Chapter, it is proposed the following heterogeneous partitioning and scheduling for the following modules on state-of-the-art mobile CNN:

Fire Module (SqueezeNet): This architecture was one of the first resource efficient models [IHM+16]. Its module achieves this high resource efficiency by replacing multiple 3×3 filters by 1×1 convolutions as depicted in Figure 3.5. The data path is branched

using a Grouped Convolution creating two data-flows with the same data workload, as depicted in Figure 3.5a. However, its heterogeneity is exploited by the fact that the two data paths are treated differently. One path calls a 3×3 convolution while the other calls a 1×1 convolution and then the resulting output tensors are concatenated as explained in section 3.3. Additionally, this GConv allows us to split execution in a parallel manner on both devices as seen in Figure 3.5b. Considering that the filters values are fixed, it is interesting to have them stored on the FPGA to avoid external memory accesses on the GPU side (9x less than 1×1 convolutions). Therefore, the 3×3 convolution is executed on the FPGA. This way, global latency of the communication latency could be hidden on the parallel execution.



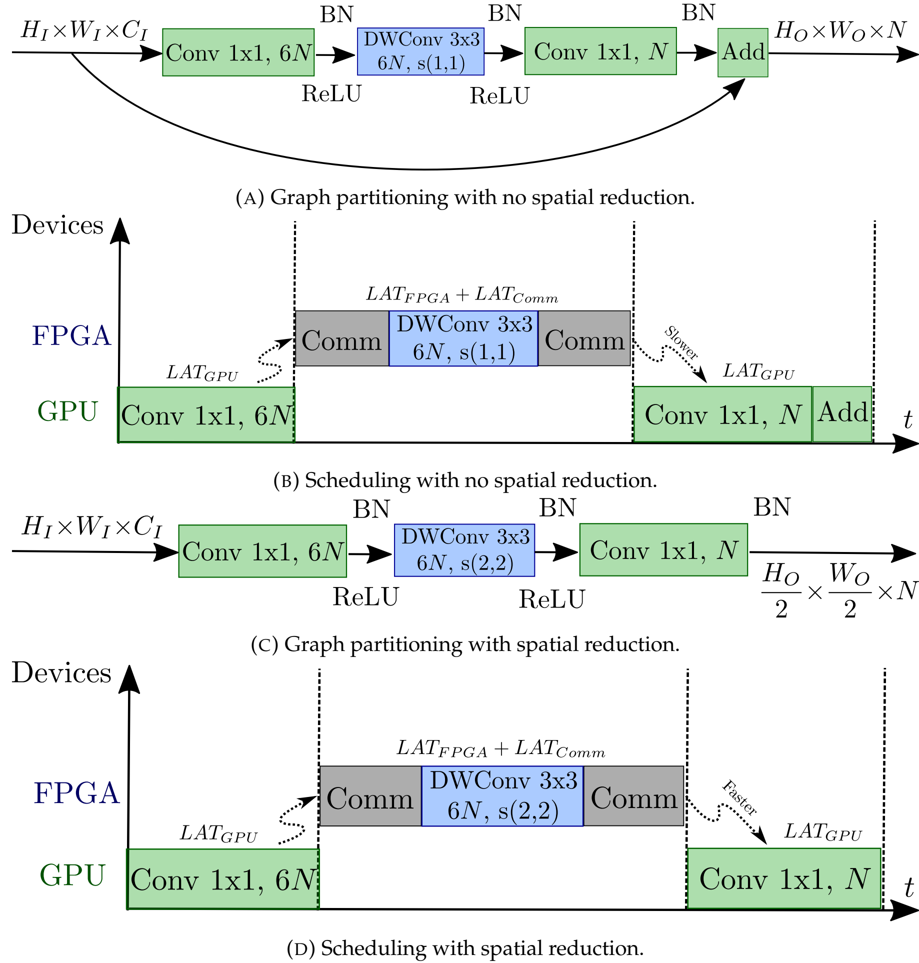
(A) Graph partitioning.

(B) Scheduling.

FIGURE 3.5: SqueezeNet's Fire module (a) graph representation and (b) scheduling. The nodes from the graph in blue are scheduled on the FPGA while the ones in green are scheduled on the GPU.

Bottleneck Module (MobileNetv2): Similar to MobileNetv1 [HZC+17], the modules of MobileNetv2 in Figure 3.6, use the concept of DWConv from section 3.3 as a form of factorization for convolution layers with less operations [SHZ+18]. Moreover, this architecture introduces linear bottlenecks connection between module layers, as seen in Figure 3.6a, to be added element-wise after processing, and a module for 2x spatial reduction using a stride of 2 on the DWConv, as seen in Figure 3.6c. The linear bottlenecks do not execute any processing and the features are added at the end of the module. This way, the features of previous layers are considered for deeper layers. In this case, differently as in SqueeezeNet's module, the execution of the graph is done sequentially and is highly dependant on the communication throughput, as seen in Figures 3.6b and 3.6d. However, because of the spatial reduction, the latency could be less, compared to bottleneck modules with no spatial reduction, as less data loads are transferred to modules with smaller OFMs. MobileNetv2 has multiple output sizes as described in [SHZ+18].

Stage Module (ShuffleNetv2 0.5x): ShuffleNetv1 [ZZLS17] introduces a channel shuffle operation and a GConv followed by a DWConv. In ShuffleNetv2 [MZZS18], the channel



(A) Graph partitioning with no spatial reduction.

(B) Scheduling with no spatial reduction.

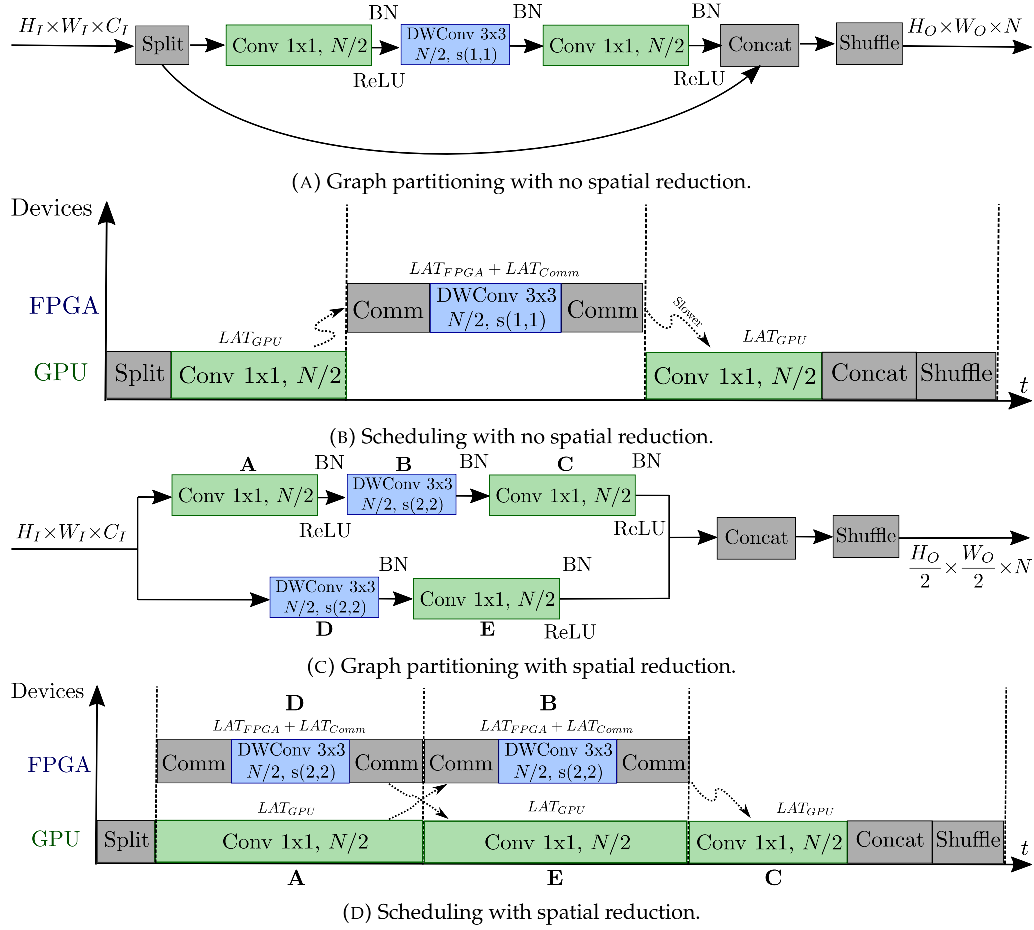
(C) Graph partitioning with spatial reduction.

(D) Scheduling with spatial reduction.

FIGURE 3.6: MobileNetv2's Bottleneck module with no spatial reduction (a) graph representation and (b) scheduling and with spatial reduction (c) graph representation and (d) scheduling. The nodes from the graph in blue are scheduled on the FPGA while the ones in green are scheduled on the GPU.

split is done at the beginning of the module and the shuffle at its end as seen in Figure 3.7. Each stage module consists of one module with spatial reduction from Figure 3.7c followed sequentially by several modules with no spatial reduction from 3.7a. For the first stage it is repeated 3 times, then 7 times for the second stage and finally 3 times for the third and last stage. Because of DWConv and GConv execution in Figure 3.7b and the branching of GConv data-paths in Figure 3.7d, the same techniques of parallel and sequential scheduling from the Fire module and the Bottleneck module, can be respectively applied. The spatial reduction is done during the DWConv execution over the half of the input feature channels result of the GConv. This feature reduction, also considerably relieves some of the overhead for inter-device communication, reducing latency on the heterogeneous platform.

The three presented modules for mobile CNNs offer different opportunities for parallel and pipelined execution. For instance, because SqueezeNet's Fire module includes a GConv block, it is possible to deploy computation concurrently on different accelerators (Figure 3.5). On the other hand, although the DWConv computation on the MobileNetV2



(A) Graph partitioning with no spatial reduction.

(B) Scheduling with no spatial reduction.

(C) Graph partitioning with spatial reduction.

(D) Scheduling with spatial reduction.

FIGURE 3.7: ShuffleNetv2's Stage module with no spatial reduction (a) graph representation and (b) scheduling and with spatial reduction (c) graph representation and (d) scheduling. The nodes from the graph in blue are scheduled on the FPGA while the ones in green are scheduled on the GPU.

(Figure 3.6) introduces data dependency and can only be executed sequentially. this greatly reduces the number of MACs and the memory requirements. This memory requirement reduction is a desired consequence of DWConv aimed for embedded design with limited memory elements. Finally, the ShuffleNetV2 Stage module, combines characteristics of both modules aforementioned modules. For this module, the data paths can be executed concurrently for spatial reduction (similar to the GConv execution), which also reduces the communication overhead with a stride window of two. Each data path introduces a DWConv block, therefore each one of them must be executed in a pipelines fashion as presented in Figure 3.7. In Chapter 4, we demonstrate how to combine these partitioning methods for automatic accelerated partition selection.

3.5.2 DHM for FPGA synthesis definition

In this Chapter, the architecture of an heterogeneous embedded platform describing both individual device architecture is covered. While the architecture of a GPU has drawn the attention of an extensive number of research papers [MV15, MV19], its single-device hardware architecture based on SIMD with shared memory space and shared control unit

remains mostly fixed. GPUs focus more on the memory handling patterns, since multiple metrics like energy consumption and latency heavily depend on the data orchestration between host and device communication, as to be expected from a SIMD-based device. On the other hand, FPGAs, as custom-logic reconfigurable devices, offer the benefit of architecture design flexibility which can benefit from the relative heterogeneity in tasks scheduling and data partitioning when porting module based CNNs.

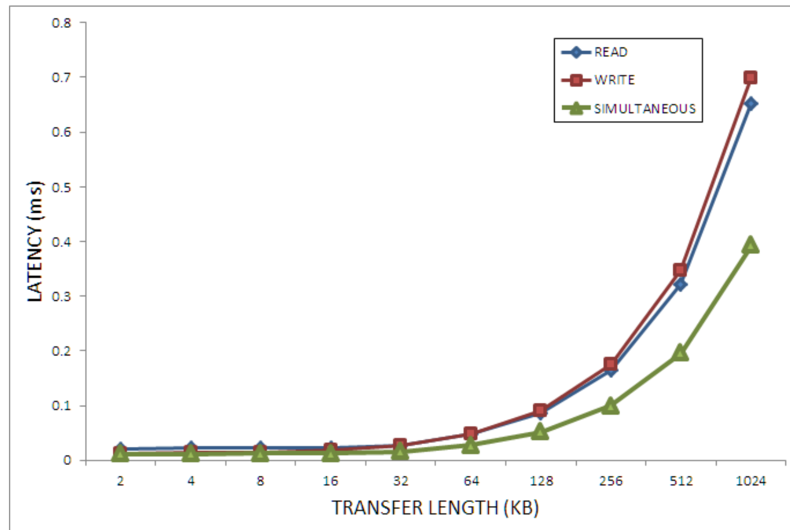
A data-driven approach is used that fully exploits the resources in a FPGA to concurrently map and execute multiple layers on the device in a pipeline manner. DHM was first introduced in [APS+17] as a technique to map processing nodes to logical elements or DSPs. The synthesized accelerators using this methodology take advantage of the fused layer technique from section 3.3, since the intermediate feature maps are stored internally in the device, as well as the kernel weights as data streams. This storage avoids the communication bottleneck of the external memory accesses, increasing energy, latency and throughput efficiency. This efficiency can be even more considerable if custom fixed-point logic is implemented. In this manuscript, every operation on the FPGA is done with an 8bit representation and cast when needed after communication on the host device, reducing communication requirements by a factor of 4x. Additionally, all weights are stored closer to the logic elements on on-chip registers, so no external memory accesses are needed, which in DL applications introduce a considerable overhead. LUT-based CNN porting [WDCC19] exploits this same idea of direct hardware mapping. Although this method offers an indisputable high performance efficiency gain, it comes at the cost of enormous resource requirements. This constrain is only intensified by the fact that the embedded FPGAs do not have large resources element counts. As a result of this constraint, only small designs can be mapped to fully exploit this high-throughput benefit.

Taking the opportunities and limitations of DHM into consideration, it may not be evident if the fitting of a full module on an FPGA device could be feasible. However, the combination of this technique on a heterogeneous platform with the objective to reduce memory accesses on a GPU and have a performance gain seems promising at first glance. As it is further developed in following sections: We show that, in fact, even if the FPGA is better than the GPU in all evaluation metrics on individual neurons, the complementary execution and time multiplexing are requested to make the porting of real-life CNNs feasible.

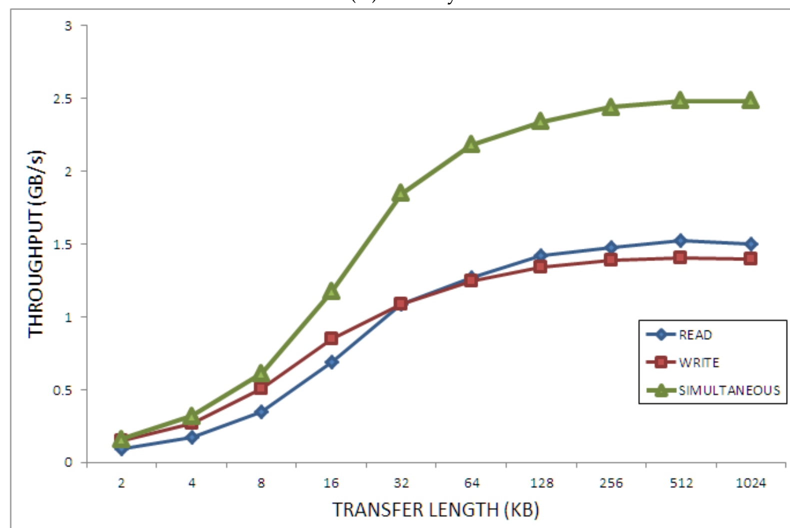
3.5.3 Inter-device communication modeling

Communication latency have a substantial impact on the hybrid platform for bandwidth performance estimation. Since communication overheads depend only on the size of the IFMs and OFMs tensors to be transferred, latency and throughput can be proportionally estimated by the size of tensors. Thus, we measure the transferred information from the PCIe channel between FPGA and GPU with several FM data sizes. These measurements are obtained directly from several data chunks for read, write and simultaneous transfers. While IFM size depends on the application, the OFM dimensions can be obtained from Equation 2.2. Figure 3.8 shows the latency 3.8a and throughput 3.8b for different data

sizes to be transferred between processing blocks. These measurements were obtained directly from several data sizes for read, write and simultaneous transfers.



(A) Latency.



(B) Throughput.

FIGURE 3.8: (a) Latency and (b) Throughput in communication using PCIe Gen2 with x4 lanes over multiple transfer sizes for read, write and simultaneous between the Cyclone10GX FPGA and TegraTX2 GPU.

In the next subsection, it is shown an analysis on how the latency and throughput performance measurements for the communication modeling has a direct impact on heterogeneous time execution and energy efficiency. As a step further on hybrid modeling, these proposed communication nodes must be included on the heterogeneous model definition. Depending on the size of data chunks to transfer and the divergence between CCR, this overhead can be considerable or even critical on considered applications.

3.5.4 CUDA microarchitecture comparison of current Nvidia embedded GPUs

Evaluation of heterogeneous implementation solutions are difficult to analyze given the nature of the development environment and the complex prototyping necessary before

conducting measurements. As a matter of fact, it is time-consuming to evaluate the performance of a single design by analyzing each processing device and communication link and the task is even more complex on multi-chips designs. In Chapter 2, it has been discussed how ensemble-based methodologies can be used to derive several light-weight multivariate device models from CNN hyper-parameters. Such a model is useful to quickly infer heterogeneous performance partitions by simply evaluating a learned analytical model with a few parameters. In addition, in the previous Subsection, it has been proposed a similar approach to profile communication overhead.

The performance of a heterogeneous model highly depends also on the architectural characteristics of the individual processing and communication nodes in the system. In current state-of-the-art, DL-oriented embedded platforms operate on either integer or floating-point numerical precision. In some cases, even mixed-precision computation is employed over dedicated hardware. One of the most well-adopted accelerators are the Tensor Cores introduced since the Volta microarchitectures on Nvidia® GPUs [MCL+18]. The inclusion of these accelerators has shown an increment in throughput and better memory caching for servers [MAMS18, SSB+20] and computing on the edge [RMJ+19, RMGV+20] in comparison to previous microarchitecture generations. A diversity of precision representation have been proposed ranging from 4bit integers to 8bit integers and single precision floating point in 32bits to double precision floating point in 64bits [MEA+19]. Moreover, in [MEA+19], authors have demonstrated that there is a high-correlation, between quantized low-precision and full-precision weights and features. For some tasks, such as classification inference, 8bits quantization from pre-trained CNNs suffice. Unfortunately, although Tensor Cores offer a clear acceleration for DL workloads, not all current embedded Nvidia® GPU microarchitectures support integer computing [HSKR21]. For instance, in [HSKR21] the Nvidia® Jetson Xavier NX offers 50% CUDA cores and supplementary 48 Tensor Cores with the Volta microarchitecture in comparison to Nvidia® Jetson TX2 with Pascal microarchitecture. Nvidia® results show that, on ResNet18, Nvidia® Jetson Xavier NX overcomes Nvidia® Jetson TX2 by more than 50% in terms of average throughput. However, as shown in Figure 3.9, for single batch or for single classification inference, the performance is heavily reduced, to only around 15%∼30% gain for common CNNs models. Nvidia® obtained empirical results from different CNNs models: AlexNet, DenseNet, ResNet18, ResNet50, SqueezeNet and VGG16. In Subsection 3.6, we show a similar performance gain acceleration and energy consumption gain using the FPGA with custom DHM architectures without the need to increase the CUDA cores count on the GPU. Another interesting remark is that this configuration offers a better performance even with higher-latency off-chip data transfers.

3.6 Partition experimental methodology, evaluation and results

In this section, we describe the experimental methodology measuring proposed metrics. In subsection 3.6.1, the individual performance for each device is measured and in subsection

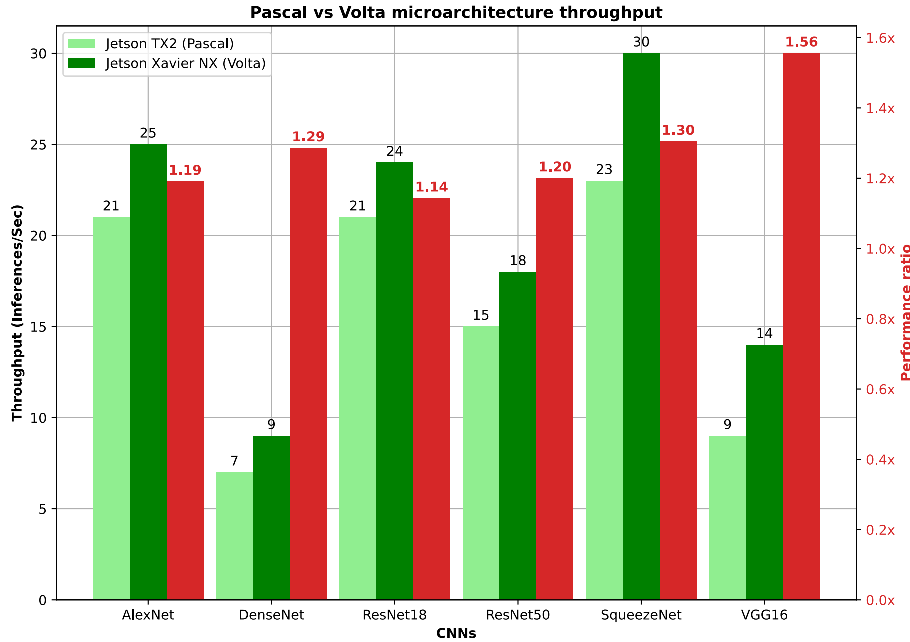


FIGURE 3.9: Performance comparison between the Nvidia® Jetson TX2 with Pascal microarchitecture against Nvidia® Jetson Xavier NX with Volta microarchitecture over many CNNs, as discussed in [HSKR21].

3.6.2, it is discussed the results of the heterogeneous platform measurements for different modules with the partitioning from section 3.3.

3.6.1 Measurement-based performance metrics comparison

This subsection compares the KPIs of common individual tasks of popular embedded CNNs to be used on the estimated heterogeneous platform on both FPGA and GPU devices. The chosen subset of convolutional filter operations are $k \in \{1, 3, 5, 7, 11\}$, since these are the common parameters to be found in state-of-the-art models.

3.6.1.1 Partition latency (KPI)

Using a similar methodology as in the dataset generation from chapter 2, it is possible to directly compare measurements of accelerated partitions against model prediction. This can be done on the individual devices of the platform. This way, inter-device comparison is achievable for case-by-case partition evaluation. In Figure 3.10 it is shown an example of a comparison between the FPGA using DHM [APS+17] and GPU implementation of different convolution workloads. In this case, not only the FPGA clearly outperforms the GPU, but the latency remains almost constant when convolution size increases. This is because the compiler tries to reduce the critical path latency for synthesis, preserving the given operational frequency of around 100MHz for the platform and operations are strongly parallelized. This means, that the more intensive the computational workload is, greater is also the difference in latency performance between FPGA and GPU devices. This claim is true as long as the design can be mapped on the device and the critical path is constant.

