THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : EEATS - Electronique, Electrotechnique, Automatique, Traitement du Signal
Spécialité : Nano électronique et Nano technologies
Unité de recherche : CEA /LIST - Laboratoire d'intégration de systèmes et de technologies

Méthodologie de modélisation basée sur l'apprentissage machine pour l'estimation rapide de métriques de performances des accélérateurs embarqués de réseaux de neurones convolutionnels

An ML-Driven Modeling Methodology for Fast PPA Estimation of NoC-Based CNN Accelerators

Présentée par :

William FABRE

Direction de thèse :

Gilles SICARD INGENIEUR DOCTEUR HDR, CEA Leti	Directeur de thèse
Hana KRICHENE INGENIEURE DE RECHERCHE, CEA Paris-Saclay	Co-encadrante de thèse
Maria LEPECQ INGENIEURE DE RECHERCHE, CEA Paris-Saclay	Co-encadrante de thèse
Choukataly-Caaliph ANDRIAMISAINA INGENIEUR DE RECHERCHE, CEA Paris-Saclay	Co-encadrant de thèse

Rapporteurs :

Philippe COUSSY
PROFESSEUR DES UNIVERSITES, Université de Bretagne-Sud
Daniel CHILLET
PROFESSEUR DES UNIVERSITES, Université de Rennes

Thèse soutenue publiquement le **26 mars 2026**, devant le jury composé de :

Lorena ANGHEL, PROFESSEURE DES UNIVERSITES, Grenoble INP-UGA	Présidente
Gilles SICARD, DIRECTEUR DE RECHERCHE, CEA centre de Grenoble	Directeur de thèse
Philippe COUSSY, PROFESSEUR DES UNIVERSITES, Université de Bretagne-Sud	Rapporteur
Daniel CHILLET, PROFESSEUR DES UNIVERSITES, Université de Rennes	Rapporteur
Abdoulaye GAMATIÉ, DIRECTEUR DE RECHERCHE, CNRS délégation Occitanie ouest	Examinateur

Invités :

Choukataly-Caaliph ANDRIAMISAINA
INGENIEUR DE RECHERCHE, CEA Paris-Saclay
Hana KRICHENE
INGENIEURE DE RECHERCHE, CEA Paris-Saclay

Acknowledgments

J’ai commencé ma thèse convaincu de vouloir faire de la recherche, mais sans savoir si j’en avais réellement les capacités. Je tiens tout d’abord à exprimer ma gratitude à mon directeur de thèse, Gilles SICARD, qui m’a soutenu et accordé sa confiance durant l’ensemble de ce parcours. Je remercie Philippe COUSSY et Daniel CHILLET d’avoir accepté de rapporter ce manuscrit, Lorena ANGHEL d’avoir présidé le jury, ainsi qu’Abdoulaye GAMATIÉ pour son rôle d’examinateur. Leurs remarques et la qualité de nos échanges lors de la soutenance ont enrichi ce travail. Mes trois premières années passées au LIAE ont été placées sous la supervision de Stéphane CHEVOBBE, Maria LEPECQ et Vincent LORRAIN : merci de m’avoir accueilli et accompagné. Merci également à l’ensemble des doctorants et CDD qui ont fait de ces années une période si enrichissante, entre discussions passionnantes et échanges techniques. J’adresse une pensée toute particulière à Karim HAROUN et Kevin MAMBU, dont le soutien m’a été plus que précieux. Mon parcours aurait pu s’achever à ce moment-là. Grâce à Fabian CHERSI et Rohit PRASAD, j’ai eu l’opportunité de continuer aux côtés de Choukataly-Caaliph ANDRIAMISAINA et Hana KRICHENE. Leur confiance et leur bienveillance m’ont permis de m’épanouir pleinement. Merci à eux. Je tiens à exprimer ma gratitude aux membres de mon CSI, Damien QUERLIOZ et Bertrand GRANADO, ainsi qu’à Jumana BOUSSEY, ancienne directrice de l’ED EEATS, et à Olivier ROSSETTO, co-directeur de l’ED EEATS, pour la confiance qu’ils m’ont accordée et pour leur soutien. Merci à mes amis, qui ont toujours été présents. Je dois beaucoup à ma famille. À mon père, qui m’a transmis le goût de la science dès mon enfance et a éveillé en moi cette curiosité qui m’accompagne encore aujourd’hui. À ma mère, qui m’a toujours soutenu dans toutes mes décisions et m’a encouragé sans relâche tout au long de mes études. À mon oncle, qui m’a accueilli et hébergé à Paris, rendant ainsi ces années d’études possibles : sans sa générosité, ce parcours n’aurait pas été le même. Merci. Je remercie aussi les parents d’Angéline, qui m’ont accueilli avec bienveillance et dont l’aide m’a été précieuse. Enfin un grand merci à toi, Angéline. Dès la rédaction de ma lettre de motivation pour cette thèse, tu m’as soutenu, tu m’as accompagné tout au long de ces cinq années, et tu es encore présente aujourd’hui. Merci à tous.

Publications

Journal articles

- **W. Fabre**, K. Haroun, V. Lorrain, M. Lepecq, and G. Sicard, "From Near-Sensor to In-Sensor: A State-of-the-Art Review of Embedded AI Vision Systems," *Sensors*, 24(16):5446, 2024. doi:10.3390/s24165446.
- M. Lepecq, T. Dalgaty, **W. Fabre**, and S. Chevobbe, "End-to-End Implementation of a Convolutional Neural Network on a 3D-Integrated Image Sensor with Macropixel Array," *Sensors*, 23(4):1909, 2023. doi:10.3390/