Since communication latency may have a substantial impact on the hybrid platform performance, the inter-device communication time from the PCIe channel with several IFM data sizes is measured. These measurements are obtained directly from several data chunks for read, write and simultaneous transfers. Therefore, it can be observed in Figure 3.14 that the transmission would highly depend on the size of the feature map to be sent to a device.

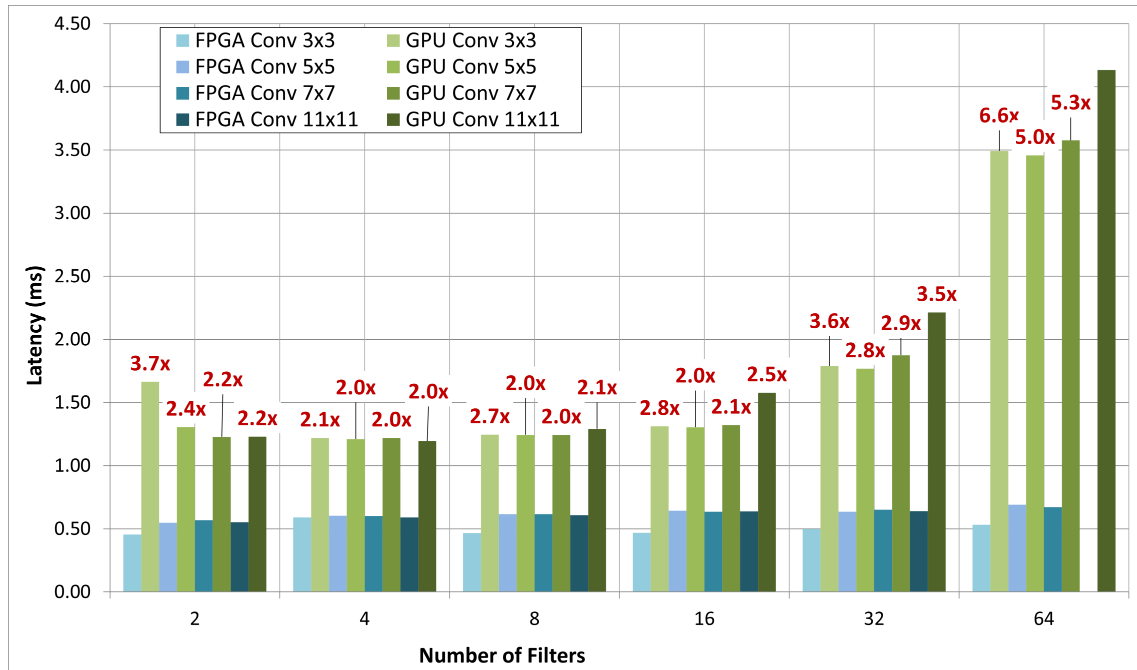


FIGURE 3.10: Latency comparison between multiple convolution function sizes on Cyclone10GX FPGA (blue) and Jetson TX2 GPU (green) for different CNN layers on an input image of $224 \times 224 \times 3$. Blue bars represent the layers implemented on the FPGA and the green bars represent the latency on the GPU. Notice that the missing blue bars represent unfeasible tasks to be mapped on more computational intensive task given the logic or memory element constrains for the FPGA device.

3.6.1.2 Partition energy (KPI)

On the Jetson TX2 MoC, a Tegra TX2 SoC is incorporated, which at the same time, includes an integrated multi-channel power monitor (Texas Instrument INA3221) used to obtain the measurements of power dissipation on the Pascal-based architecture GPU. The 3-channel monitor is configured with a 64-sampling average filter to compute a reliable average power. In Figure 3.11 the average filtering in time domain works as a low-pass filter on frequency domain, that is the reason why the measurements show a "rounded" shape between experiment iterations. The Sigma-Delta ADC used on the IC uses a 500KHz sampling rate, following the specifications from the manufacturer (Texas Instruments). For measurements acquisition, it is passed through the OS to retrieve the data which introduces an overhead on the number of samples taken per experiment iteration. However, this is the reason it is iterate multiple times on a single computing task, Conv1×1 for example, averaging over 5000 iterations. One mono-threaded program is executed on the CPU cores

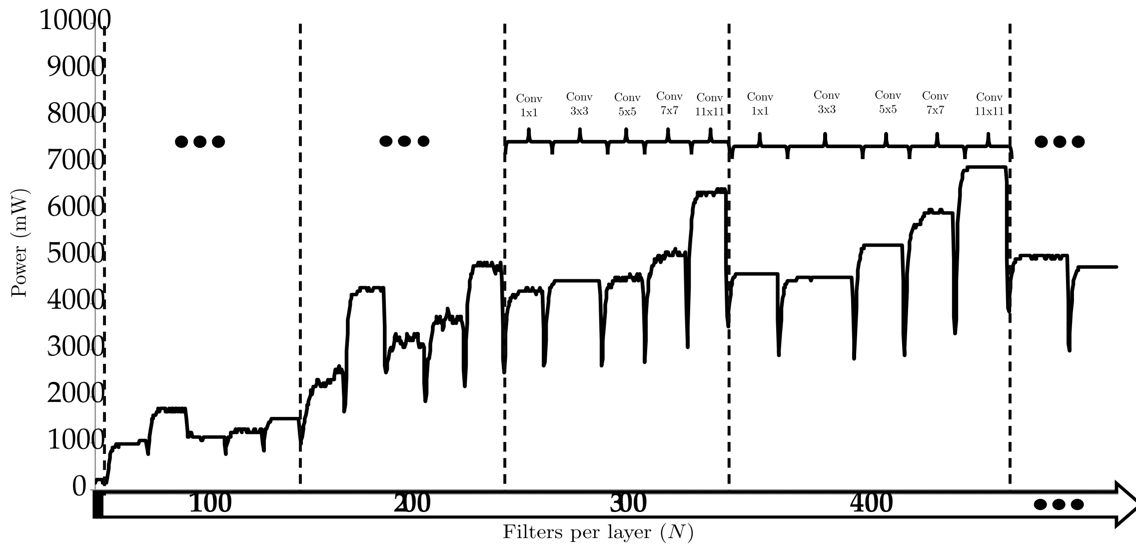


FIGURE 3.11: Power dissipation measurement over multiple experiment iterations on the Nvidia Pascal GPU architecture of the Jetson TX2 SoC. Common convolution filter sizes are iterated 5000 times on a fixed size input tensor ($224 \times 224 \times N$ for this example) with a random uniform distribution for the input and the kernel values. The number of filters N increases progressively over time and the resulting observations are averaged.

to read from the GPU power rail. Since shallow layers have less number of filter kernels, these layers take less time to be executed in this example. Therefore, high frequency sampling variations are taken into consideration for the average accumulator. The Jetson TX2 board includes multiple performance modes that vary the operating frequency of the embedded GPU to save energy. The power mode to the Max-N mode sets a GPU frequency of 1.3GHz and allowing the full use of the multi-core processor for the sampling. The software tools used to do the computation on the GPU are Pytorch [PGC+17] version 1.0 with CUDA 8.0 support for the Jetson TX2. The ImageNet pre-trained mobile CNN models are obtained from the torchvision model zoo.

On the FPGA side, it is used the Power Estimation tool® from Intel Quartus Pro Edition® version 17.1 targeting multiple convolutional task operations on the Intel® Cyclone10GX FPGA. The function synthesis is based on the DHM technique described in Section 3.5. This estimation is based on the use of logic and memory resources. As discussed in Section 3.5, DHM maps directly the functions on fully-specialized hardware, therefore its power varies rapidly with the number of processing elements mapped on the device. In Figure 3.12, it is shown an example of average power efficiency comparison between both devices. It can be seen that the FPGA has a better power efficiency that outperforms the GPU by orders of magnitude. This effect increases with the number of kernel filters on a fixed IFM. Nonetheless, this is only true as long as the design can fit on the FPGA device. Being the Cyclone10GX an embedded FPGA, this limitation is quickly met for a fixed kernel filter size and feature input map, for example, 64 filters for Conv7x7 do not fit in the device. This only allows the mapping of small functions for deeper modules or layers in a typical CNN application. This highlights the importance and interest of mobile CNNs deployment on embedded processors.

The energy is computed by integrating over time the instant power dissipation of a

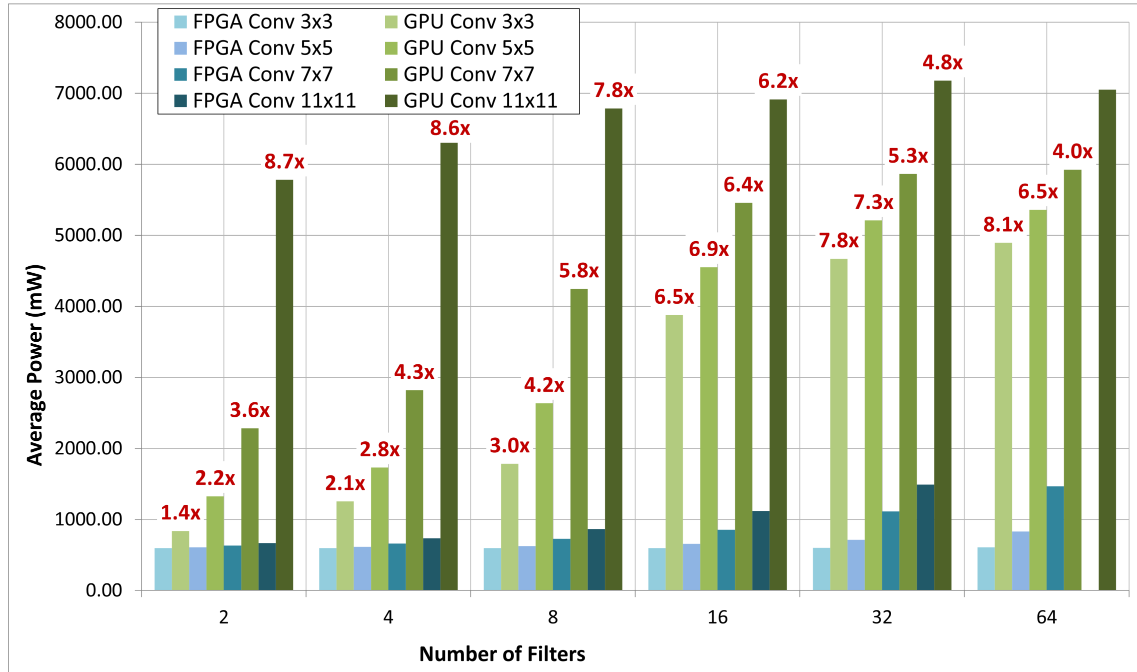


FIGURE 3.12: Average power dissipation comparison between multiple convolution function sizes on Cyclone10GX FPGA (blue) and Tegra TX2 GPU (green) for different CNN layers on an input image of 224 × 224 × 3. Blue bars represent the layers implemented on the FPGA and the green bars represent the power dissipation on the GPU. Notice that the missing blue bars represent unfeasible tasks to be mapped on more computational intensive task given the logic or memory element constrains for the FPGA device.

device and its computation consumed (Equation 2.4). Energy can thus be calculated by multiplying power and average execution latency $E(IFM) = P(IFM) \cdot LAT(IFM)$. In Figure 3.13, it can be observed that the performance in μJ difference between the FPGA and GPU implementations is increased. This is the result of having a large performance increase on both power and latency measurements. Since the energy is a product of both metrics, the resulting metric performance factor is also multiplied if they are both bigger than the unitary performance factor.

3.6.1.3 Partition throughput (KPI)

In Figure 3.14 we show a comparison in throughput between the FPGA and the GPU. As can be observed, the throughput of the GPU decreases when the layer has more kernel filters, while the efficiency of the FPGA increases as a better mapping of the task to the hardware resources is implemented. Because this performance difference increases, the efficiency factor increases as well, similarly to the other metrics. Again, this claim can only be assured as long as the design is small enough to be entirely deployed on the FPGA. The dashed red line in Figure 3.14 represents the maximum throughput for inter-device communication with the technology described in the architecture model in section 3.5. This is a mayor issue for more workload intensive tasks, since the FPGA processor spends more time in communication than in processing, creating a bottleneck and increasing latency. Throughput is an interesting metric to be considered on heterogeneous platforms,

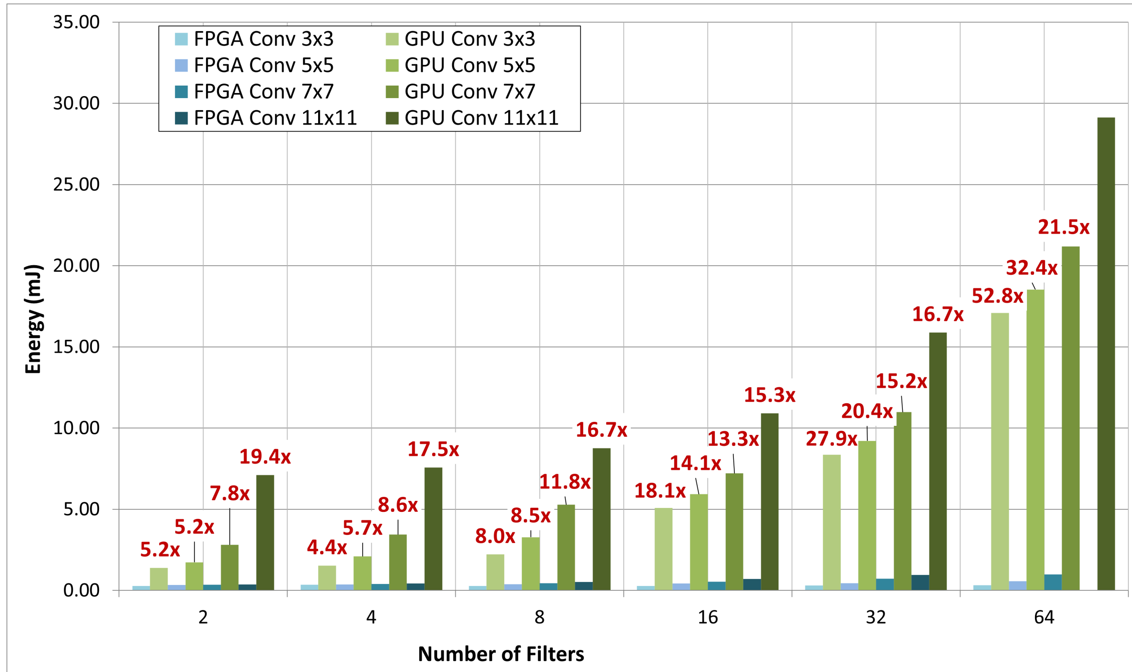


FIGURE 3.13: Energy comparison between multiple convolution function sizes on Cyclone10GX FPGA (blue) and Jetson TX2 GPU (green) for different CNN layers on an input image of $224 \times 224 \times 3$. Blue bars represent the layers implemented on the FPGA and the green bars represent the energy consumption on the GPU. The performance factor in this measure is increased result of multiplication on both power and latency metrics.

normally being one of the most constraining variables in design and benefiting from pipeling several processors. Therefore, similar to latency of PCIe, the throughput is highly dependant on the size of both IFM and IFM tensors to be sent. Since this effect may reach the theoretical maximum on higher workloads, it tends to limit real-time applications.

3.6.2 Heterogeneous partitioning results

In previous Chapter 2, we have described the heterogeneous hardware setup from Figure 2.6. Similar to the graph abstraction for software representations in Subsection 3.5.1, both the processing element nodes and the communication nodes, are treated as a black-box approach and their KPIs are measured for multiple metrics to compare the heterogeneous system. The problem definition in the following subsection discusses this procedure, combining both software model to the hardware architecture abstractions.

Given the measurements on individual devices and the module graphs for the heterogeneous platform as a result of the partitioning and scheduling, their efficiency is validated and evaluated, subject to hardware configurations described in the architecture model of Section 3.5. In order to have a fairer comparison between the monolithic homogeneous GPU-only and the heterogeneous FPGA-GPU module estimations, both were tested with the same configuration parameters and task workloads. The selected CNN models were pre-trained with ImageNet data-set, therefore all the operations respect the same format of input and OFMs dimensions.

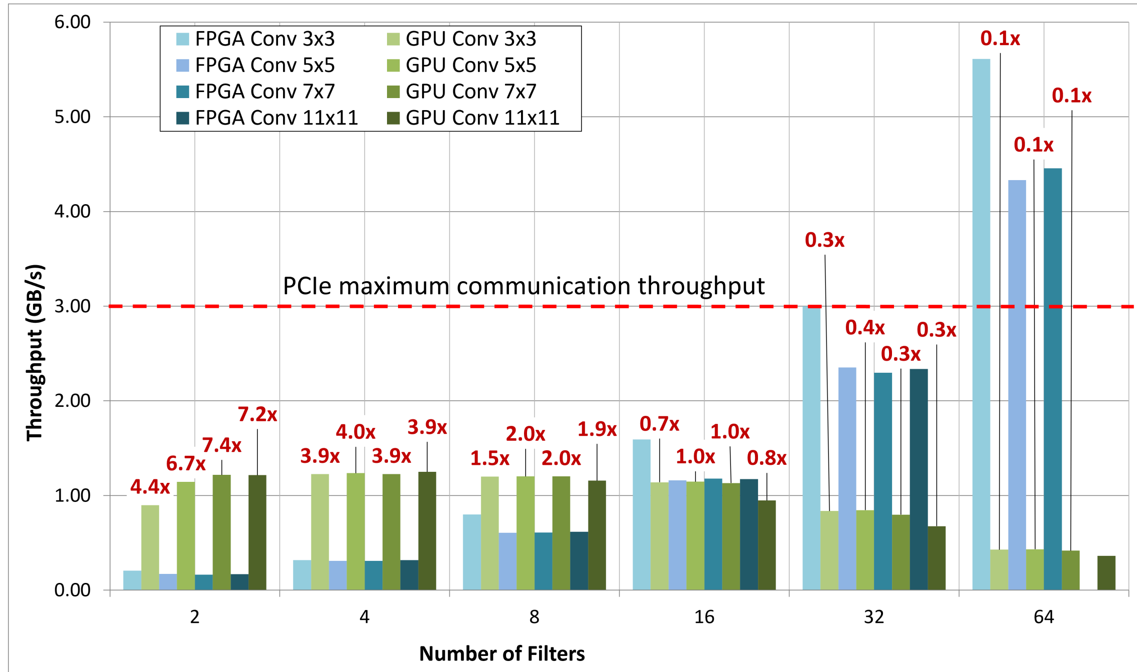


FIGURE 3.14: Throughput comparison between multiple convolution function sizes on Cyclone10GX FPGA (blue) and Jetson TX2 GPU (green) for different CNN layers on an input image of 224 × 224 × 3. Blue bars represent the layers implemented on the FPGA and the green bars represent the latency on the GPU. The red dashed line represents the maximal theoretical throughput for the PCIe communication lanes.

Table 3.2 shows the set of operations to be executed on each CNN's module described in Section 3.5.1 with its respective input and output parameters. As discussed in Section 3.5.2, because of the resource-intensive mapping technique on the custom logic, it is infeasible to fully implement CNN models on the FPGA using DHM. This is the case for the Cyclone 10 family with a maximum number of 220K logic elements. However, the overall energy consumption and latency can be extrapolated by individually analyzing single layer executions on each module. From the heterogeneous model, it is possible to estimate the performance of the hybrid platform and compare it with the homogeneous solution. Although Table 3.2 also includes the number of parameters of each module, it is needed to be mindful of the fact that computation-communication complexity actually relies on the module graph structure. There are some considerations for the validation of each module estimation that need to be mentioned.

To keep up with a sufficient module precision, the first two dimensions of the IFMs of the modules, H_I and W_I, are sampled following the typical architecture tensor sizes factors of two, i.e. 224 × 224, 112 × 112 and so on, down to 4 × 4. The C_I and N dimensions are iterated as seen in Figure 3.11, SqueezeNet [IHM+16] includes a residual connection inside its module. This tensor storage is not transferred and it is cached by the host to avoid an unnecessary communication overhead between devices. This allows us to fully exploit the dimension reduction of the IFM, as result of the GConv. A similar effect can be noticed on the module with no spatial reduction on MobileNetv2 [SHZ+18] and after the channel split on ShuffleNetv2 [MZZS18], as seen in Subsection 3.5.1 from Section 3.5. Because the

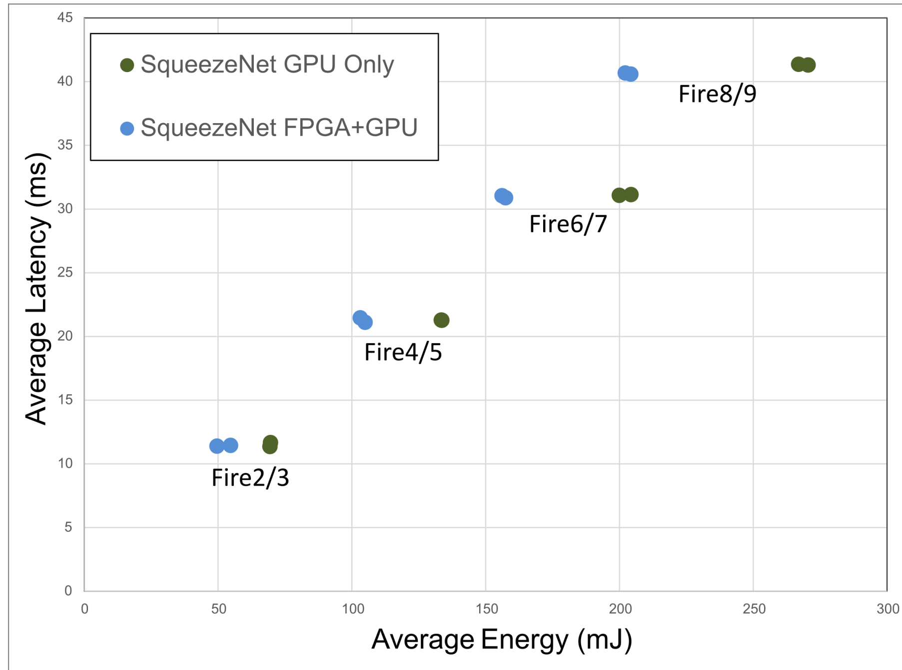


FIGURE 3.15: Average metric performance space of the tested SqueezeNet's modules with different workloads on an homogeneous GPU-only platform (green) and the FPGA-GPU heterogeneous platform (blue). x-axis represents the average energy and y-axis the average latency.

hardware setup is highly bounded by the PCIe throughput of 2.5GB/s, as observed in Figure 3.14, these observations are crucial to keep up with a good performance. In this Chapter, the sparsity on some of the modules of SqueezeNet is not considered, and since this is the only difference between some modules, they have the same performance for the analysis.

As an additional partial solution to the burden of data movement, MobileNetv2 and ShuffleNetv2 include modules with an stride of two ($s = 2$), which facilitate the objective of reducing data transfers. However, the KPIs do not take into consideration an explicit declaration of the stride variable. Nevertheless, this can be easily solved by modifying the input of each metric model by shifting both dimension, or equivalently H_I/s and W_I/s.

From Table 3.2, we can be observe a comparison between the energy (E) in mJ and latency (LAT) in ms from both the GPU that the Fire modules from SqueezeNet have a significant energy efficiency gain, up to 28%, with no significant impact on the latency, as can be observed from the metric average in Figure 3.15. This is mostly because the energy efficiency of the Conv3 $\times$ 3 task on the FPGA is higher than that on the GPU, or $E_{FPGA} << E_{GPU}$. In the case of latency, because both the time spent in communicating between devices and the processing time on the FPGA are shorter than the execution of the Conv1 $\times$ 1 task on the GPU, it is possible to hide its latency during the execution time of the GPU. This means that if $LAT_{FPGA} + LAT_{Comm} < LAT_{GPU}$, then the max function as consequence of the heterogeneous model's parallel execution, $\max(LAT_{FPGA} + LAT_{Comm}, LAT_{GPU})$, will be dominated by the execution time of the GPU. This is highly beneficial because this sub-task is small enough, thanks to the GConv, to be fully mapped on the FPGA for every Fire module on the CNN. On the other hand, when latency is

dominated by inter-device communication, the heterogeneous setup is less effective than the GPU-only counterpart. These cases cover mostly shallower modules with CNN layers processing bigger FMs. The impact of higher communication latency can be seen from example in modules Fire1, Fire5 and Bottleneck2 for instance in Table 3.2.

For the MobileNetv2 CNN, although the partition only considers a sequential execution of the diverse tasks in the Bottleneck modules, there are both an increased energy and latency performance. This is a result from the fact, that $LAT_{FPGA} + LAT_{Comm} < LAT_{GPU}$ and $E_{FPGA} < E_{GPU}$ for the DWConv 3×3 on every module configuration. This speed-up and energy efficiency factor increases with the size of the IFM as seen in Figure 3.16 up to 23% and 30%, respectively.

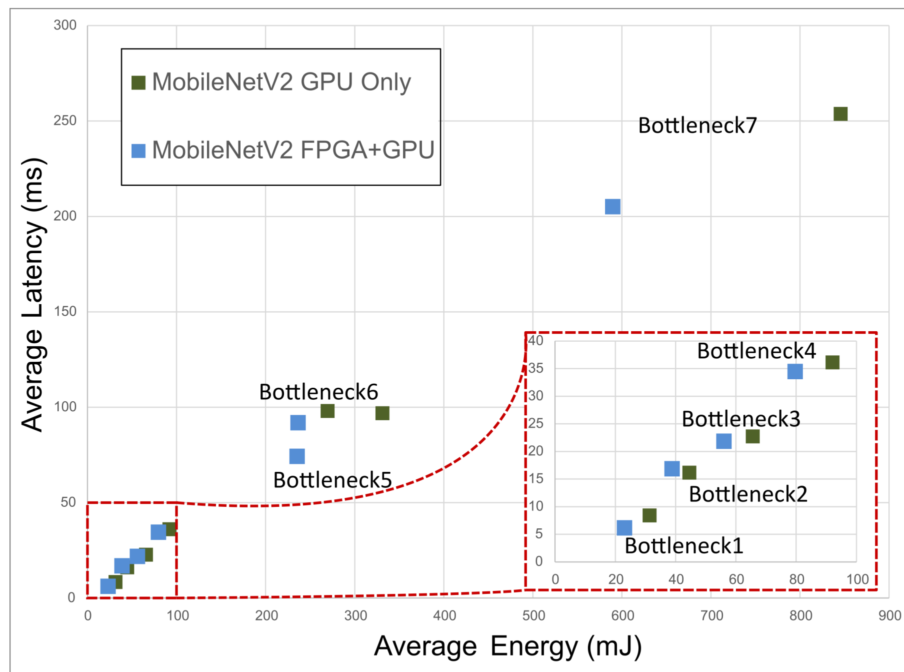


FIGURE 3.16: Average metric performance space of the measured MobileNetv2's modules with 0.5x parameters for different workloads on an homogeneous GPU-only platform (green) and the FPGA-GPU heterogeneous platform (blue). x-axis represents the average energy and y-axis the average latency. A zoomed subfigure highlights the performance of the first four modules in a more detailed way.

Combining the strategies from both previous partitioning and scheduling, ShuffleNetv2 (compressed version with $0.5\times$ parameters from [MZZS18]) profits from a speed-up factor on both module types, with and without spatial reduction from Figure 3.7. The first section of the Stage module incorporates a spatial reduction module that profits from a similar benefit of parallel execution. Therefore, the gain follows the same trend as the Fire module from SqueezeNet, but with a DWConv 3×3 instead of a traditional Conv3×3. The second section of the Stage module repeats a sequential execution with no spatial reduction. As a consequence, the result is similar to the Bottleneck modules from MobileNetv2. Because of this connection, it has the highest speed-up factor of 25% and energy efficiency of 21% compared to its homogeneous GPU counterpart as seen in Figure 3.17.

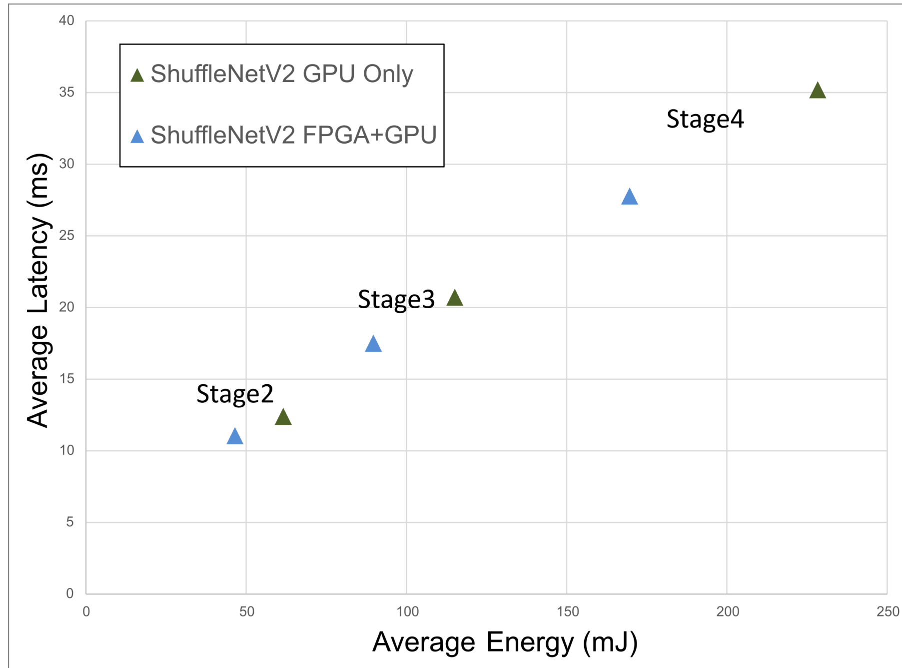


FIGURE 3.17: Average metric performance space of the measured ShuffleNetv2's modules with 0.5× parameters for different workloads on an homogeneous GPU-only platform (green) and the FPGA-GPU heterogeneous platform (blue). x-axis represents the average energy and y-axis the average latency.

Table 3.3 summarizes the speed-up factors and energy performance with selected works closest to the platform setup from Section 3.4. As covered in 3.2, different levels of heterogeneity can be exploited in state-of-the-art CNN models for partitioning. Current works focus on task partitioning like in [TST+19], where the embedded GPU is used as a feature extractor and the FPGA deploys classification with a fully-connected layer. Although, this task-partitioning solution is well suited for platforms with a considerable communication overhead, this is highly limited to small CNN models like LeNet-5 trained on MNIST dataset, where the fully-connected layer is relatively small and may under-utilize the FPGA resources. Additionally, this execution is deployed sequentially, not fully exploring parallel heterogeneous execution. On the other hand, IFM batching partitioning offers more flexibility than task partitioning as proven in [VGG+20]. This works has higher memory constrains, enabling more flexibility but with a greater design-space. Whereas this work considers memory constrains on each device, it was not tested on embedded devices, where the infeasibility of storing batches of FMs is likely to occur. This is one of the biggest limitation of why training remains a challenge for embedded devices on embedded SIMD devices. Thus, a finer granularity partitioning technique is desired in some cases. Layer-wise partitioning, like in [OHY+18], delivers this finer granularity, at the expense of evaluating each layer on each device. This solution explodes in complexity since the number of combinations grows exponentially with the number of layers and devices. Nevertheless, it allows to dedicate different workloads to setups with several devices. Module partitioning offers the flexibility of layer-wise by exploiting smaller sub-blocks irregular operations and reducing the memory or resources constrains on

embedded devices.

This Section demonstrates a similar performance to the three works discussed, showing clear heterogeneity-related gains using a module partitioning. Notice that the evaluated algorithms in this Chapter are more complex than the compared state-of-the-art partitioning techniques in terms of number of layers and parameters, but still achieving similar results. Thus, this proves that the use of heterogeneous partitioning to more complex cases can be extended. Additionally, this module-based partitioning takes into consideration the mobile-aimed nature of the tested CNN models. Therefore, because of the high parallel deployment for inference task, the use of FPGA-GPU heterogeneous embedded platforms also for mobile DL CNN topologies is justified, and shall result in very high gains if GPU and FPGA substrates are put closer to each other than in the tested multi-board setup.

3.7 Conclusions

In this Chapter, module-level partitioning and scheduling of state-of-the-art pre-trained mobile CNN architectures on a FPGA-GPU heterogeneous platform have been experimented and evaluated. Experimental results are conducted on a heterogeneous platform embedding an SoC Nvidia® Jetson TX2 CPU-GPU and an Intel® Cyclone10GX FPGA. The ShuffleNetv2, SqueezeNet and MobileNetv2 mobile-oriented CNNs are experimented. It has been shown, that heterogeneous FPGA-GPU acceleration outperforms GPU acceleration for classification inference task over SqueezeNet (21%-28% energy reduction, same latency), MobileNetv2 (12%-30% energy reduction, 4%-26% latency reduction) and ShuffleNetv2 (25% energy reduction, 21% latency reduction).

It has been also demonstrated, that an FPGA exploiting DHM outperforms GPU implementation at the cost of high resource requirements. It has been demonstrated that the considered deep learning workloads all benefit from a heterogeneous FPGA/GPU infrastructure at module-level. Indeed, the designed heterogeneous systems all outperform a homogeneous GPU solution over energy and/or latency on inference for classification tasks. These results motivate for new fully programmable architectural solutions for deep learning combining reconfigurable logic and streaming multiprocessor architectures. Following chapter considers both the modeling of Chapter 2 of each individual device and the heterogeneous partitioning techniques of this Chapter to formulate the optimization problem.

Model	Operations (N, k, s)	Module	#Parameters	Input $(H_I \times W_I \times C_I)$	Output $(H_O \times W_O \times C_O)$	GPU only		Het. FPGA+GPU	
						E	*LAT*	*E*	*LAT*
SqueezeNet	Conv1x1,N GConv Conv3x3,$N/2$ Conv1x1,$N/2$ Concat Add	Fire2	11,920	55x55x96	55x55x128	**69.4**	11.3	49.6	**11.3**
		Fire3	12,432	55x55x128	55x55x128	**69.6**	**11.6**	54.7	11.4
		Fire4	45,344	55x55x128	55x55x256	**133.6**	**21.2**	104.9	21.1
		Fire5	49,440	55x55x256	27x27x256	**133.4**	21.2	103.1	**21.4**
		Fire6	104,880	27x27x256	27x27x384	**199.9**	**31.0**	157.5	30.8
		Fire7	111,024	27x27x384	27x27x384	**204.3**	**31.1**	156.1	31.0
		Fire8	188,992	27x27x384	27x27x512	**267.0**	**41.3**	202.2	40.6
		Fire9	197,184	27x27x512	13x13x512	**270.5**	**41.2**	204.2	40.5
MobileNetV2	Conv1x1,$6N$ DWConv3x3, $6N$,$s(1,1)$ DWConv3x3, $6N$,$s(2,2)$ Conv1x1,N Concat+Add	Bottleneck1	1,824	112x112x32	112x112x16	**31.3**	**8.4**	23.0	6.1
		Bottleneck2	13,968	112x112x16	56x56x24	**44.5**	16.1	38.8	**16.8**
		Bottleneck3	39,696	56x56x24	28x28x32	**65.5**	**22.7**	56.0	21.8
		Bottleneck4	183,872	28x28x32	14x14x64	**92.0**	**36.1**	79.6	34.4
		Bottleneck5	303,168	14x14x64	14x14x96	**331.1**	**96.8**	235.3	74.2
		Bottleneck6	795,264	14x14x96	7x7x160	**269.5**	**98.0**	236.4	91.9
		Bottleneck7	886,080	7x7x160	7x7x320	**845.7**	**253.7**	589.9	205.1
ShuffleNetV2	GConv Conv1x1,$N/2$ Conv1x1,$N/2$ DWConv3x3, $N/2$,$s(1,1)$ DWConv3x3, $N/2$,$s(2,2)$ Concat Shuffle	Stage2	6,936	56x56x24	28x28x48	**61.6**	**12.3**	46.4	11.0
		Stage3	45,552	28x28x48	14x14x96	**115.0**	**20.7**	89.6	17.5
		Stage4	89,952	14x14x96	7x7x192	228.3	35.1	**169.6**	**27.7**

TABLE 3.2: Result comparison of GPU homogeneous vs FPGA-GPU heterogeneous design on ImageNet-like dataset format for several module architectures with different parameters configurations.

Work	Heterogeneous platform		Partitioning granularity	Evaluated CNN model	Energy Gain	Latency Speedup
Eun-Young Oh, et al. [OHY+18]*	GPU	Nvidia Quadro M2000	Layer-wise	Cifar10	$\sim 1.1\times - 1.9\times$	$\sim 2\times - 2.5\times$
	FPGA	Xilinx KCU 1500				
Vanishree, K. et al. [VGG+20]	GPU	Nvidia GTX750	IFM Batch-wise	Cifar10	1.15×	1.43×
	FPGA	Xilinx Virtex 7		SqueezeNet	1.15×	1.56×
Yuexuan Tu, et al. [TST+19]	CPU+GPU	Nvidia Jetson TX2	Feature extraction +Classification	LeNet5-like (N=16)	2.11×	1.3×
	FPGA	Xilinx Nexys Artix 7		LeNet5-like (N=32)	1.94×	1.19×
				LeNet5-like (N=64)	1.9×	1.17×
This work	CPU+GPU	Nvidia Jetson TX2	Module-wise	SqueezeNet's Fire	1.34×	1.01×
	FPGA	Intel Cylone 10 GX		MobileNet's v2 Bottleneck	1.55×	1.26×
				ShuffleNet's v2 Stage	1.39×	1.35×

TABLE 3.3: Energy and latency comparison with state-of-the-art partitioning techniques on heterogeneous FPGA-GPU against homogeneous implementations.
* In [OHY+18] values were estimated for a single FPGA-GPU pair.

Chapter 4

CNN Model Partitioning Optimization

4.1 Chapter abstract

CNN processing at the edge opens new challenges by requiring embedded systems to support strong computational workloads. This rapid evolution fosters more complex hardware architectures comprising interconnected heterogeneous elements. GPU, dedicated ASIC and FPGA accelerators are currently platforms of choice for porting CNNs, as their programmability and internal parallelism fit well the concurrency and the customization needs of modern CNNs. The hardware/software co-design intricacy increases when logic and memory constraints are taken into consideration concurrently in system DSE with the objective to optimize energy consumption and throughput.

In this Chapter, an automated CPU-GPU-FPGA partitioning selection is proposed for a given CNN layer. It is shown that using a Generalized Geometric Programming (GGP) optimization problem formulation, the CPU-GPU-FPGA partitioning problem can be modeled by considering a set of system performance metrics and constraints. Each metric is expressed in a posynomial form depending on CNN hyperparameters and architecture resource models. As for the partitioning method, the state-of-the-art techniques from Chapter 3 are covered, these are: tiling, grouped convolution and fused-layer; as presented in Section 3.3. The proposed analytical formalization is then employed to derive a set of objective functions and constraints as a GGP problem. It is demonstrated that it is possible to relax some problem constraints by including a penalization term, and reduce the problem to multiple simpler GP sub-problems. Experimental results targeting an embedded FPGA-GPU platform with state-of-the-art CNN layer configurations show that the simplified problem is solvable in polynomial time with a speed-up gain and energy reduction of around 20% and 15%, respectively, when compared against an arbitrary balanced partitioning. If the flydels obtained in Chapter 2 and objective functions from Chapter 3 are constrained to preserve the posynomial form and log-log convexity, it is demonstrated that GGP is an efficient optimization solution to the DSE problem.

4.2 Introduction

CNN inference deployment at the edge with high performance per Watt requires careful co-design of hardware architecture and algorithm. Systems are currently becoming more complex on both sides: architecturally and algorithmically. Embedded platforms include several asymmetrical processing elements and different levels of memory hierarchies. Thus, design solution selection from DSE requires optimization techniques to choose an appropriate partition considering available resources and performance goals. To facilitate DSE on the edge with heterogeneous platforms, an optimization problem formulation is formalized. The resulting objective and constraint functions have the form of a GGP problem. This family of optimization problems are mostly non-convex, and thus do not have a unique optimal point. However, in some cases they can be reduced and solved as GP problems. Since GP problems are fast solvable, in polynomial time, and ensure a global optimal solution, it is strongly desired to transform a GGP into a GP problem when possible. Nevertheless, this is not a trivial task, as it requires a deep expertise on the nature of the problem at hand. In this Chapter, the following contribution is presented:

Given a system performance objective and constraints, it is shown that a partitioning method of a CNN layer over a set of heterogeneous processing elements can be analytically described, expressing objective and constraints in a posynomial form for individual processing elements. A GGP optimization problem is formulated and a demonstration that the GGP equality constraints can be relaxed so as to solve the problem in the form of a set of simpler GP problems. This relaxation preserves the properties of GP problems, such as the existence of a global optimal solution and the polynomial solving time.

The Chapter is organized as follows: In Section 4.3, an analysis of related state-of-the-art works is presented, focusing on system modeling for DL, as well as on partitioning, scheduling and optimization techniques. Section 4.4 explains the context of measurement-based system performance modeling of CNN inference and introduces the concepts of monomial and posynomial. In Section 4.5, the theory behind GGP and GP is presented and explained, how a GGP problem can be relaxed to a GP problem using a penalization technique based on the condensation solution. In Section 4.6, GP optimization is applied to common CNN layer configurations and find the optimal partitions. Finally, in Section 4.7, it is discussed the results and observations.

4.3 Related Works

Since the early years of DL-oriented embedded hardware platforms, research has dedicated large efforts on partitioning machine learning efficiently over several edge devices [ZWTD19]. The partitioning solutions must take into consideration the hardware profiling, partitioning, scheduling and deployment. As explained in Section 3.3 in Chapter 3, Fused-layer is a popular technique permitting two or several layers to be mapped on a same device, reducing inter-device communication. Similarly, a set of containers, such as Docker containers, can be instantiated to a model and treat partitions as cloud

services. In [ZWTD19], a layer-wise containerization of a Deep Neural Networks (DNN) with fused-layer is proposed by using analytical regression models of different DNN configurations and optimizing analytically with dynamic programming. In [dOB19], a DNN is partitioned at a finer granularity, mapping individual neurons to different IoT devices. However, the optimization of [dOB19] is focused on reducing inter-device communication using a similar heuristic as Kernighan-Lin heuristic [KL70], swapping partition nodes in a graph abstraction. This solution is tailored to very constrained resources where communication is a dominating bottleneck. In [VGG+20], the previous use case is extended to heterogeneous platforms with different devices including CPUs, FPGAs and GPUs. Therefore, communication channels with several latencies and throughputs are considered in the optimization problem. Vanishree et al., create a Roofline analytical model to choose the appropriate batch partitioning ratio of each device the platform. In [ZBG18], data redundancy is exploited on fused-layers for contiguous partitions with the objective to reduce inter-systems communication overhead. For this purpose, the optimization process decides when to allow or when to avoid layer fusing. In the same publication [ZBG18], a discussion on partition size and communication overhead is covered. Extending the work of [ZBG18], Stahl et al., demonstrate that layer-wise partitioning can be found using ILP optimization problems considering resource constraints and minimizing communication [SZMG+19]. Finally, in [BMS+21], an assisting tool solution is introduced for embedded hardware characterization using computation and communication knowledge from heterogeneous platforms. The estimation precision is increased by introducing detailed information of the system for different CNN operations. Then, the scheduler uses a greedy layer-wise mapping as optimization strategy, selecting the most performing device iteratively for each layer. While this hardware-awareness is usually known to the designer, many internal parameters are difficult to acquire in practise or may be hidden to the designer. This solution from [BMS+21], however, does not require a performance-based measurement database generation, which in many cases may save some development time.

Same authors in [SHMG+21] add the consideration of weight-dominated CNN layers for layer fuse. Their objective functions seeks an even weight distribution on several edge devices. However, in heterogeneous systems, this may not be a desired property, since some elements are more efficient with memory access handling or embed more memory.

With respect to this state-of-the-art, the proposed method is less specific to a given deep learning solution. A resource and objective formulation are proposed for the fast optimization of multi-system CNN partitioning that combines resource constraints, performance constraints and performance objectives. The embarrassingly parallel nature of CNNs is exploited to simplify the problem formulation. The proposition is intended to be widely applicable and adaptable to a large set of CNN partitioning problems.

4.4 Flydels as monomials and posynomials formulations

From Chapter 2, it was discussed that the convolutional layer is the most common operation in CNN models, therefore the most time consuming and most parameter intensive workload [CX14]. The efforts were then focused in Chapter 2 on obtaining an analytical behavioural model for this operation on given devices, modeled together with a ReLU activation function. With the selection of the representative structural features, the computation and communication workloads were described. Afterwards, a dataset was created by stochastically exciting the heterogeneous system. Finally, a set of performance models were derived with ensemble modeling from these data points. The obtained flydels are mostly represented in a monomial or posynomial form. It is discussed why this specific form is required to be solvable with GGP in Section 4.5. In this Chapter, two KPIs are considered: processing latency (LAT) and processing energy (E). As many other physical models in electronic devices, these can represent a system in posynomial form. The same platform from Chapters 2 and 3 is used as use case. It contains one CPU-FPGA SoC and one GPU SoC. The CNN mapping to the FPGA is done through Direct Hardware Mapping (DHM) [APS+17, WDCC19], resulting in an energy-efficient but resource-hungry FPGA implementation. The results can however be extended to larger architectures and more constraints, with limited effort.

GP is a an optimization technique that is useful to solve large scale problems by formalizing them into not-too-restrictive mathematical models. The system performance models must comply with GP specific analytical formulation based on two forms of expressions. The first form to consider is the *monomial*. A monomial has the form presented in Equation 4.1:

$$u(\boldsymbol{X}) = c \prod_{i=1}^{n} x_i^{a_i} \tag{4.1}$$

Where the function $u : \mathbb{R}^n \rightarrow \mathbb{R}$ maps an input feature vector $\boldsymbol{X}$ (such as $\boldsymbol{X}$ in previous section) to a real value. c is strictly positive $c > 0$, $a_i \in \mathbb{R}$ and the domain is also strictly positive $\mathcal{D}(u) = \mathbb{R}^n_{++}$, or explained differently, the input feature vector $\boldsymbol{X}$ must be fully composed of non-zero positive real values. As a second condition, a *posynomial* is a linear combination of monomials as shown in Equation 4.2:

$$v(\boldsymbol{X}) = \sum_{k=1}^{K} c_k u(\boldsymbol{X}) = \sum_{k=1}^{K} c_k \prod_{i=1}^{n} x_{ik}^{a_{ik}} \tag{4.2}$$

Where, similarly to Equation 4.1, each element c_k is strictly positive, $c_k > 0$. To fully exploit the posynomials in the GP context, the performance system models for latency (LAT) and energy (E) must follow these rules. These performance metrics, when evaluated for CNNs, tend to fit well the GP theory. Indeed, as CNNs layers heavily parallelize, their cost in terms of energy and time tend to be proportional to the product of their structural dimensions, leading to monomial formulations. Moreover, the cost of a complete algorithm will tend to be the sum of individual layers costs, leading to posynomial formulations.

These intuitions motivate the study of GP for CNN partitioning optimization.

4.5 Optimization Problem Formulation

After selecting analytical modeling and partitioning technique, the methodology further proceeds with the formal definition of the problem formulation of a CNN layer. The solution of the optimization problem heavily depends on the nature of the objective function and constraint choice. Mathematical properties, such as, curvature and monotonicity have a critical role on the analytical or numerical tools to be exploited. Furthermore, expertise and a-priori knowledge on the problem is key to mathematically manipulate the problem and transform it from a non-convex to a convex function [BV04].

In this Section, it is described how the posynomial models of Section 4.4 are used to formulate the optimization problem as a set of GP problems. Nevertheless, because of monomial equality GP constraint violation, it is proposed a *relaxation* technique based on objective function *penalization*. The added penalization term is obtained by the transformation of the equality constraints to posynomials, this technique is known as *condensation* [RR82]. This reformulation provides a quick numerical solution in tenths of iterations, being each iteration solvable in polynomial time with interior-point methods using *CVXPY 1.0* library [ADB19]. Interior-point methods approximate a numerical solution by implicitly adding the inequality constraints to the objective function. Some mathematical transformations are applied to convert the problem to a convex and differentiable one and solvable by Newton's method. The solution converges to a set of solutions towards the optimal solution per each iteration, this trajectory is known as the *central path* [BV04].

4.5.1 GGP Formulation of the Heterogeneous CNN Layer Partitioning

GP problems are used in multiple domains for their great versatility. Because usually GP are non-convex but easily transformed into convex problems, they have gained interest in optimization formulation, since GP convex problems have been proven to be fast solvable [BV04]. The fact that the GP solution is analytically found makes it highly desirable but the formulation formulation of the problem can be difficult. The strict mathematical conditions on the objective and constraint functions require specific analytical forms. A GP problem has the form of Equation 4.3:

$$\begin{aligned} \text{minimize} \quad & f_0(\boldsymbol{X}) \\ \text{subject to} \quad & f_i(\boldsymbol{X}) \leq 1;\ i = 1, 2, \cdots, p \\ & h_i(\boldsymbol{X}) = 1;\ i = 1, 2, \cdots, m \end{aligned} \tag{4.3}$$

Where the objective function $f_0(\boldsymbol{X})$ and the inequality constraint functions $f_i(\boldsymbol{X})$ shall be posynomials (Equation 4.2 from Section 4.4), and the equality constraints $h_i(\boldsymbol{X})$ shall be monomials (Equation 4.1 from Section 4.4) [BV04]. If the problem is non-convex, it is possible to perform mathematical transformations to convert it to the convex form [BV04].

The GGP formulation of the GPU-FPGA partitioning of a CNN layer can be formalized below in Equation 4.4:

$$\begin{aligned} \text{minimize} \quad & LAT_{Het}(\boldsymbol{X}) \\ \text{subject to} \quad & ALM_F(\boldsymbol{X_F}) \leq ALM_{max} \\ & ALUT_F(\boldsymbol{X_F}) \leq ALUT_{max} \\ & LAB_F(\boldsymbol{X_F}) \leq LAB_{max} \\ & M20K_F(\boldsymbol{X_F}) \leq M20K_{max} \\ & \boldsymbol{X_F} + \boldsymbol{X_G} = \boldsymbol{X} \end{aligned} \tag{4.4}$$

Where $\boldsymbol{X}$ is the input feature vector containing the structural features from Section 2.4 in Chapter 2 of a CNN layer. These are H_I, W_I, C, k and N. $\boldsymbol{X_F}$ and $\boldsymbol{X_G}$ are the structural input feature vectors and sub sets of $\boldsymbol{X}$ to be allocated on the FPGA and GPU, respectively. $LAT_{Het}(\boldsymbol{X})$ is the posynomial model of the execution latency. $ALM_F(\boldsymbol{X_F})$, $ALUT_F(\boldsymbol{X_F})$, $LAB_F(\boldsymbol{X_F})$ and $M20K_F(\boldsymbol{X_F})$ are posynomial resource model inequalities for the target FPGA, respecting the maximum logic and memory element count of the device. In the experiments (Section 4.6), the Intel® FPGA resources were selected: ALMs, ALUTs, LABs and M20K memories.

Not only does the objective function directly depend on the taken partitioning strategy, but also the constraints last equality. In Chapter 3, three well-known partitioning techniques. With the purpose of forcing the algorithm to converge to non-trivial solutions (e.g. executing nothing), the equality constraint $\boldsymbol{X_F} + \boldsymbol{X_G} = \boldsymbol{X}$ is added. As an example, for the grouped convolution, the number of channels on the FPGA (C_F) and the number of channels on the GPU (C_G) must match the total number of channels before splitting, while keeping other features constant, or $\boldsymbol{C_F} + \boldsymbol{C_G} = \boldsymbol{C}$. Unfortunately, this equality constraint is a sum of monomials, therefore a posynomial. For GP, this definition violates the monomial equality constraint from Equation 4.3. A GP is a particular case of GGP, which accepts posynomials as equality constraints, but the GGP problem is no longer convex nor a method to find an optimal solution is known. In this case, multiple heuristics have been proposed to convert a GGP to a GP using penalization terms [Bur87]. In next Section, a condensation solution to relax the posynomial equality constraints to the given use-case is proposed.

4.5.2 GGP Penalization by Equality Constraints Condensation

As discussed in previous section, Equation 4.4 includes equality constraints that violate the conditions of GP. Nevertheless, these expressions can be transformed from posynomials to single set of monomials of the following form using the condensation technique proposed in [RR82] A new function $\bar{h}$ is computed:

$$\bar{h}(\boldsymbol{X}, \hat{\boldsymbol{X}}) = \prod_{t=1}^{T} \left[\frac{u_t(\boldsymbol{X})}{\epsilon_t(\hat{\boldsymbol{X}})} \right]^{\epsilon_t(\hat{\boldsymbol{X}})} \tag{4.5}$$

Where $\hat{X}$ is a feasible solution starting point. For example, an *a-priori* solution $\hat{X}$ can be a naive partitioning that respects the problem constraints. $u_t, t \in T$ are each of the T monomial terms (Equation 4.1) in all posynomial equality constraints h_i (in the form v of Equation 4.2). Finally, the exponent $\epsilon_t(\hat{X}) = \frac{u_t(\hat{X})}{h_i(\hat{X})}$ is the result of the *arithmetic-geometric inequality* approximation, transforming any posynomial to a monomial. Since ϵ_t is a ratio, then the basic property $\bar{h}(X, \hat{X}) \leq h(X)$ of the arithmetic-geometric inequality applies. $\bar{h}$ is known as a posynomial condensed to a monomial. Notice that, because $\bar{h}$ has all the properties of a monomial, these constraints cab be relaxed by including a penalization term in the objective function, while keeping the posynomial conditions and changing the equality to inequality constraints:

$$\begin{aligned} \text{minimize} \quad & f_0(X) + \sum_{k=1}^{m} \alpha_k \bar{h}_k(X, \hat{X})^{-1} \\ \text{subject to} \quad & f_i(X) \leq 1;\ i = 1, 2, \cdots, p \\ & h_i(X) \leq 1;\ i = 1, 2, \cdots, m \end{aligned} \tag{4.6}$$

Where each α_k is the penalization weight of the original objective function. Note that these weights add parameters to be tuned in the optimization process. Now f_i and h_i constraint functions are both posynomial inequalities, as a result of the relaxation by condensation. Although now the problem is GP-compliant, a new problem arises. The penalization weights α_k must be carefully selected for each equality constraint. For this purpose, several heuristics exist [Bur87]. In simple cases, α_k selection can be solved numerically by incrementally changing the weight values, this process is known as *tightening* [RR82]. When the tightened condensed constraints approximate the desired value, the solution is accepted, and that value for α_k is selected. This technique is similar to other *primal-dual optimization problems (duality)*, like Lagragian-based optimization techniques [BV04]. Where, in order to solve the primal formulation of a possibly non-convex problem without any modification, the dual problem must be first formulated. These dual optimization problems incorporate, after some manipulation or transformation, the constraints into the objective function as penalization. For dual problem formulation, the posynomial condensation into a monomial from the penalization is based on the result of an approximation using the algebraic manipulation of the arithmetic-geometric inequality. Solving the dual problem is more tractable than solving the primal problem [BKVH07].

In the following Section 4.6, we empirically demonstrate that it is possible to find the penalization parameters within a few iterations for different CNN layer configurations. Since each iteration is a GP problem, it is solvable in polynomial time. Thus, the final solution remains upper-bounded by the polynomial time complexity for the selected partitioning techniques of Chapter 3.

4.6 Experimental results

In this Section, the results of the optimization problem definition of Section 4.5 with different CNN layer configurations are presented. The first results presented here consider

a platform with only an embedded FPGA with a GPU, afterwards the objective function is easily modified to include more compute elements with different communication links between a CPU-GPU-FPGA platform. This is also the case for the constraints functions, which can be extended to include other embedded devices constraints.

4.6.1 Single layer optimization

Exploiting the heterogeneity of an embedded platform with a single FPGA-GPU coupling and a single communication link, an optimization problem is formulated in the form of Equation 4.7 with the models of Section 4.4 and the objective function of Chapter 3. Figure 4.1 depicts the setup test for allocation of the obtained partitions on a single CNN layer. This setup considers a sequential execution of the FPGA and GPU workloads to minimize the latency of an individual CNN layer. That is, the objective function is the sum of processing times on each device, considering the transfer time of the intermediate FMs from one node to the other (LAT_{Comm}). Since the host shares the same memory between the CPU and GPU, but not for theFPGA (Appendix B); the FM with the structural features $\boldsymbol{X_F}$ must be transferred to the FPGA accelerator. Therefore, the communication latency depends on the shape of this partitioned IFM, or $LAT_{Comm}(\boldsymbol{X_F})$.

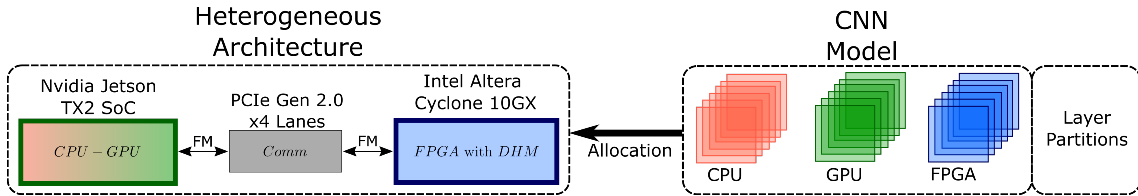


FIGURE 4.1: **Setup 1:** Single layer setup of a single CNN layer allocation.

In Equation 4.7, the selected partitioning technique is the grouped convolution. Therefore, the equality constraint only considers the channels on each device (C_F and C_G) to match the total number of channels on the layer (C).

$$\begin{aligned}
\text{minimize} \quad & LAT_F(\boldsymbol{X_F}) + LAT_G(\boldsymbol{X_G}) + LAT_{Comm}(\boldsymbol{X_F}) \\
\text{subject to} \quad & ALM_F(\boldsymbol{X_F}) \leq ALM_{max} \\
& ALUT_F(\boldsymbol{X_F}) \leq ALUT_{max} \\
& LAB_F(\boldsymbol{X_F}) \leq LAB_{max} \\
& M20K_F(\boldsymbol{X_F}) \leq M20K_{max} \\
& C_F + C_G = C
\end{aligned} \tag{4.7}$$

As discussed in Section 4.5, the posynomial equality constraint violates the requirement for a GP solution. However, the constraint can be relaxed to an inequality by adding the penalization term based on the condensed posynomial. By incorporating this penalization term, the latency objective function increases smoothly when the grouped convolution constraints are not met. This is, when the solution chosen from the optimization problem does not compute the number of channels in the convolution layer. If the equality constraints are relaxed to inequality constraints, some solutions that were not originally

feasible are now possible, and even optimal. In the case of the grouped convolution, if a constraint *equality* is relaxed to a *less or equal*, the obvious and most trivial solution to minimize the latency, would simply be processing the fewest number of IFM channels as possible. Namely, processing one single channel on the FPGA ($C_F = 1$) and one on the GPU ($C_G = 1$). Evidently, this is not a desired solution, since the IFM tensor is not fully processed. First, for this purpose, the condensed penalization term $\bar{h}(C_F, C_G, \hat{C_F}, \hat{C_G})$ must be obtained. Then, the equality constraint on the number of channels is removed. Using Equation 4.5, with $T = 2$, since the constraint consists of two monomials, the posynomial is condensed to a monomial in the following Equation 4.8:

$$\bar{h}(C_F, C_G, \hat{C_F}, \hat{C_G}) = \left[\frac{\frac{C_F}{C}}{\frac{\hat{C_F}}{\hat{C_F}+\hat{C_G}}}\right]^{\frac{\hat{C_F}}{\hat{C_F}+\hat{C_G}}} \left[\frac{\frac{C_G}{C}}{\frac{\hat{C_G}}{\hat{C_F}+\hat{C_G}}}\right]^{\frac{\hat{C_G}}{\hat{C_F}+\hat{C_G}}} \tag{4.8}$$

This penalization term is a monomial that includes the channel parameters of each device. Thus by adding it to the objective function, it preserves its posynomial form. To incorporate this term in the objective function, it is only necessary to add the penalization with a penalization weight, $\alpha\bar{h}$. As for an instance, $LAT_F(\boldsymbol{X_F}) + LAT_G(\boldsymbol{X_G}) + LAT_{Comm}(\boldsymbol{X_F}) + \alpha\bar{h}(C_F, C_G, \hat{C_F}, \hat{C_G})$. For one equality constraint condensation, only one penalization weight α is required ($m = 1$ for Equation 4.6). Consequently, it is feasible to sweep over different values of α and tighten the relaxed inequality constraint, until $\frac{C_F+C_G}{C}$ approximates the unity.

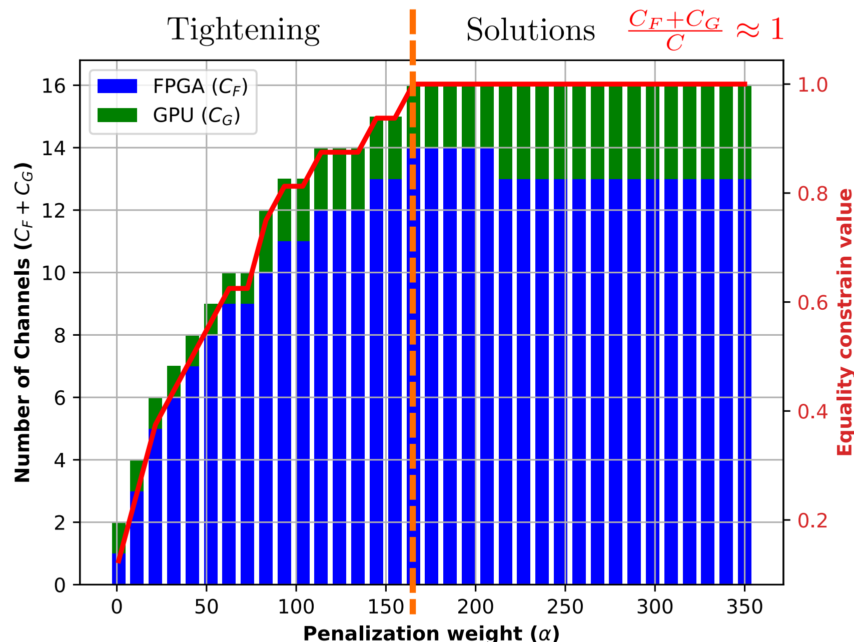


FIGURE 4.2: First iterations of a relaxed GGP sequential grouped convolution partitioning of an input tensor with 16 channels ($C = 16$) with an increasing α. The problem is solved as a set of GP problems and he tightening only takes a few iterations (iteration 17 with $\alpha = 170$) to find an acceptable solution. Each step is in polynomial time and total optimisation lasts less than a couple of hundred of milliseconds.

Figure 4.2, shows an example with an input tensor sequential processing with grouped convolution partitioning of configuration size of $H_I = W_I = 112$, $C_I = 16$, $k = 1$ and $N = 32$ for latency minimization. For a grouped convolution partitioning, all features equal for each device are kept, except for the number of channels. Resource constraints are taken into account. Each bar represents a partitioning iteration chosen by the solution of the GP solver with a given α value. The blue bars are the channels mapped on the FPGA, while the green bars are mapped to the GPU. By increasing the value of α, the constraint (red line) is tightened at each iteration, penalizing the objective function until the normalized relaxed constraint function approximates 1. For this instance, $\alpha = 170$, represented by the orange dashed line, is the first value to satisfy the constraint with $C_F = 14$ and $C_G = 2$. Additionally, for this test a balanced feasible solution is chosen for the constraint condensation. This is, the number of channels is equally distributed for both processing devices ($\hat{C_F} = \hat{C_G} = 8$). An important observation from Figure 4.2 is that multiple solutions fulfill the tightened equality constraint. Therefore, there are infinite feasible solutions after iteration 17. However, as shown in Figure 4.3, since the latency (solid purple line) has a non-decreasing monotonous nature, the following solutions perform worse than the first accepted iteration (dashed orange line). The solutions found on each iteration are found in polynomial time with respect of the inputs $\boldsymbol{X}$, $\boldsymbol{X_F}$ and $\boldsymbol{X_G}$. As a consequence, the solution of each GP problem remains bounded by polynomial time and, fixing a step size on α, so is the iterative solution.

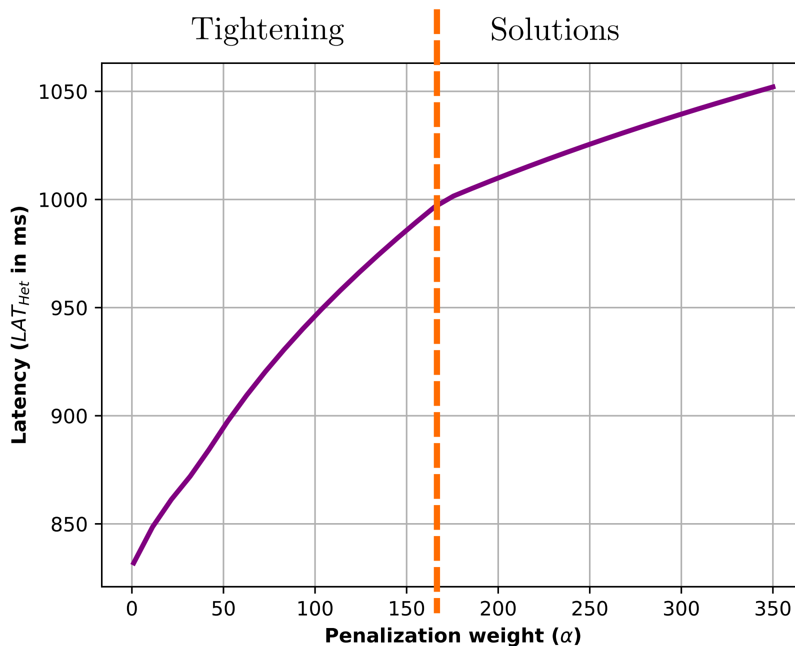


FIGURE 4.3: Heterogeneous objective function per iteration (without penalization term) from problem in Equation 4.7.

The GGP objective function can be easily modified to incorporate several computing devices with their respective communication bus linked to the other devices. Equation 4.9 shows an example of an objective function reformulation with the CPU latency model

inclusion ($LAT_C(\boldsymbol{X_C})$). Where X_C are the features of the CNN layer deployed on the CPU. Also communication links $LAT_{Comm}^{C-G}(\boldsymbol{X_{C-G}})$ and $LAT_{Comm}^{C-F}(\boldsymbol{X_{C-F}})$ are incorporated to consider the latency of the communication overhead of inter-device FM transfers between CPU and both the other devices. Additionally, the condensation and penalization term ($\bar{h}$) in the objective function must also include the features of the modified equality constrain ($C_C + C_F + C_G = C$).

$$\begin{array}{ll} \text{Computation} & LAT_C(\boldsymbol{X_C}) + LAT_F(\boldsymbol{X_F}) + LAT_G(\boldsymbol{X_G}) + \\ \text{Communication} & LAT_{Comm}^{F-G}(\boldsymbol{X_{F-G}}) + LAT_{Comm}^{C-G}(\boldsymbol{X_{C-G}}) + LAT_{Comm}^{C-F}(\boldsymbol{X_{C-F}}) + \\ \text{Penalization} & \alpha\bar{h}(C_C, C_F, C_G, \hat{C_C}, \hat{C_F}, \hat{C_G}) \end{array} \tag{4.9}$$

Figure 4.4 shows an example of a partitioning by relaxation and tightening over the same channel-wise layer partitioning of the Figure 4.2. The red bars represent the partition hosted in the CPU, the green bars those on the GPU and finally, the blue bars represent the channels . The found solution fits the partitions mostly on the FPGA, until the equality constraint is tight enough to fulfill the desired value ($C = 16$).

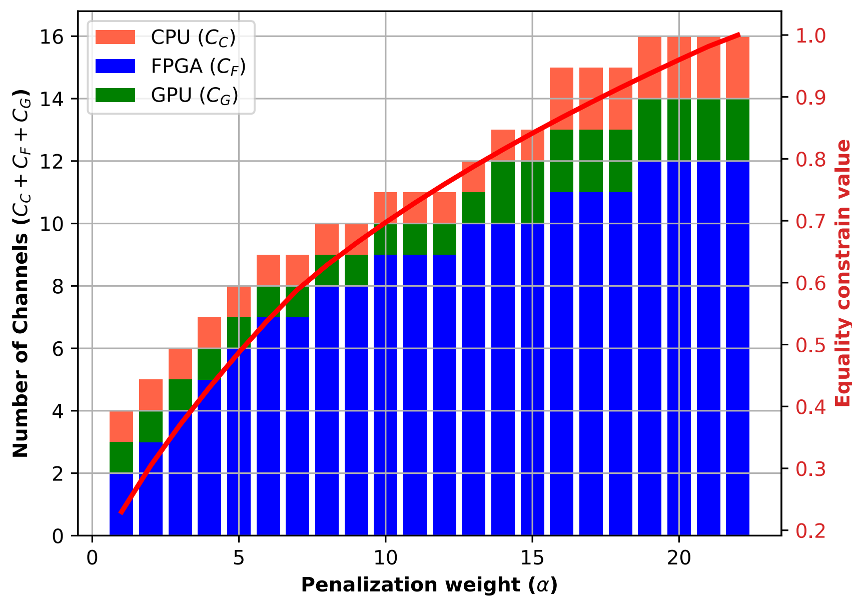


FIGURE 4.4: Relaxed GGP sequential grouped convolution partitioning of an input tensor with 16 channels ($C = 16$) with an increasing α over a CPU-GPU-FPGA network.

4.6.2 Full CNN model optimization

In previous Subsection, the limited size of the problem still allowed greedy methods to solve single-layer optimization partitioning and scheduling. An approximate solution can be found by simply mapping the biggest partition to the fastest device with available resources. In this specific use-case, the FPGA dominates in both execution time and energy

consumption. Therefore, GGP will find an optimal solution similar to greedy algorithms. However, this claim does not take into consideration the complexity of partition mapping in deeper layers. If the algorithm chooses to fit the whole first layer in one device, deeper layers are not considered to be mapped on that device. Since the device has already exhausted its resources or at least until it multiplexes in time, it is not available until it finishes its scheduled workload. Unfortunately, one of the main limitations of DHM is that layers can not be multiplexed in time. Therefore, each layer on full CNN models are individually mapped theoretically on several FPGA accelerators, simulating time-multiplexing with reduced resources. This can be simulated due to the analytical modeling from Chapter 2 and the presented formulation of layer-wise optimization presented in this Section. The setup from the Figure 4.5 is analyzed as use-case scenario.

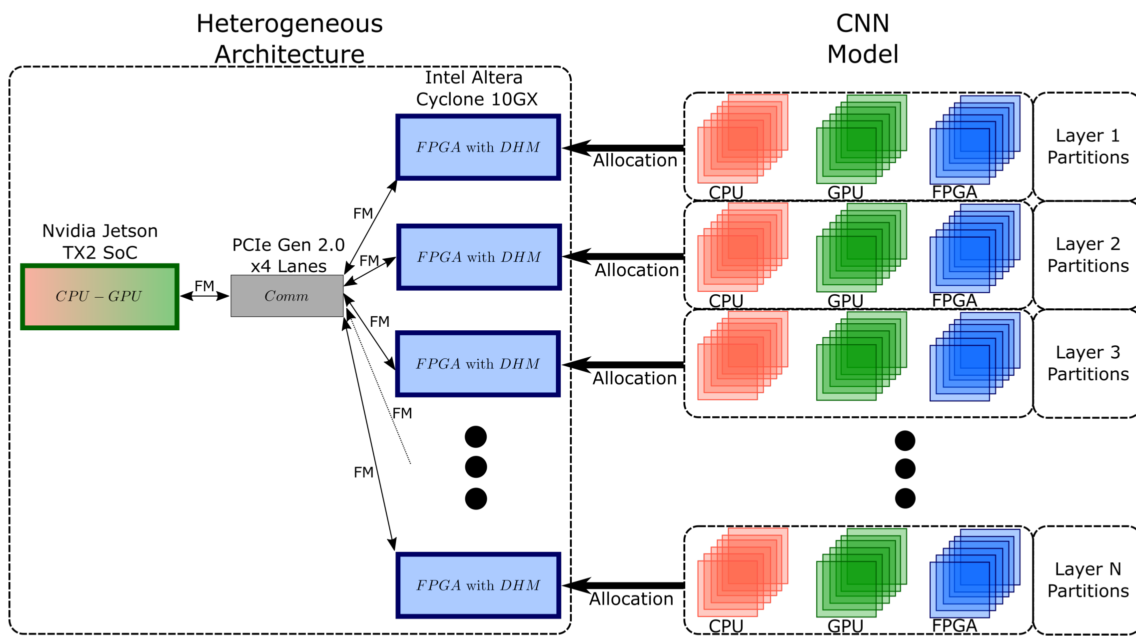


FIGURE 4.5: **Setup 2:** Multi-layer setup of a full CNN model with several layers allocated. The heterogeneous architecture is theoretically simulated.

Being L the number of convolutional layers in a CNN model, Equation 4.10 represents the optimization problem reformulation for a complete network. The objective function now adds the individual performance model (latency for this example) of each layer l ($l = 1, 2, 3, \ldots, L$). The resource inequality constraints are also modified to include the memory and computing elements utilization on each layer.

$$
\begin{aligned}
\text{minimize} \quad & \overbrace{\sum_{l=1}^{L} LAT_D(\boldsymbol{X}_D^l)}^{\text{Computation}} + \overbrace{\sum_{l=1}^{L} LAT_{Comm}(\boldsymbol{X}_D^l)}^{\text{Communication}} + \overbrace{\sum_{l=1}^{L} \alpha_l \bar{h}_l(\boldsymbol{X}_D^l, \hat{\boldsymbol{X}}_D^l)^{-1}}^{\text{Penalization}} \\
\text{subject to} \quad & \sum_{l=1}^{L} ALM_F(\boldsymbol{X}_F^l) \leq ALM_{max} \\
& \sum_{l=1}^{L} ALUT_F(\boldsymbol{X}_F^l) \leq ALUT_{max} \\
& \sum_{l=1}^{L} LAB_F(\boldsymbol{X}_F^l) \leq LAB_{max} \\
& \sum_{l=1}^{L} M20K_F(\boldsymbol{X}_F^l) \leq M20K_{max} \\
& \sum_{l=1}^{L} \boldsymbol{X}_D^l = \boldsymbol{X}^l
\end{aligned}
\tag{4.10}
$$

As covered in Appendix A, the summation and scaling (linear combination) of posynomial functions is also a posynomial. Thus, the objective and constraint functions are still posynomial and can be solved with GGP. Additionally, notice that from Equation 4.10, the number equality constraints also increases linearly with respect to the number of layers in the CNN model. Therefore, several penalization weights (α_l) and penalization function ($\bar{h}_l$) must be handled to iterative tighten the reformulated objective function. Each parameter α_l can be individually tightened as presented in previous subsection, until each constraint is fulfilled. Similarly, it is possible to increase simultaneously all the parameters on each step and individually stop each one when that specific equality constraint is fulfilled.

Although most of partitioning techniques from Chapter 3 are covered with the formulation of Equation 4.10 (tiling, grouped convolution or channel-wise loop unrolling and depth-wise separable convolution), there is still one that can not be adapted to this formulation. The fused-layer considers that some FMs are not transferred between devices, in both sequential and concurrent execution. However, in Equation 4.10, every single layer is considered to output an OFM that is intercommunicated between devices. The fused-layer technique consists in selecting which OFMs remain on the device to be computed as IFMs for the next layer, eliminating this way, the need of communication overhead. Consequently, a strategy must be chosen to reduce some terms of communication models $LAT_{Comm}(\boldsymbol{X}_D^l)$. Since communication links are usually modelled with linear functions, ILP is a simple enough solution to explore all the combinations in polynomial time [SZMG+19, SHMG+21]. In Equation 4.11, each communication term in the heterogeneous objective function from Equation 4.10 is multiplied by the Heaviside function $S(x) \in \mathbb{B}$, also known as step function. This way, the optimization technique chooses between keeping the FM in the device and skip communication ($S(x) = 0$) or to transfer the tensor ($S(x) = 1$) on each layer.

$$\text{minimize} \quad \sum_{l=1}^{L} LAT_D(\boldsymbol{X_D^l}) + \sum_{l=1}^{L} S(\boldsymbol{X_D^l}) \cdot LAT_{Comm}(\boldsymbol{X_D^l}) \tag{4.11}$$

Although the formulation from Stahl et al. [SZMG+19, SHMG+21] is simple and solvable with ILP, it does not consider the execution computation time. Therefore, this is only useful in cases where the CCR is low, result of a hardware platform heavily bounded by communication. On the other hand, while Equation 4.11, is a generalization and extension of their work it is, from a GGP perspective, unsolvable. This is because the formulation from Equation 4.11 presents several drawbacks. The most important being, that the shifted Heaviside function $H(x)$, from Equation 4.12, is not a smooth differentiable function. Thus, it can not be solvable using interior-point algorithms, that depend on iterative gradient evaluation [BKVH07].

$$H(x) = \begin{cases} 0 & x < 1 \\ 1 & x \geq 1 \end{cases} \tag{4.12}$$

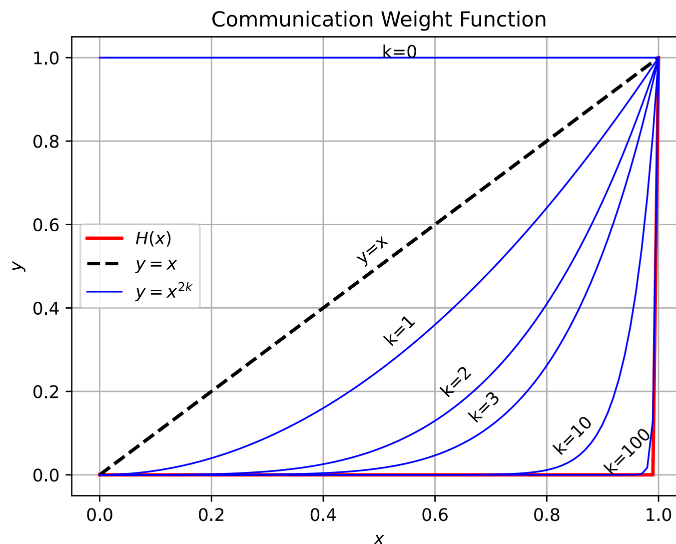


FIGURE 4.6: Communication weight function $S(x) = x^{2k}$ for different values of k.

As a consequence, analytical approximations of the step function must be used. Well-known sigmoid-like functions, such as the logistic function or trigonometric functions like *arctan* and *tanh*, are suitable candidates [BV04]. Approximation techniques such as, cubic and spline interpolations are also commonly employed to smooth and clip or bound the proposed function [AGN07]. Nevertheless, as discussed in Section 4.4, to preserve the posynomial properties, both terms in the product $S(x)$ and $LAT(\boldsymbol{X_D})$, must be also posynomials (Appendix A). Furthermore, convexity and monotonicity must be also preserved to be solvable with GGP. This restricts the number of usable functions for fused-layer formulation, since they must follow the algebraic form. Thus, in Equation 4.13, a simple exponential algebraic function is defined as communication weight function.

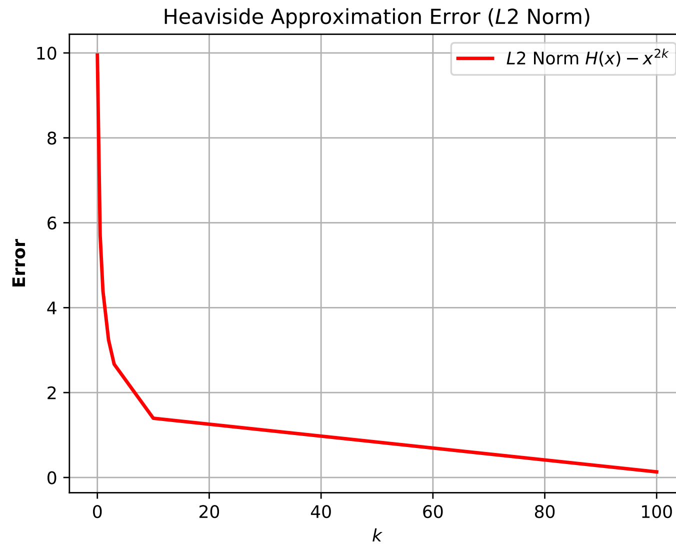


FIGURE 4.7: Approximation error function based on $L2$-Norm

$$S(x) = x^{2k}, k \in \mathbb{N} \tag{4.13}$$

Figure 4.6 shows many algebraic communication weight functions (blue solid line) for different values of $k \in \mathbb{N}$. Notice that, the bigger the value of k is, the better the approximation is to the Heaviside function $H(x)$ (red solid line). Additionally, using a symmetrical algebraic function (like with an even exponent) k can take also negative integer values. For simplicity purposes, we restrict k to take only natural numbers, $\mathbb{N}$. Another important remark is that, this is only true for the interval $0 \leq x \leq 1$. Therefore, these newly introduced interval constraints must be also considered on the full CNN optimization problem formulation.

Although, formulation from Equation 4.13 solves the approximation problem in a posynomial form that can be solved by GGP, it introduces a new heuristic value k. As shown in Figure 4.7, the choice of this value has a direct impact on the function approximation. The $L2$-Norm is chosen to visually and numerically evaluate the error difference between the Heaviside function and the communication weight function ($||H(x) - S(x)||$). For the full CNN model optimization formulation, a value of $k = 100$ is selected with an $L2$-Norm error of around 0.12. It is important to considerate that a big value of k can cause numerical issues that complicates convergence with no substantial difference between solutions.

Finally, the communication weight function $S(x)$ is included in the GGP formulation for the full model optimization. Equation 4.14 shows the modified objective function with the interval constraints of the approximation domain of $S(x)$. Since, $S(x)$ is a smooth differentiable posynomial, the weighted communication terms are still posynomials (A). Thus, this can be solved with the interior-point techniques, typical of GGP solutions. Considering that the derivative is mostly 0 for almost any value, except for values close

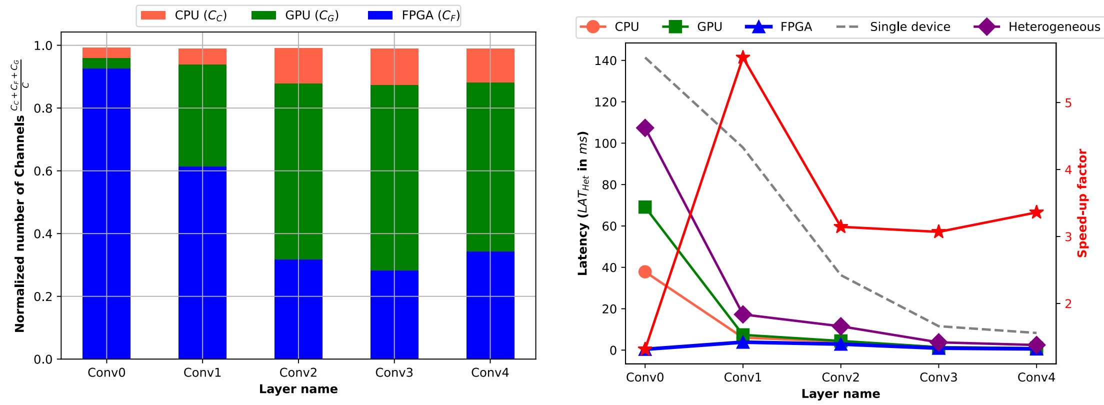
to 1, any gradient-based function is heavily penalized by choosing features around 1. Additionally, the objective function also increases with the number of $S(\boldsymbol{X}_D^l)$ for each layer that the features are transferred to another device.

$$
\begin{aligned}
\text{minimize} \quad & \overbrace{\sum_{l=1}^{L} LAT_D(\boldsymbol{X}_D^l)}^{\text{Computation}} + \overbrace{\sum_{l=1}^{L} S(\boldsymbol{X}_D^l) \cdot LAT_{Comm}(\boldsymbol{X}_D^l)}^{\text{Weighted Communication}} + \overbrace{\sum_{l=1}^{L} \alpha_l \bar{h}_l(\boldsymbol{X}_D^l, \hat{\boldsymbol{X}}_D^l)^{-1}}^{\text{Penalization}} \\
\text{subject to} \quad & \left.\begin{aligned}
& \sum_{l=1}^{L} ALM_F(\boldsymbol{X}_F^l) \leq ALM_{max} \\
& \sum_{l=1}^{L} ALUT_F(\boldsymbol{X}_F^l) \leq ALUT_{max} \\
& \sum_{l=1}^{L} LAB_F(\boldsymbol{X}_F^l) \leq LAB_{max} \\
& \sum_{l=1}^{L} M20K_F(\boldsymbol{X}_F^l) \leq M20K_{max}
\end{aligned}\right\} \text{FPGA Resource Constraints} \\
& \left.\sum_{l=1}^{L} \boldsymbol{X}_D^l = \boldsymbol{X}^l \right\} \text{Partitioning Constraints} \\
& \left. 0 \leq \frac{\boldsymbol{X}_D^l}{\boldsymbol{X}^l} \leq 1 \right\} \text{Interval Constraints}
\end{aligned}
\tag{4.14}
$$

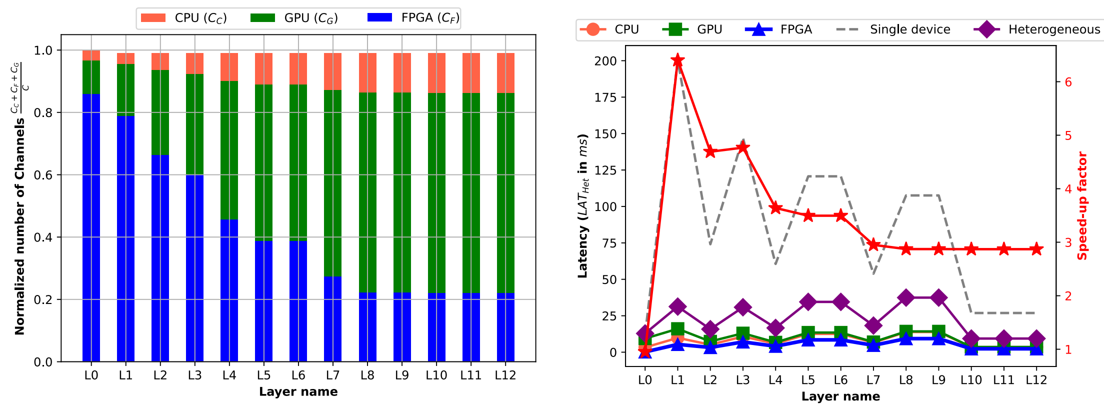
To evaluate the full model optimization partitioning, Figure 4.8 shows the resulting partitions per layer from Equation 4.14 formulation on three CNN model configurations introduced in Chapter 2. For instance, the partitioning was processed channel-wise with the heterogeneous GConv partitioning on a full model, presented in Chapter 3. For visualization purposes, the normalized number of channels computed per device is presented instead of the actual number of channels per layer. The channel layer-wise partitioning for AlexNet [KSH12] is presented in Subfigure 4.8a. Subfigure 4.8b shows the channel distribution for VGG16 [SZ14]. Finally, in Subfigure 4.8c, the resulting partitioning for ResNet18 [HZRS15] is presented. The latency of each device and the heterogeneous platform are compared against a single-device solution. The single-device is the CPU on the embedded platform with neither inter-device communication nor partitioning. This means, that the results of the dashed gray line represent the full layer execution on CPU. The solid red line represents the speed-up factor of the heterogeneous platform for each accelerated layer partition; compared against the single-device with no partition optimization nor acceleration. The model configurations are based on ImageNet dataset [DDS+09] with an input image with dimensions $3 \times 224 \times 224$. From these results it can be observed that, compared against single layer channel optimization from previous Subsection, the resource utilization on the FPGA is distributed through all layers. Instead of mapping all channels of the first layers on the custom logic, as greedy-based algorithm, GGP optimization maps channel through deeper layers. However, as seen in the three Subfigures from Figure 4.8, the GGP optimization formulation favors the first CNN layers to be directly mapped on the FPGA, which have a higher CCR. Deeper layers are bound

by the communication overhead, which is a critical part of inter-device tensor transfers on heterogeneous platforms. Similarly, as evidence of this high CCR preference scalability, the three models present a different number of layers, but also a similar result. This is, smaller partition percentages for deeper layers on FPGA accelerator and a workload dominated mostly by GPU. It is also important to mention, that in some cases a feasible solution can not be found if the resource constraints are too strict [SZMG+19, SHMG+21]. Nevertheless, because of the mathematical properties of GGP, when a solution is found, this one is the *optimal* solution.

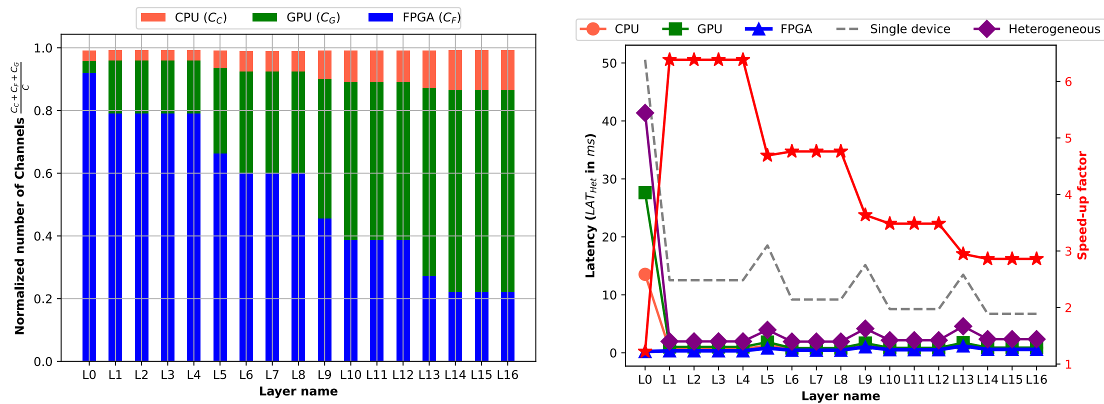
Table 4.1 displays the results from Figure 4.8 using the Setup 2 from Figure 4.5. These results are compared to state-of-the-art related works addressing partitioning optimization for CNNs models on the edge, for both simulated/theoretical and real-world case scenarios. Many approaches have been proposed for optimization formulation depending on the objective and constraint functions. Additionally, different partitioning schemes have an impact on the partition selection. Zhao et al. [ZBG18] propose a workload sharing and stealing for a Raspberry Pi 3 CPU cluster based on the memory capabilities of each node. The memory footprint and communication overhead are treated as constraints in the scheduling problem formulation, achieving a time execution speed-up from $1.7\times \sim 2.5\times$ for YOLOv2 [RF16]. As seen in Table 4.1, this technique, as many other partitioning methods, is inspired by the fused-layer from Section 3 [AFM16]. For bigger edge device networks, Oliveira and Borin [dOB19], proposed the treatment of the hardware architecture network as a graph, taking advantage this way, of graph theory and operations to modify the graph. The authors demonstrate that their technique is less effective while using greedy techniques and hand-made layer-wise partitions. The graphs include a form of communication heterogeneity by allowing different bandwidths on the WIFI link between nodes. In [dOB19], authors optimize the latency of inference by balancing the workload on LeNet [LHBB99] with a speed up of $1.8\times$. Stahl et al. [SZMG+19] present a convex ILP optimization formulation aiming for communication overhead reduction. The authors obtained from this formulation a binary selection of layers to fuse for a platform simulation. Authors test the partitioning on YOLOv2 with a speed-up of 15%. Stahl et al. [SHMG+21] extended their work for a physical platform consisting of multiple Raspberry Pis 4 on YOLOv2, AlexNet [KSH12], VGG16 [SZ14] and GoogLeNet [SLJ+14] with a speed-up factor up to $2.8\times$, $1.2\times$, $2.4\times$ and $1.7\times$; respectively Dynamic Programming (DP). With a similar hardware network on the edge, Zhou et al. [ZSB+19] proposed an unconstrained DP problem formulation that also includes the computation in the objective function achieving for YOLOv2 and VGG16a speed-up of around $1.5\times \sim 3.4\times$ and $1.1\times \sim 2.3\times$, respectively. Relaxing objective and constraint functions are a common method to accelerate solution evaluation and selection. In [ZCZ+21], Zeng et al. reduce a ILP to a LP problem by modifying some integer variables to continuous variables. Approximating this way, to a local minima solution. Additionally, in [ZCZ+21], the authors address the energy consumption of their heterogeneous platform by integrating direct energy measurements constrained to latency deadlines.



(A) AlexNet.



(B) VGG16.



(C) ResNet18.

FIGURE 4.8: Resulting GConv channel-wise optimized partitions for AlexNet 4.8a, VGG16 4.8b and ResNet 4.8c.

From Figure 4.8, it has been demonstrated that by relaxing the problem formulation, similarly to [ZCZ+21]), it is possible to obtain a speed-up gain similar to the state-of-the-art works. As many of the discussed works [ZBG18, SZMG+19, SHMG+21, ZCZ+21], the partitioning techniques include a mixture of layer-wise schemes with fused layer selection. However, the constraint function, for this chapter includes the logic and memory resources of the programmable logic of the FPGA. These considerations extend the capabilities of the partitioning optimization for hardware DSE. This custom logic awareness contrasts to other approaches for heterogeneous platforms. For instance, in the above discussed solutions, it is mostly focused on a fixed amount of memory resources, which is a common decision for CPU-GPU edge platforms. Furthermore, since not only latency can be modeled as a posynomial, but also the energy, the modification of the objective function is feasible for energy optimization. As observed in Figure 4.8, the optimization solution tends to map shallower CNN layers on the FPGA for the three models. These layers are not only the most computational intensive in terms of number of MACs, but also the tensor to communicate are lighter [dOB19]. This means, that these layers have a high CCR, which allow a suitable mapping on custom logic. However, since the communication overhead on heterogeneous systems is substantial, the first memory transfers without layer fusion introduces a considerable latency in the first layer to FPGA accelerator. Nevertheless, even considering this slow transfer the optimized heterogeneous partition solution still outperforms the single-device CPU solution. The speed-up factor is then reduced for deeper layers with values that ranges between $3.1\times \sim 5.7\times$, $2.8\times \sim 6.4\times$, and $2.8\times \sim 6.3\times$ for AlexNet, VGG16 and ResNet18, respectively.

4.7 Conclusions

This Section has proposed an automated method for CPU-GPU-FPGA partition selection of a given CNN layer. It has been shown that the partitioning problem can be modeled within the GGP framework, modeling each system performance metric in a posynomial form depending on CNN hyperparameters and architecture resource modeling. Well-known partitioning techniques in the state-of-the-art have been analyzed for layer-wise partitioning: tiling, grouped convolution, depth-wise separable convolutions and fused layers. An analytical formalization is then employed to derive a set of objective functions and constraints as a GGP problem, solvable in polynomial time without requiring a heuristic. It has been demonstrated that it is possible to relax some equality constraints by including a penalization term based on posynomial condensation, and reduce it as multiple simpler GP sub-problems. Experimental results targeting an embedded CPU-FPGA-GPU platform with state-of-the-art CNN layer configurations have demonstrated that the simplified problem is solvable in polynomial time.

Work	Edge platform	Optimization method	Partitioning Technique	Objective function	Constraints	Models	Gain
Zhao et al. [ZBG18]	6 Raspberry Pi 3 Model B (CPU)	Work stealing/sharing	Fused tile	Latency minimization	Memory footprint	YOLOv2	1.7× ∼ 3.5×)
					Communication overhead		
Oliveira and Borin [dOB19]	Up to 63 CPU-based IoT clusters	Graph partitioning and swapping	Layer-wise	Inference rate maximization	Memory footprint	LeNet	1.8×)
			Greedy	Communication minimization	Load balance		
			METIS[1]				
Stahl et al. [SZMG+19]	Simulation of up to 7 devices	ILP	Layer-wise	Communication minimization	Partitioning selection	YOLOv2	1.15×)
			Fused layer				
Zhou et al. [ZSB+19]	8 Raspberry Pi 3 Model B+ (CPU)	DP	Layer-wise	Comp/comm minimization	-	YOLOv2	1.5× ∼ 3.4×)
			Fused layer			VGG16	1.1× ∼ 2.3×)
Stahl et al. [SHMG+21]	6 Raspberry Pi 4 (CPU)	ILP	Layer-wise	Communication minimization	Partitioning selection	YOLOv2	Up to 2.8×)
						AlexNet	Up to 1.2×)
			Fused layer			VGG16	Up to 2.4×)
						GoogLeNet	Up to 1.7×)
Zeng et al. [ZCZ+21]	4 Raspberry Pi Jetson TX2 GPU Desktop CPU	Relaxed ILP to LP	Single- and Multi-layer	Energy minimization	Latency deadlines	AlexNet	0.66×)
						VGG16	0.64×)
						GoogLeNet	0.46×)
						MobileNet	0.25×)
This[2]	Jetson TX2 (CPU-GPU) + Cyclone10 GX (FPGA)	Relaxed GGP to GP	Tile-wise	Latency or Energy comp/comm minimization	Resources and partitioning	AlexNet	3.1× ∼ 5.7×)
			Channel-wise			VGG16	2.8× ∼ 6.4×)
			Fused layer			ResNet18	2.8× ∼ 6.3×)

TABLE 4.1: Optimization state-of-the-art comparison.

[1] Open-source software for automatic large graph partitioning .
[2] Using theoretical architecture from Setup 2.

Chapter 5

Conclusions and discussion

This manuscript has studied the inference deployment of CNNs models on embedded heterogeneous platforms. The algorithmic integration was specifically tailored for hybrid systems with an accelerator topology consisting of: multicore CPUs, SIMD GPUs and custom configurable logic FPGAs based on Direct Hardware Mapping (DHM). At first, the challenges and problem definition were evoked to understand the limitations and difficulties of heterogeneous design. However, during the development of this project, it has been proven that different hardware-software heterogeneity levels offer opportunities to accelerate a given set of partitions on a DL context. This was achieved by defining a three step methodology to overcome previously defined challenges consisting of *modeling*, *partitioning* and *optimization*.

The modeling in Chapter 2, covered the individual accelerator characterization. This chapter proposed a modeling method called *flydeling* to create CNN performance models by exciting the system with a random sequence using a black-box System Identification (SI) approach. For an embedded CPU-FPGA-GPU hybrid accelerator platform, it has been proven that it is possible to obtain quite accurate KPIs *flydels*. The partitioning in Chapter 3, benefited from these models to evaluate the resulting workloads. In that chapter, it has been shown that the considered DL accelerated partition workloads benefit from a heterogeneous FPGA/GPU infrastructure when using module level splitting. Furthermore, the proposed heterogeneous systems outperform a homogeneous GPU solution in terms of energy and/or latency KPIs on inference for classification tasks, showing that the performance gain compensates for communication overhead. Finally, the optimization in Chapter 4, combines the accelerated partition evaluation from Chapter 2 with the module-level partitioning rules of Chapter 3 to formulate an optimization problem. An analytical objective and constraint formalization is derived as a GGP problem. It has been demonstrated that it is possible to relax and add a penalization term to reduce the problem to more simple GP sub-problems, solvable in polynomial time.

Hardware specialized DL has been steadily evolving. Newer unconventional architectures have been constantly adopted in the embedded system domain and these hybrid systems are becoming more and more complex. As expected, during the research and writing of this thesis, this progress did not stagnate. Many opportunities have arisen with their own challenges and their own research questions. This phenomenon has opened many possible research paths that could potentially extend the scope of this project. The

author has identified the following:

- **Heterogeneity- and hardware-aware Neural Architecture Search (NAS):** In this thesis, it has been assumed that the pretrained CNN model architecture was previously chosen by the designer. This means that the optimization was elaborated over a static CNN model. Thus, the optimization was carried out in a two stage process: the model training and the partitioning result of this thesis. Nevertheless, both optimization could be performed on a single stage. NAS is a technique that generates and optimized CNN model topology for a problem. Hardware-awareness is introduced with profiling tools, in the form of the proposed analytical heterogeneous modeling. The topology evolution, training and model evaluation are simultaneously executed per iteration. This iterative improvement is usually achieved with Reinforcement Learning (RL), but this demands a high offline computation requirement.

- **DHM FPGA acceleration through partial reconfiguration:** As discussed by the authors in [APS+17], Direct Hardware Mapping (DHM) offers a high throughput with low latency at the expense of logic and memory resources. The kernel weights of CNN layers are directly mapped. Therefore, available resources are quickly exhausted and cannot be remapped during execution. This means that the communication link must be interrupted for each accelerator reprogramming for the FPGA. However, as discussed in Chapter 3, the programming time can be *hidden* in the execution, since GPU execution is order of magnitudes slower than the FPGA. Employing partial reconfiguration, communication link can be kept while remapping a different set of kernel weight values. Furthermore, this programming latency could be considered into the optimization problem formulation. Allowing to choose in-between which partitions this reprogramming could take place.

- **GPU-FPGA memory hierarchy optimization in SoC:** Many heterogeneous edge devices have already adopted shared memory hierarchies between unconventional accelerators. Some newer architecture generations from GPU vendors have placed Tensor Processing Unit (TPU) like accelerators close to the processing elements on the same silicon die. A similar design has been chosen for CPU-FPGA coupling on embedded SoCs. However, GPU-FPGA memory coupling solutions are yet to appear. Although, this may look like a pure technical drawback, it has been shown in Chapters 3 and 4, the huge importance of inter-device communication. At least from a hypothetical simulation analysis, heterogeneous systems could finally handle one of the most critical challenges. Applications with more balanced Computation-to-Communication Ratio (CCR) can be planned ahead and benefit from the appearance of this theoretical topologies.

The rise of novel unconventional architecture paradigms will always be followed by an early-phase of heterogeneous adoption with previous accelerators. Embracing this design heterogeneity in software-hardware co-design environments, will push forward innovation, unlocking newly capabilities.

Appendix A

Convexity, monomials and posynomials

A.1 Convexity definition

Definition: A function $f : \mathbb{R}^n \mapsto \mathbb{R}$ is considered *convex* over any point $x, y \in \mathbb{R}^n$ with $0 \leq \lambda \leq 1$, if :

$$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y) \tag{A.1}$$

Similarly, a function $f : \mathbb{R}^n \mapsto \mathbb{R}$ is considered *strictly convex* by restricting the inequality:

$$f(\lambda x + (1-\lambda)y) < \lambda f(x) + (1-\lambda)f(y)$$

Focusing on this definition it is possible to further derive some properties like linearity and non-linear function convexity test by composition.

A.2 Convexity properties

A.2.1 Summation

Theorem: Being F a set of convex functions $F = \{f_i : \mathbb{R}^n \mapsto \mathbb{R}, \forall i, i = 1, 2, 3, ..., m\}$, then the sum of all elements is also convex.

Proof: If we assume that function $g(f_1, f_2, ..., f_m) = \sum_{i=1}^{m} f_i$ is also convex, then by definition:

$$g(\lambda x + (1-\lambda)y) \leq \lambda g(x) + (1-\lambda)g(y)$$

$$g(\lambda x + (1-\lambda)y) \leq \lambda \sum_{i=1}^{m} f_i(x) + (1-\lambda) \sum_{i=1}^{m} f_i(y) \tag{A.2}$$

Since each single function f_i is convex, then from Equation A.1 we can algebraically rearrange:

$$\lambda \sum_{i=1}^{m} f_i(x) \geq \sum_{i=1}^{m} f_i(\lambda x + (1-\lambda)y) - (1-\lambda) \sum_{i=1}^{m} f_i(y) \tag{A.3}$$

By substituting $f_i(x)$ from Equation A.3 into Equation A.2, we obtain:

$$g(\lambda x + (1-\lambda)y) \leq \sum_{i=1}^{m} f_i(\lambda x + (1-\lambda)y) \qquad \square$$

Therefore, it has been demonstrated that function g is also convex.

A.2.2 Scaling

Theorem: Let $f : \mathbb{R}^n \mapsto \mathbb{R}$ be a convex function and α a non-negative real number, then αf is also convex.

Proof: If f is a convex function then, for $0 \leq \lambda \leq 1$, αf can be expressed as:

$$\alpha f(\lambda x + (1-\lambda)y) \leq \alpha \lambda f(x) + \alpha(1-\lambda) f(y) \tag{A.4}$$

Because α is a non-negative real number, the inequality is preserved and by rearranging terms:

$$\alpha f(\lambda x + (1-\lambda)y) \leq \lambda \alpha f(x) + (1-\lambda) \alpha f(y) \tag{A.5}$$

Being $f^* = \alpha f$ a newly introduced function then:

$$f^*(\lambda x + (1-\lambda)y) \leq \lambda f^*(x) + (1-\lambda) f^*(y) \qquad \square \tag{A.6}$$

Which follows the convexity definition from Equation A.1.

It is important to notice, that from *scaling* and *summation*, the affine or linear property for convex function can be inferred, proving that *non-negative weighted summation* is also a convex-preserving function. This property is important for Chapter 4 problem formulation.

A.3 Posynomial properties

In Chapter 4, the definition of monomial and posynomials were introduced in Equations 4.1 and 4.2 for the context of CNN performance modeling. In more general terms a monomial can be a function with the form:

$$f(x) = c \prod_{i=1}^{n} x_i^{a_i} \tag{A.7}$$

Where the function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ maps an input vector x to a real value. c is strictly positive $c > 0$, $a_i \in \mathbb{R}$ and the domain is also strictly positive $\mathcal{D}(f) = \mathbb{R}^n_{++}$. From Equation A.7, the concept of posynomial is defined as the sum of monomials:

$$f(x) = \sum_{k=1}^{K} c_k \prod_{i=1}^{n} x_{ik}^{a_{ik}} \tag{A.8}$$

As for any monomial, each element c_k in Equation A.8 is strictly positive, $c_k > 0$ for all K monomials. It is important to mention, although a posynomial can be convex, this is not necessarily case. Therefore, the definition from Section A.1 must be evaluated to determine if a posynomial is convex or concave. Additionally, some solvers execute a variable change to convert the posynomials to a convex $\log - \log$ form [BV04], like *CVXPY* [ADB19].

As covered in Chapter 4, the posynomial form is a requirement to efficiently find a solution with GP. The posynomial forms depend on the modeling approach from Chapter 2. To combine the obtained models for more complex optimization problem formulation, two main properties of posynomials forms were used. The summation and product.

A.3.1 Summation

In Chapter 4, the objective function was built with the summation of several posynomial. For this formulation to be solvable with GP, the resulting sum function must also be a posynomial.

Theorem: Being F a set of posynomial/monomial functions $F = \{f_j : \mathbb{R}^n \mapsto \mathbb{R}, \forall j, j = 1, 2, 3, ..., m\}$ as for Equations A.7 and A.8, then the sum of all terms is also a posynomial.

Proof: The resulting function $g(x)$ is defined by:

$$g(x) = \sum_{j=1}^{m} f_j \tag{A.9}$$

Since each f_j has either the form of Equation A.7 or Equation A.8, by definition, $g(x)$ can be rewritten as:

$$g(x) = \sum_{j=1}^{m} \sum_{k=1}^{K} c_{jk} \prod_{i=1}^{n} x_{ijk}^{a_{ijk}} \tag{A.10}$$

Where $K = 1$ in case an specific f_j is a monomial. By associative property, summation operands can be grouped into a single one. Therefore, the function $g(x)$ can be reformulated as:

$$g(x) = \sum_{l}^{m \times K} c_l \prod_{i=1}^{n} x_{il}^{a_{il}} \qquad \square \tag{A.11}$$

Which has the same form as Equation A.8, proving that $g(x)$ is also a posynomial by introducing a new set of $m \times K$ strictly positive coefficients c_l.

A.3.2 Product

In Chapter 2, from *flydeling*, it was assumed an *aggregation function* that combines the selected single-variable models to a multi-variable function. This function is the product of all models. For this function to result in the posynomial form, each model must also be a posynomial.

Theorem: Being F a set of posynomial/monomial functions $F = \{f_j : \mathbb{R}^n \mapsto \mathbb{R}, \forall j, j = 1, 2, 3, ..., m\}$ as for Equations A.7 and A.8, then the product of all terms is also a posynomial.

Proof: The resulting function $g(x)$ is defined by:

$$g(x) = \prod_{j=1}^{m} f_j \tag{A.12}$$

Which, similar to previous subsection, the expression can be rewritten as:

$$g(x) = \prod_{j=1}^{m} \sum_{k=1}^{K} c_{jk} \prod_{i=1}^{n} x_{ijk}^{a_{ijk}} \tag{A.13}$$

By using the commutative property of product the expression can be expanded to:

$$g(x) = \sum_{l=1}^{m \times K} \prod_{k=1}^{m} c_{lk} \prod_{i=1}^{n} x_{ilk}^{a_{ilk}} \prod_{j=1}^{n} x_{jlk}^{a_{jlk}} \cdots \tag{A.14}$$

Because it has been assumed that f_j are posynomial, it is possible to introduce a new set of strictly positive coefficients $c_l^* = \prod_{k=1}^{m} c_{lk}$ and group each independent product of the independent variables as:

$$g(x) = \sum_{l=1}^{m \times K} c_l^* \prod_{i=1}^{n^m} x_{il}^{a_{il}} \qquad \square \tag{A.15}$$

This expression has the same posynomial form as Equation A.8, proving that the product of posynomial functions is also a posynomial.

Appendix B

Heterogeneous Smart Camera Architecture Conception

DL techniques have become in the last decade the *de-facto* choice for multiple domains, achieving a similar human performance or event outperforming it in popular and well-known competitions. During this period of turbulent and continuous maturity with enough ground-breaking modifications, it has been observed high accuracy in tasks such as classification, object tracking, feature selection or detection, segmentation, input generation or input reconstruction in multiple domains like natural language processing or in vision domains like image processing and video analytic. This broad domain utility and high performance has reawakened the interest and further adoption of many scientists, researchers, developers and the industry community with enough data availability and computing power to solve a desired problem. However, this has restrained the use of heavily parameterized models of DL for embedded applications or the computing load in the edge. *Smart cameras* design has been an active research field in computer vision architectures, that has taken advantage of the current algorithmic development to further elaborate in specific and dedicated task architectures.

This appendix proposes the design of a heterogeneous smart camera architecture with a GPU and a FPGA used in Chapters 2, 3 and 4 for the DSE on multiple DL models and vision task algorithms for optimized data and task partitioning. It has been decided to name the platform **X-*MERA*** (pronounced ***CHIMERA*: C**o-processor **H**eterogeneity **I**ntegration for smart ca**MERA**s). On the first part of this appendix, the efforts are focused on the hardware interfaces and description of the platform. Then the focus is reoriented on the firmware and low-level drivers required to operate the camera. Finally, it is shown some examples from high-level software tools combining Pytorch and OpenCV to write and read tensors and deploy CNNs layers on the smart camera; using Intel Quartus® for Intel Cyclone® 10 GX FPGA and CUDA® for Nvidia Jetson TX2® GPU.

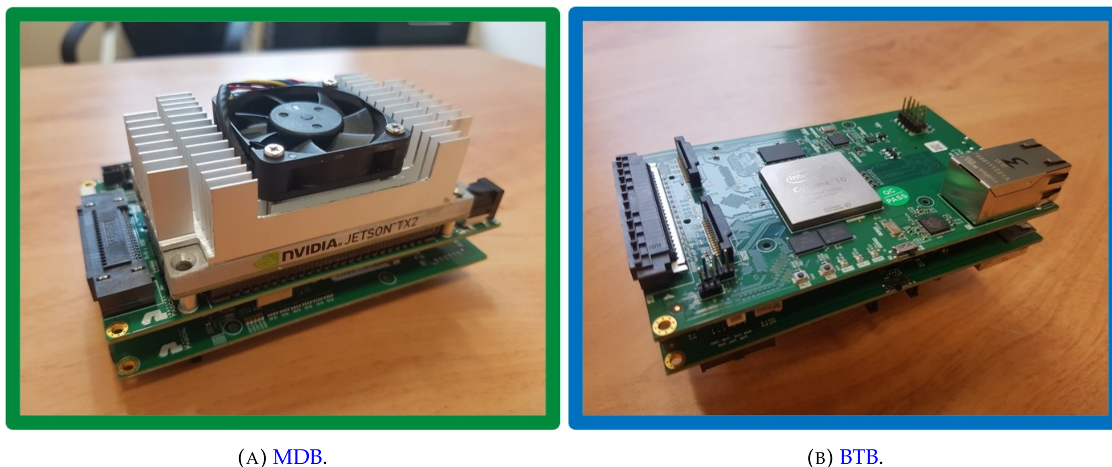
B.1 X-MERA: Co-processor Heterogeneity Integration for smart caMERAs

The purpose of this platform is to facilitate the testing of the methodologies defined in Chapters 2, 3 and 4; integrating vision applications, processing and sensors on the

edge. With this goal in mind, the platform hardware incorporates multiple interfaces to communicate with several external devices like (but not restricted to): custom camera sensors, HDMI displays, USB devices, external memories, Ethernet nodes, commercial MIPI cameras, etc. In this Section the communication and synchronization interfaces are covered.

B.1.1 Middle Board (MDB) and Bottom Board (BTB)

X-MERA is composed of two main boards: the MiDdle Board (MDB) and BoTtom Board (BTB). The MDB mostly includes the main connector to the Nvidia® Jetson TX2® System-on-Module (SoM) (detachable from the board), most of the interfaces connected to it, power rails and power supply components. While the BTB includes the Intel® Cyclone® 10 GX FPGA (soldered to the board) and most of the interfaces connected to the FPGA. Both boards have some shared interfaces, specifically the connectors that include the PCIe interface signals, power supply to the FPGA and the connector custom camera sensor at the front of the camera. Figure B.1, shows *X*-MERA MDB (B.1a) and BTB (B.1b).



(A) MDB. (B) BTB.

FIGURE B.1: *X*-MERA.

The MDB includes all the required interface interconnections for the Nvidia® Jetson TX2® SoM, including the main connector port for this module. Most important interfaces from the evaluation board are compatible on this prototype. They were tested under certain conditions and customized for establishing the communication link. The BTB, in the other hand, includes all the required interface interconnections for the Intel® Cyclone® 10 GX FPGA (10CX220YF780E5G). The interfaces for the MDB, depicted in Figure B.2 and Figure B.3 are listed below with a short description:

- **Jetson Module Connector:** A 400-pin (8x50) compatible with the Nvidia® Jetson TX2® SoM.
- **Mini HDMI:** Output connector for HDMI compatible displays and screens.
- **USB 3.0:** USB port for external devices like memories, cameras, etc.

- **Ethernet RJ45:** 10/100/1000 Ethernet port for internet connection, *ssh* remote connection to the embedded Linux OS (L4T), video streaming, etc.
- **MicroSD:** SD compatible port for memory interfacing on boot or for storage.
- **6-lanes MIPI CSI-2 x2:** 2 camera support with MIPI CSI-2 standard. Driver and Device Tree must be included on L4T kernel compilation.
- **INA3221 serial jumpers:** Selects which device has access to the Serial component INA3221 for power consumption monitoring (either the FPGA or the CPU/GPU).
- **User GPIO expander:** GPIO expander controlled by I2C serial communication from the TX2 module.
- **JTAG Debugger:** Used to debug the multicore ARM® A57 (4x) CPU.
- **MDB to BTB PCIe and serial signals connector:** The PCIe x4 signal lanes, USB signals and serial are transferred through this connector. Some enable signals also included.

Similarly, as shown in Figure B.2, the BTB interfaces are listed below:

- **Ethernet RJ45 Port:** 10/100/1000 Ethernet interface connected directly to the controller and high-speed transceivers on the FPGA.
- **JTAG programmer port:** Main programming port interface to transfer configuration bitstreams to the FPGA.
- **6-lanes MIPI CSI-2 x2:** 2 camera support with MIPI CSI-2 standard connected directly to the FPGA. Designs must be MIPI compliant.
- **Custom Camera Connector:** 200-pin connector to the custom camera sensor board. This connector on the BTB contains all the signals to the I/O ports on the FPGA and the high-speed transceivers. 6 channels of 4-lane transceivers are connected to the FPGA. All spare pinouts of the FPGA are connected here. 95 pins from different banks and 20 high-speed transceivers for a total of 115 pins.
- **USB2.0 to JTAG connector:** JTAG is the main programming interface to the FPGA. However, the user has the option to use the Arrow® USB-Programmer2 through the USB2.0 to JTAG connector. This module is based upon FT2232H and is compatible with Intel® Quartus® Programmer.
- **Programming mode selection:** This switch selects the configuration scheme. It supports fast Active Serial (AS 4x), standard AS x1, Passive Serial (PS) and Fast Passive Parallel (FPP x8). For fast AS, the BTB has a 1Gb QSPI NOR Flash memory from Micron (MT25QU01GBBB) to store programming files.

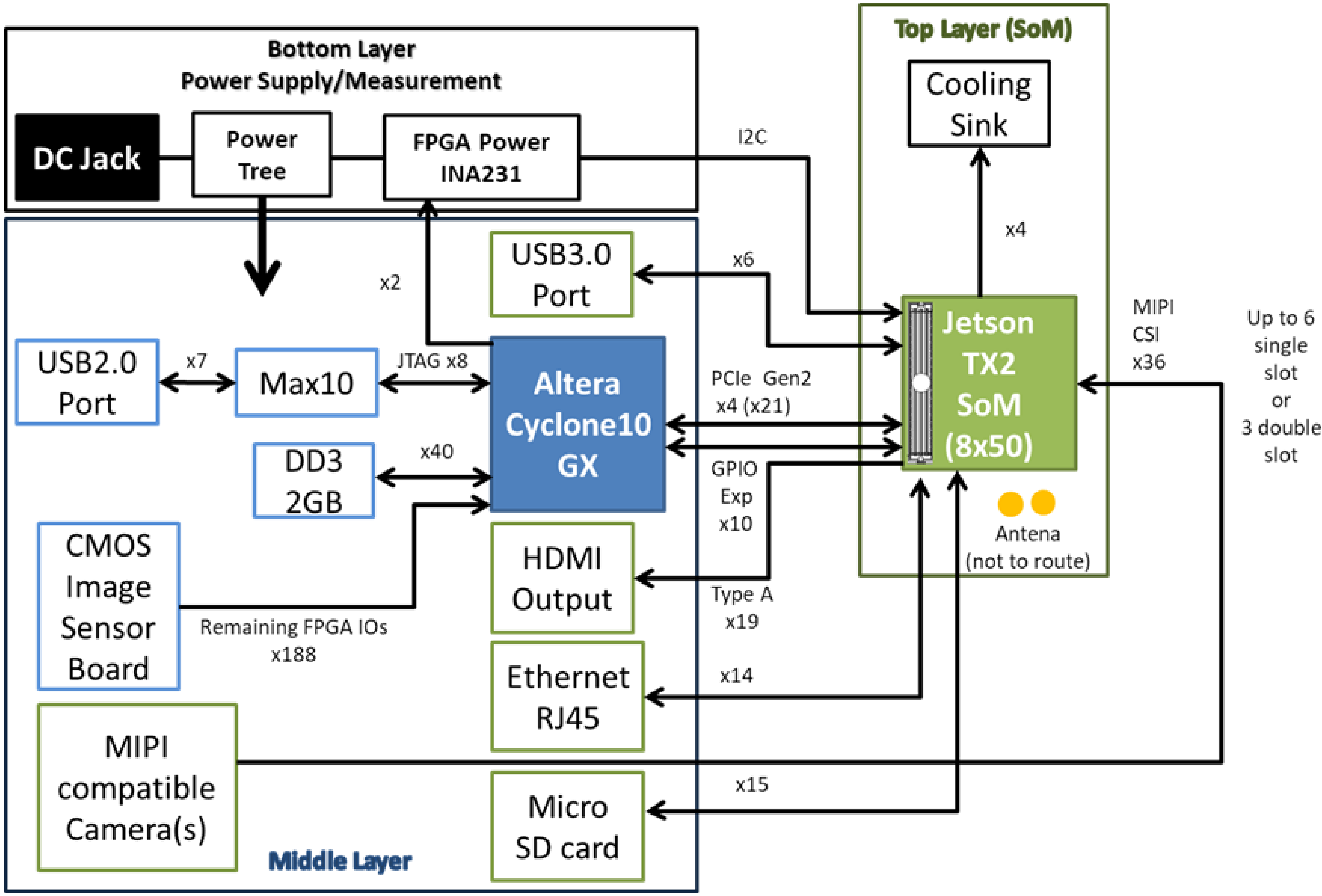


Figure B.2: X-MERA power supplies, communication interfaces and heterogeneous processors synoptic scheme.

B.1.2 Power measurements and PCIe inter-device communication

As discussed in Chapter 2, for the dataset generation, *X*-MERA counts with three I2C serial devices for power monitoring. These are the INA3221 triple-channel voltage monitors. These components are useful to measure energy performance of applications on different processors with real-time and event control or application limits. *X*-MERA can communicate to the monitors with software I2C interfaces, "*i2c-tools*" for instance, by changing low-level parameters following the vendor datasheet. A driver has been loaded for boot and set basic configuration on the device. Therefore, the serial information can sample energy performance of applications on each accelerator. Depending on the device address we can obtain information of the three channels of each one of the three INA3221. The measured rails are the following:

- For the GPU, SoC and WIFI (internal to SoM):
  - VDD_SYS_GPU
  - VDD_SYS_SOC
  - VDD_4V0_WIFI
- For the CPU, general system and DDR memory (internal to SoM):
  - VDD_IN
  - VDD_SYS_CPU
  - VDD_SYS_DDR
- For the FPGA (only for *X*-MERA):
  - C10_0V95 (FPGA core)
  - IO_1V8 (pinout)
  - 3V3_STB (board)

Figure B.3 shows the interfaces on both accelerator devices described in the two previous subsections. The power performance measurements are retrieved through the CPU/GPU SoM using serial communication. While this SoM includes MIPI interfaces for CSI cameras, the FPGA includes a configurable pinout for a custom CMOS sensor interfacing. Table B.1 shows the main three interface pinouts for the FPGA, describing the DDR memory, PCIe and CMOS image sensor communication.

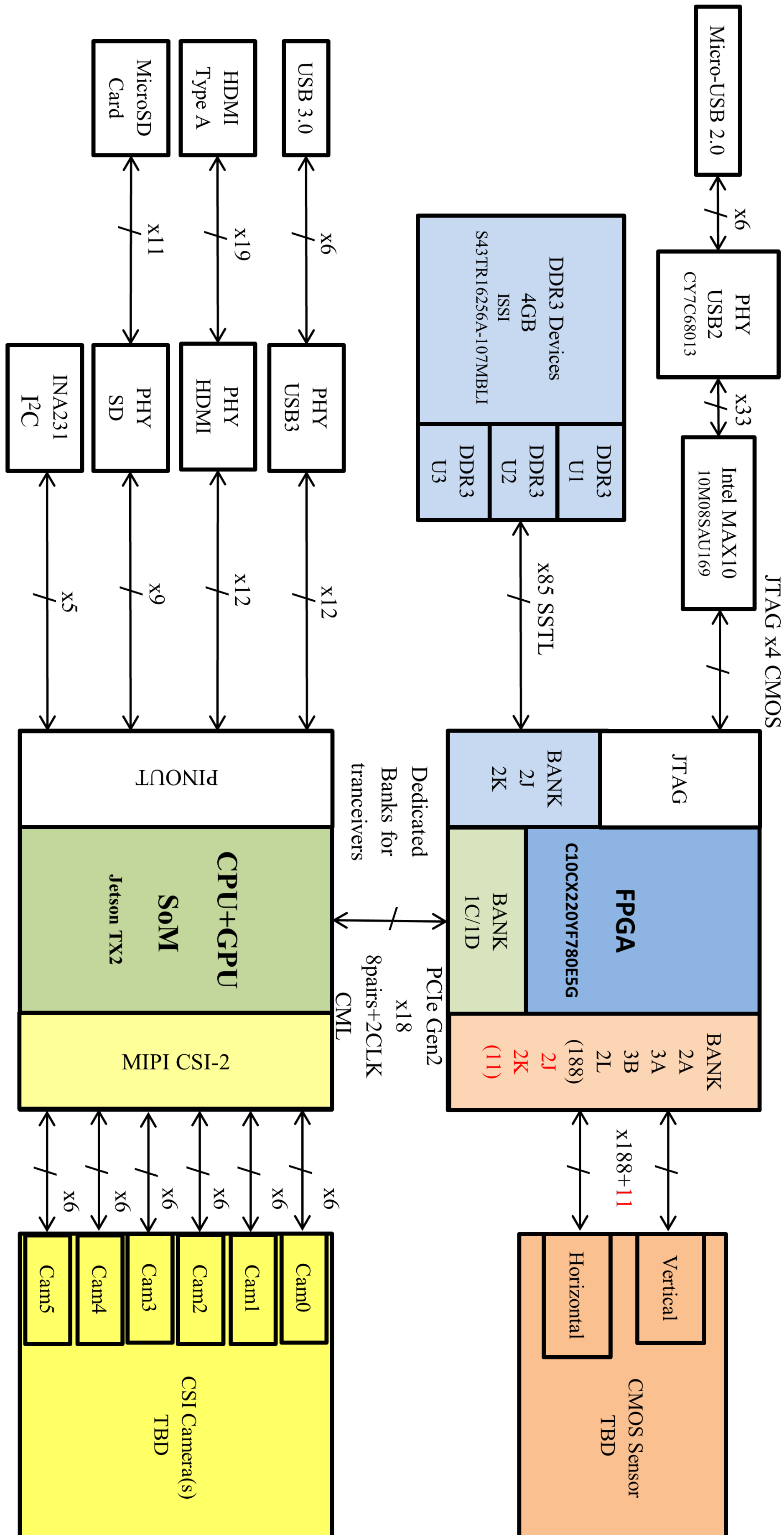


FIGURE B.3: Communication interfaces pinout.

I/O Bank[1]	I/O Type	GPIO number	Used	Direction	Free	I/O Standards	Interface
2J	LVDS	48	41	3 INPUT 28 OUTPUT 10 BIDIR	7	LVDS (1.8V) SSTL Class I (1.5V) SSTL (1.5V) Differential SSTL (1.5V)	DDR3
2K	LVDS	48	44	0 INPUT 4 OUTPUT 40 BIDIR	4	SSTL (1.5V) Differential SSTL (1.5V)	DDR3
1D	Dedicated Transceiver	28	10	6 INPUT 4 OUTPUT 0 BIDIR	18	High-Speed Differential Current Mode Logic CML	PCIe Gen2 x4
1C	Dedicated Transceiver	28	8	6 INPUT 4 OUTPUT 0 BIDIR	20	High-Speed Differential Current Mode Logic CML	PCIe Gen2 x4
2A	LVDS	48	0	TBD	48	2.5V,1.8V,1.5V, 1.2V LVCMOS	CMOS Image sensor
3A	LVDS	44	0	TBD	44	SSTL 1.2V Class I and II	CMOS Image sensor
3B	LVDS	48	0	TBD	48	1.8V,1.5V,1.2V HSTL	CMOS Image sensor
2L[2]	3.0V	48	0	TBD	48	3.0V LVTTL 3.0V	CMOS Image sensor
	Total avail.	**284**		**Total free**	**199**		
	True LVDS	**118**					

TABLE B.1: X-MERA Intel® Cyclone® 10 GX FPGA pinout table.

[1] All I/O in the same bank have the same VCC_{IO} and V_{REF}.
[2] This is the only Bank that supports 3.0V (in/out) and 2.5V (in).

B.2 Chimera library

Pytorch is a widely adopted and popular library for DL model deployment with support for different current operations, techniques and acceleration through diverse hardware platforms for Python [PGC+17]. A pretrained model zoo is also included with Torchvision software to take full advantage of the capabilities of Pytorch. Nvidia® provides a compiled Pytorch distribution targeting the Jetson family. Therefore, Pytorch is already installed in *X*-MERA. This Subsection covers how to extend the libraries to include custom C/C++/CUDA code to add support to the defined PCIe Direct Memory Access (DMA) 256 driver communication link. This will allow inter-device communication to the FPGA at a high-level programming language like Python with powerful libraries such as Pytorch.

Pytorch is built around ATen, a Tensor library that defines the basic object and operations in Torch coded in C++ used in Python. An API extension with C/C++/CUDA is therefore supported using this fundamental library to link to C/C++ drivers or applications. Additionally, this wrapper tools can be exploited to generate custom library C bindings with PCIe DMA driver implementation for FPGA communication with ATen. This binding library has been named "*chimera_lib*". The main source file "*chimera_lib.cu*" includes the following important header importing all the required utilities from ATen and Pybind11 to extend Pytorch functionalities. For this initial release, the wrapper "*chimera_lib.cu*" defines the functions using Pybind11 called directly from Pytorch. The basic functions are the following:

- **open:** Opens the FPGA device and initializes intermediate buffer variables for data transfers.
- **close:** Closes the FPGA devices and frees memory for buffers.
- **quantize:** Takes as an input a 4-dimension float Tensor (batch × channels × height × width) and as output it rounds it to the closest integer and casts it to 8bit format.
- **write:** Transfers a quantized Tensor (batch × channels × height × width) as input to FPGA device packed in a 32 bit (4 values of 8b).
- **read:** Reads from On-Chip memory a Tensor with the same size as the input Tensor (batch × channels × height × width) from the FPGA. It returns the unpacked Tensor as output.

"*chimera_lib*" is a simple, yet useful library to close the gap between high-level and low-level communication for the processing devices, specially with the complexity of FPGA design. The implementation takes advantage of the Intel® PCIe Hard IP Core and its driver to transfer pre- and post-processed tensors as FMs once the driver has been successfully installed on the (CPU/GPU) host. The FMs are the intermediate tensors of CNNs and depending on the perspective of a CNN layer, they are divided in IFMs or OFMs. For heterogeneous system evaluation, we can profit from custom logic from the FPGA to define the custom accelerator architecture. This flexibility comes with a well-known trade-off

between complexity and performance. To alleviate the burden of mapping CNN at a high-level abstraction to a low-level Hardware Description Language (HDL) implementation, Abdelouahab et al. [APS+17] have created *Delirium*, a tool to automatically generate Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) code from high-level *ONNX* model declaration based on DHM, as described in Section 3. This VHDL parser is mostly implemented with Python and also including a VHDL library for different layer types. It is designed with custom fixed precision operations in mind. Herewith, arithmetic and logic computing elements are also designed in this way. Furthermore, *ONNX* is a widely adopted tool to extend the compatibility within different DL frameworks like Tensorflow [AAB+16], Pytorch [PGC+17], Caffe2 [JSD+14] or MXNet [CLL+16]; by translating supported CNN models. Although not all operations on state-of-the-art models are supported, most used functions are automatically generated. The workflow of how *Delirium* can be integrated to generate VHDL code on *X*-MERA is shown in Figure B.5. After CNN model declaration on the DL framework, one partition or the entire model can be exported to *ONNX* and then fed to the Delirium tool to generate the VHDL files for synthesis with Intel® Quartus®.

After successful VHDL code generation with *Delirium* from a desired CNN model, implementation solution can be combined to add to the "*chimera_lib*" a custom Neural Processing Unit (NPU) based on the DHM technique from [APS+17]. As depicted in Figure B.4, the design expects a IFM tensor and processes an OFM layer partition with preloaded generated weights from Pytorch exported through *ONNX* and fed to *Delirium*.

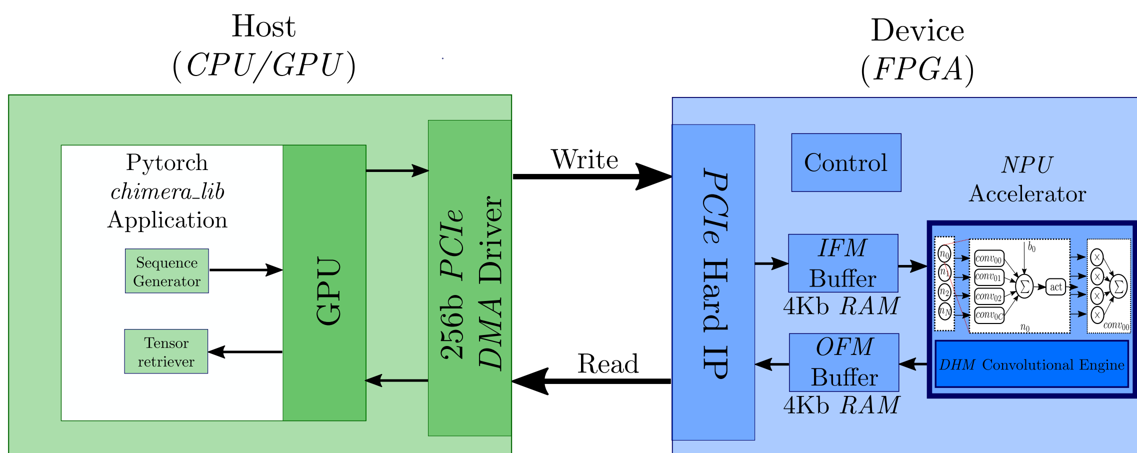


FIGURE B.4: FPGA as NPU accelerator using *Delirium*.

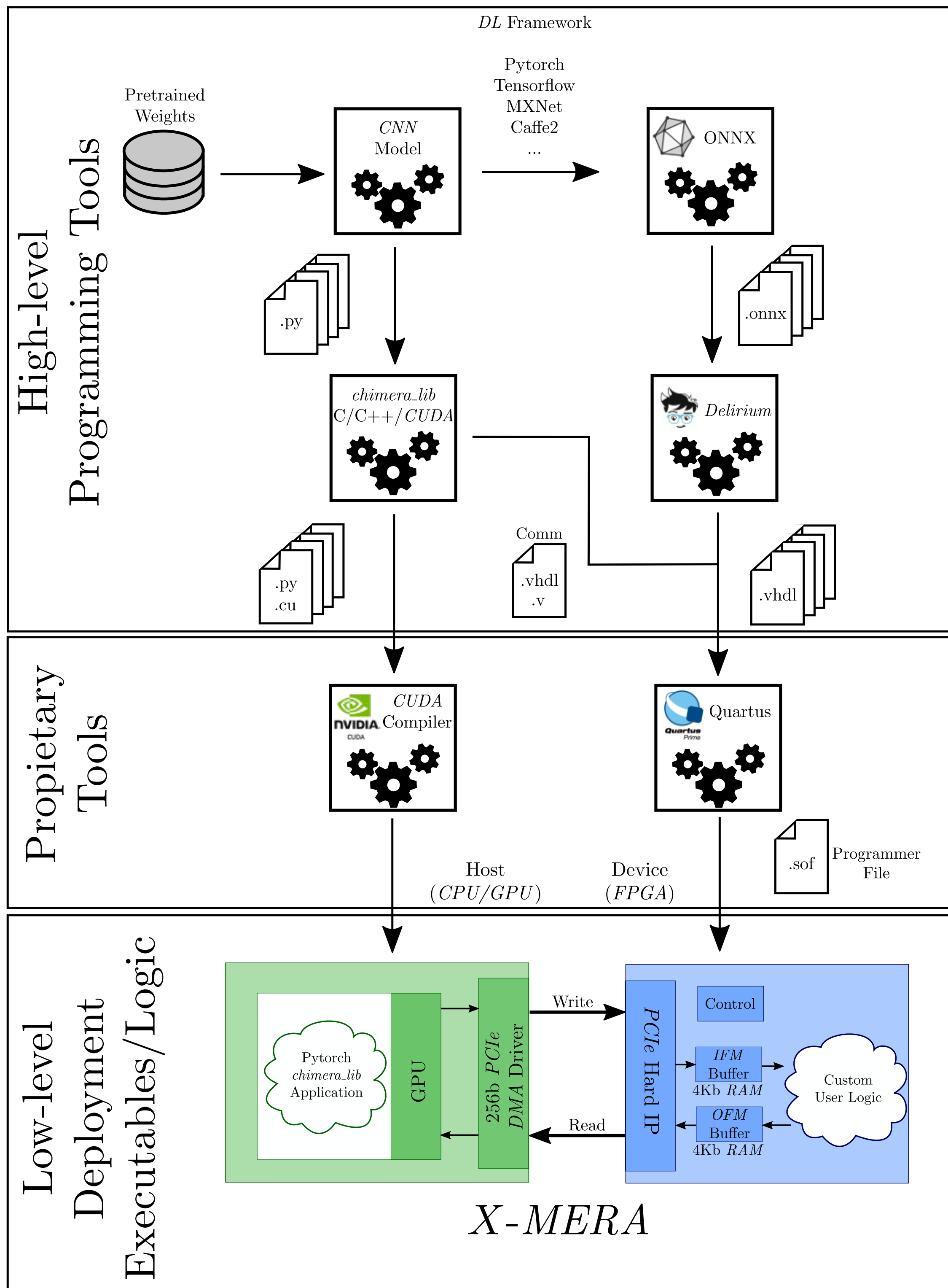


FIGURE B.5: General workflow from high-level programming tools to low-level on-device deployment of CNN models with *Delirium* and “*chimera_lib*”.

Additional deliverables and technical reports

Université Clermont-Auvergne, as part of the ACHIEVE H2020 ITN partner consortium, collaborated in the elaboration of the following deliverables and technical reports, where more details on the smart camera development can be found:

- **Deliverable D4.1:** Description of the Smart Camera Architecture.
- **Deliverable D4.2:** Survey on Digital Signal Processing Accelerators for DL Inference.
- **Deliverable D4.3:** Programming Tools for Parallel Processing Architectures.
- **Deliverable D4.4:** Smart Camera Node.
- **Deliverable D6.5:** Algorithmic Integration on Hardware.

Bibliography

[AAB+16] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, D. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *Distributed, Parallel, and Cluster Computing*, March 2016.

[ADB19] Akshay Agrawal, Steven Diamond, and Stephen Boyd. Disciplined geometric programming. *Optimization Letters*, 13(5):961–976, mar 2019.

[AFM16] M. Alwani, M. Ferdman, and P. Milder. Fused-layer CNN accelerators. *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, October 2016.

[AGN07] P. Auquiert, O. Gibaru, and E. Nyiri. On the cubic l 1 spline interpolant to the heaviside function. *Numerical Algorithms*, 46(4):321–332, dec 2007.

[AHMSNY18] S. Amiri, M. Hosseinabady, S. McIntosh-Smith, and J. Nunez-Yanez. Multi-precision convolutional neural networks on heterogeneous hardware. *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, March 2018.

[ALPP17] H. Alemdar, V. Leroy, A. ProstBoucle, and F. Pétrot. Ternary neural networks for resource-efficient ai applications. *International Joint Conference on Neural Networks*, May 2017.

[Amd67] G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS '67*, pages 482–485, April 1967.

[APS+17] K. Abdelouahab, M. Pelcat, J. Sérot, C. Bourrasset, and F. Berry. Tactics to directly map CNN graphs on embedded FPGAs. *IEEE Embedded Systems Letters*, 9(4):113–116, August 2017.

[BGCO18] Mohammed Bakiri, Christophe Guyeux, Jean-François Couchot, and Abdelkrim Kamel Oudjida. Survey on hardware implementation of random number generators on FPGA: Theory and experimental analyses. *Computer Science Review*, 27:135–153, feb 2018.

[BIM16] Robert A. Bridges, Neena Imam, and Tiffany M. Mintz. Understanding GPU power: A survey of profiling, modeling, and simulation methods. *ACM Computing Surveys*, 49(3):1–27, dec 2016.

[BKVH07] Stephen Boyd, Seung-Jean Kim, Lieven Vandenberghe, and Arash Hassibi. A tutorial on geometric programming. *Optimisation and Engineering*, 8(1):67–127, apr 2007.

[BMS+21] Paola Busia, Svetlana Minakova, Todor Stefanov, Luigi Raffo, and Paolo Meloni. ALOHA: A unified platform-aware evaluation method for CNNs execution on heterogeneous systems at the edge. *IEEE Access*, 9:133289–133308, 2021.

[BRF14] R. Bittner, E. Ruf, and A. Forin. Direct GPU/FPGA communication via PCI express. *Cluster Computing the Journal of Networks, Software Tools and Applications*, 17(2):339–348, June 2014.

[Bur87] Scott A. Burns. Generalized geometric programming with many equality constraints. *International Journal for Numerical Methods Engineering*, 24(4):725–741, apr 1987.

[BV04] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, mar 2004.

[Cas18] C. Case. Mixed precision training of neural networks. In *GPU Technology Conference (GTC) Europe*, October 2018.

[CHBPAE19] Walther Carballo-Hernández, François Berry, Maxime Pelcat, and Miguel Arias-Estrada. Towards embedded heterogeneous FPGA-GPU smart camera architectures for CNN inference. In *Proceedings of the 13th International Conference on Distributed Smart Cameras*. ACM, sep 2019.

[CHO12] Franck Cassez, René Rydhof Hansen, and Mads Chr Olesen. What is a timing anomaly? In *12th International Workshop on Worst-Case Execution Time Analysis*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012.

[Cho17] F. Chollet. Xception: Deep learning with depthwise separable convolutions. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[CHPB21] Walther Carballo-Hernández, Maxime Pelcat, and François Berry. Why is FPGA-GPU heterogeneity the best option for embedded deep neural networks? *Presented at DATE Friday Workshop on System-level Design Methods for Deep Learning on Heterogeneous Architectures (SLOHA 2021)*, February 2021.

[CHPB+22] Walther Carballo-Hernández, Maxime Pelcat, Shuvra S. Bhattacharyya, Ricardo Carmona Galán, and François Berry. Flydeling: Streamlined

performance models for hardware acceleration of CNNs through system identification. Under review for ACM Transactions on Modeling and Performance Evaluation of Computing Systems (ToMPECS), 2022.

[CHPCGB22] Walther Carballo-Hernández, Maxime Pelcat, Ricardo Carmona Galán, and François Berry. A module-level CNN partitioning method for FPGA-GPU heterogeneous acceleration. 2022.

[CHS+16] M. Courbariaux, I. Hubara, D. Soudry, R. ElYaniv, and Y. Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1. *Neural Information Processing Systems (NIPS)*, March 2016.

[CLL+14] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam. Dadiannao: A machine-learning supercomputer. *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, December 2014.

[CLL+16] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *Journal In Neural Information Processing Systems, Workshop on Machine Learning Systems*, 2016.

[CMHM10] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai. Single-chip heterogeneous computing: Does the future include custom logic , fpgas and gpgpus. *43rd Annual IEEE/ACM International Symposium on Microparchitecture*, December 2010. 2010.

[CWV+14] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. October 2014.

[CX14] Jason Cong and Bingjun Xiao. Minimizing computation in convolutional neural networks. In *Artificial Neural Networks and Machine Learning – ICANN 2014*, pages 281–290. Springer International Publishing, 2014.

[DDS+09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, jun 2009.

[DGB+20] Nicolas Derumigny, Fabian Gruber, Théophile Bastian, Christophe Guillon, Louis-Noel Pouchet, and Fabrice Rastello. From micro-ops to abstract resources: constructing a simpler cpu performance model through microbenchmarking. December 2020.

[DMM+18] D. Das, N. Mellempudi, D. Mudigere, D. Kalamkar, S. Avancha, K. Banerjee, S. Sridharan, K. Vaidyanathan, B. Kaul, E. Georganas, A. Heinecke,

P. Dubey, J. Corbal, N. Shustrov, R. Dubtsov, E. Fomenko, and V. Pirogov. Mixed precision training of convolutional neural networks using integer operations. *Sixth International Conference of Learning Representations (ICLR) 2018*, April 2018.

[dOB19] Fabíola Martins Campos de Oliveira and Edson Borin. Partitioning convolutional neural networks to maximize the inference rate on constrained IoT devices. *Future Internet 2019: Innovative Topologies and Algorithms for Neural Networks*, 11(10):209, sep 2019.

[DRPDDP15] Kaeli David R., Mistry Perhaad, Schaa Dana, and Zhang Dong Ping. *Heterogeneous Computing with OpenCL 2.0*. Elsevier Science and Technology, May 2015.

[DZW+18] X. Dai, P. Zhang, B. Wu, H. Yin, F. Sun, Y. Wang, M. Dunkhan, Y. Hu, Y. Wu, Y. Jia, P. Vajda, M. Uyttendaele, and N. K. Jha. Chamnet: Towards efficient network design through platform-aware model adaptation. *Computer Vision and Pattern Recognition (CVPR)*, 2018.

[Eec10] Lieven Eeckhout. Computer architecture performance evaluation methods. *Synthesis Lectures on Computer Architecture*, 5(1):1–145, dec 2010.

[FRW02] Y. Frayman, B. F. Rolfe, and G. I. Webb. Solving regression problems using competitive ensemble models. *Australian Joint Conference on Artificial Intelligence*, 2557:511–522, November 2002.

[GKBS20] S. Goel, R. Kedia, M. Balakrishnan, and R. Sen. INFER: Interference-aware estimation of runtime for concurrent CNN execution on DPUs. *International Conference on Field Programmable Technology (FPT)*, December 2020.

[GlLJ18] Chen Guo, Yue lan Liu, and Xuan Jiao. Study on the influence of variable stride scale change on image recognition in CNN. *Multimedia Tools and Applications*, 78(21):30027–30037, nov 2018.

[GMRRG19] Eva García-Martín, Crefeda Faviola Rodrigues, Graham Riley, and Håkan Grahn. Estimation of energy consumption in machine learning. *Journal of Parallel and Distributed Computing*, 134:75–88, dec 2019.

[Gus88] J. L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, May 1988.

[GZ12] Asghar Ghasemi and Saleh Zahediasl. Normality tests for statistical analysis: A guide for non-statisticians. *International Journal of Endocrinology and Metabolism*, 10(2):486–489, dec 2012.

[HBNY19] M. Hosseinabady, M. A. BinZainol, and J. Nunez-Yanez. Heterogeneous fpga+gpu embedded systems: Challenges and opportunities. *7th International Workshop on High Performance Energy Efficient Embedded Systems HIP3ES 2019*, April 2019.

[HSKR21] Jianwei Hao, Piyush Subedi, In Kee Kim, and Lakshmish Ramaswamy. Characterizing resource heterogeneity in edge devices for deep learning inferences. In *Proceedings of the 2021 on Systems and Network Telemetry and Analytics*. ACM, 2021.

[HTL10] K. Hung Tsoii and W. Luk. Axel: A heterogeneous cluster with fpgas and gpus. *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays, FPGA 2010*, pages 115–124, February 2010.

[HZC+17] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *Computer Vision and Pattern Recognition (CVPR)*, April 2017.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[IHM+16] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size. *Computer Vision and Pattern Recognition (CVPR)*, 2016.

[JAH+16] P. Judd, J. Albericio, T. Hetherington, T. Aamodt, and A. Moshovos. Stripes: Bit-serial deep neural network computing. *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, October 2016.

[JIP10] Haris Javaid, Aleksander Ignjatovic, and Sri Parameswaran. Fidelity metrics for estimation models. In *2010 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, nov 2010.

[JP70] Astrom; K. J. and E. Pieter. *System Identification*. 1970.

[JSD+14] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678, November 2014.

[JYP+17] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark,

Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit, 2017.

[JZK+20] Lukas Jünger, Niko Zurstraßen, Tim Kogel, Holger Keding, and Rainer Leupers. AMAIX: A generic analytical model for deep learning accelerators. In *Lecture Notes in Computer Science*, pages 36–51. Springer International Publishing, 2020.

[Kee11] Karel J. Keesman. *System Identification*. Springer London, 2011.

[KL70] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, feb 1970.

[KLKC19] H. Kwon, L. Lai, T. Krishna, and V. Chandra. Herald: Optimizing heterogeneous dnn accelerators for edge devices. *DeepAI*, 2019.

[Kri09] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, April 2009.

[KSH12] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. *Neural Information Processing Systems (NIPS)*, 2012.

[LB06] B. C. Lee and D. M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. *ACM SIGOPS Operating Systems Review*, October 2006.

[Lev44] K. Levenberg. A method for the solution of certain non-linear problems in least squares. *Quarterly of applied mathematics*, 2(2):164–168, 1944.

[LHBB99] Y. LeCun, P. Haffner, L. Bottou, and Y. Bengio. *Object Recognition with Gradient Based Learning*. Springer, 1999.

[LSGE11] J. Lin, A. Srivatsa, A. Gerstlauer, and B. L. Evans. Heterogeneous multiprocessor mapping for real-time streaming systems. *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 2011.

[LZL16] F. Li, B. Zhang, and B. Liu. Ternary weight networks. *Neural Information Processing Systems (NIPS)*, 2016.

[MAMS18] Matt Martineau, Patrick Atkinson, and Simon McIntosh-Smith. Benchmarking the NVIDIA v100 GPU and tensor cores. In *Lecture Notes in Computer Science*, pages 444–455. Springer International Publishing, 2018.

[Mar63] D. W. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the Society for Industrial and Applied Mathematics*, 11(2):431, 1963.

[MCL+18] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. NVIDIA tensor core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2018.

[MCVS20] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-Sun Seo. Performance modeling for CNN inference accelerators on FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(4):843–856, apr 2020.

[MEA+19] Jeffrey L. McKinstry, Steven K. Esser, Rathinakumar Appuswamy, Deepika Bablani, John V. Arthur, Izzet B. Yildiz, and Dharmendra S. Modha. Discovering low-precision networks close to full-precision networks for efficient inference. In *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS Edition (EMC2-NIPS)*. IEEE, 2019.

[MFS+19] H. M. Makrani, F. Farahmand, H. Sayadi, S. Bondi, S. M. P. Dinakarrao, H. Homayoun, and S. Rafatirad. Pyramid: Machine learning framework to estimate the optimal timing and resource usage of a high-level synthesis design. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, sep 2019.

[Mit18] S. Mittal. A survey of fpga-based accelerators for convolutional neural networks. *Neural Computing and Applications*, October 2018.

[MMSJS12] João Mendes-Moreira, Carlos Soares, Alípio Mário Jorge, and Jorge Freire De Sousa. Ensemble approaches for regression. *ACM Computing Surveys*, 45(1):1–40, nov 2012.

[Mov19] Movidius. *Intel Neural Compute Stick 2*. Intel, 2019.

[MV15] S. Mittal and J. Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Computing Surveys (CSUR)*, 47(4), July 2015.

[MV19] S. Mittal and S. Vaishay. A survey of techniques for optimizing deep learning on gpus. *Journal of Systems Arrchitecture*, 99, October 2019.

[MZZS18] N. Ma, X. Zhang, H. Zheng, and J. Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. *Computer Vision and Pattern Recognition (CVPR)*, July 2018.

[NLPH21] Yehya Nasser, Jordane Lorandel, Jean-Christophe Prevotet, and Maryline Helard. RTL to transistor level power modeling and estimation techniques for FPGA and ASIC: A survey. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(3):479–493, mar 2021.

[NMN+10] Hitoshi Nagasaka, Naoya Maruyama, Akira Nukada, Toshio Endo, and Satoshi Matsuoka. Statistical power modeling of GPU kernels using performance counters. In *International Conference on Green Computing*. IEEE, aug 2010.

[NSS+16] E. Nurvitadhi, D. Sheffield, J. Sim, A. Mishra, G. Venkatesh, and D. Marr. Accelerating binarized neural networks: Comparison of fpga, cpu, gpu, and asic. *2016 International Conference on Field-Programmable Technology (FPT)*, December 2016.

[OB18] K. O'Neal and P. Brisk. Predictive modeling for CPU, GPU, and FPGA performance and power consumption: A survey. In *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, July 2018.

[OBSK17] K. O'Neal, P. Brisk, E. Shriver, and M. Kishinevsky. HALWPE: Hardware-assisted light weight performance estimation for GPUs. *54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2017.

[OHY+18] E. Y. Oh, W. G Han, E. J. Yang, J. H. Jeong, L. Lemi, and C. H. Youn. Energy-efficient task partitioning for CNN-based object detection in heterogeneous computing environment. *International Conference on Information and Communication Technology Convergence (ICTC)*, 2018.

[PGC+17] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. *NIPS 2017 Workshop*, November 2017.

[Pim17] Andy D. Pimentel. Exploring exploration: A tutorial introduction to embedded systems design space exploration. *IEEE Design & Test*, 34(1):77–90, feb 2017.

[PMD+17] Maxime Pelcat, Alexandre Mercat, Karol Desnos, Luca Maggiani, Yanzhou Liu, Julien Heulot, Jean-François Nezan, Wassim Hamidouche,

Daniel Ménard, and Shuvra S Bhattacharyya. Reproducible evaluation of system efficiency with a model of architecture: From theory to practice. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(10):2050–2063, 2017.

[QDL+19] M. Qasaimeh, K. Denolf, K. Lo, J. Vissers, J. Zambreno, and Jones. P. H. Comparing energy efficiency of CPU, GPU and FPGA implementations for vision kernels. In *Design Automation Conference (DAC). The 15th IEEE International Conference on Embedded Software and Systems*, 2019.

[Ran04] A. Ranganathan. The levenberg-marquardt algorithm. *Tutorial on LM algorithm*, 11(1):101–110, June 2004.

[RF16] J. Redmon and A. Farhadi. Yolo9000: Better, faser, stronger. *Computer Vision and Pattern Recognition (CVPR 2016)*, December 2016.

[RMGV+20] Xalo Rancano, Roberto Fernandez Molanes, Carlos Gonzalez-Val, Juan J. Rodriguez-Andina, and Jose Farina. Performance evaluation of state-of-the-art edge computing devices for DNN inference. In *IECON 2020 The 46th Annual Conference of the IEEE Industrial Electronics Society*. IEEE, 2020.

[RMJ+19] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. Survey and benchmarking of machine learning accelerators. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019.

[RR82] D. H. Rountree and A. K. Rigler. A penalty treatment of equality constraints in generalized geometric programming. *Journal of Optimization Theory and Applications*, 38(2):169–178, oct 1982.

[SBD+13] B. Silva, A. Braeken, E. H. D'Hollander, A. Touhafi, J. G. Cornelis, and J. Lemeire. Comparing and combining gpu and fpga accelerators in an image processing context. *23rd International Conference on Field programmable Logic and Applications (FPL)*, September 2013.

[SDV+14] L. Struyf, S. DeBeugher, D. H. VanUytsel, F. Kanters, and T. Goedemé. The battle of the giants: a case study of gpu vs fpga optimisation for real-time image processing. *International conference on pervasive and embedded computing and communication systems (PECCS)*, 4, 2014.

[SEP+09] M. Showerman, J. Enos, A. Pant, V. Kindratenko, C. Steffen, R. Pennington, and W. Hwu. Qp: A heterogeneous multi-accelerator cluster. *10th LCI International Conference on High-Performance Clustered Computing*, March 2009.

[SFML19] A. Shahid, M. Fahad, R. R. Manumachu, and A. Lastovetsky. Energy of computing on multicore CPUs: Predictive models and energy conservation law. *International Conference on Parallel Computing Technologies*, July 2019.

[SHMG+21] Rafael Stahl, Alexander Hoffman, Daniel Mueller-Gritschneder, Andreas Gerstlauer, and Ulf Schlichtmann. DeeperThings: Fully distributed CNN inference on resource-constrained edge devices. *International Journal of Parallel Programming*, apr 2021.

[SHZ+18] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. *Computer Vision and Pattern Recognition (CVPR)*, pages 4510–4520, June 2018.

[SLJ+14] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions, 2014.

[SS15] I. Segey and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *Proceedings of the 32nd International Conference on International Conference on Machine Learning (ICML'15)*, 37:448–456, 2015.

[SSB+20] Savvas Sioutas, Sander Stuijk, Twan Basten, Lou Somers, and Henk Corporaal. Programming tensor cores from an image processing DSL. In *Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems*. ACM, 2020.

[SSEM19] Adam Seewald, Ulrik Pagh Schultz, Emad Ebeid, and Henrik Skov Midtiby. Coarse-grained computation-oriented energy modeling for heterogeneous parallel embedded systems. *International Journal of Parallel Programming*, nov 2019.

[SVI+15] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision, 2015.

[SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

[SZMG+19] Rafael Stahl, Zhuoran Zhao, Daniel Mueller-Gritschneder, Andreas Gerstlauer, and Ulf Schlichtmann. Fully distributed deep learning inference on resource-constrained edge devices. In *Lecture Notes in Computer Science*, pages 77–90. Springer International Publishing, 2019.

[TAI+20] K. Tanaka, Y. Arikawa, T. Ito, K. Morita, F. Nemoto. N. Miura, K. Terada, J. Teramoto, and T. Sakamoto. Communication-efficient distributed deep

learning with gpu-fpga heterogeneous computing. *IEEE Symposium on High Performance Interconnects (HOTI)*, pages 43–46, August 2020.

[TCP+19] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le. Mnasnet: Platform-aware neural architecture search for mobile. *Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[TDMP15] Y. Thoma, A. Dassatti, D. Molla, and E. Petraglio. Fpga-gpu communicating through pcie. *Microprocessors and Microsystems*, 39(7):565–575, October 2015.

[TST+19] Y. Tu, S. Sadiq, Y. Tao, M. L. Shyu, and S. C. Chen. A power efficient neural network implementation on heterogeneous FPGA and GPU devices. *IEEE 20th International Conference on Information Reuse and Integration for Data Science (IRI)*, September 2019.

[UFG+17] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers. Finn: A framework for fast, scalable binarized neural network inference. *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 65–74, February 2017.

[VGG+20] K. Vanishree, A. George, S. Gunisetty, S. Subramanian, S. Kashyap, and M. Purnaprajna. CoIn: Accelerated CNN co-inference through data partitioning on heterogeneous devices. *6th International Conference on Advanced Computing and Communication Systems (ICACCS)*, March 2020.

[VMFBCGRV20] Delia Velasco-Montero, Jorge Fernandez-Berni, Ricardo Carmona-Galan, and Angel Rodriguez-Vazquez. PreVIous: A methodology for prediction of visual inference performance on IoT devices. *IEEE Internet of Things Journal*, 7(10):9227–9240, oct 2020.

[WCB+18] N. Wang, J. Choi, D. Brand, C. Y. Chen, and K. Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. *Thirty-first Conference on Neural Information Processing Systems (NIPS)*, pages 7685–7694, December 2018.

[WDCC19] Erwei Wang, James J. Davis, Peter Y. K. Cheung, and George A. Constantinides. Lutnet: Rethinking inference in fpga soft logic, 2019.

[WDZ+19] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. *Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[WH05] Haifeng Wang and Dejin Hu. Comparison of SVM and LS-SVM for regression. In *2005 International Conference on Neural Networks and Brain*. IEEE, 2005.

[WPM18] S. Wang, A. Prakash, and T. Mitra. Software support for heterogeneous computing. *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, July 2018.

[WWP09] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, apr 2009.

[WXH21] Nan Wu, Yuan Xie, and Cong Hao. Ironman: Gnn-assisted design space exploration in high-level synthesis via reinforcement learning. February 2021.

[YCS17] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. *Conference on Computer Vision and Pattern Recognition (CVPR) 2017*, November 2017.

[YYL+20] L. Yang, Z. Yan, M. Li, H. Kwon, L. Lai, T. Krishna, V. Chandra, W. Jiang, and Y. Shi. Co-exploration of neural architectures and heterogeneous asic accelerator designs targeting multiple tasks. *Design Automation Conference 2020*, 2020.

[Zah17] M. Zahran. Heterogeneous computing: Here to stay. *Communications of de ACM*, 60(3):42–45, March 2017.

[ZBG18] 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.

[ZCZ+21] Liekang Zeng, Xu Chen, Zhi Zhou, Lei Yang, and Junshan Zhang. Co-Edge: Cooperative DNN inference with adaptive workload partitioning over heterogeneous edge devices. *IEEE/ACM Transactions on Networking*, 29(2):595–608, apr 2021.

[ZL17] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. *International Conference on Learning Representations (ICLR)*, February 2017.

[ZSB+19] Li Zhou, Mohammad Hossein Samavatian, Anys Bacha, Saikat Majumdar, and Radu Teodorescu. Adaptive parallel execution of deep neural networks on heterogeneous edge devices. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*. ACM, nov 2019.

[ZWTD19] L. Zhou, H. Wen, R. Teodorescu, and David H.C. Du. Distributing deep neural networks with containerized partitions at the edge. In *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*, Renton, WA., July 2019. USENIX Association.

[ZZLS17] X. Zhang, X. Zhou, M. Lin, and J. Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. *Computer Vision and Pattern Recognition (CVPR)*, July 2017.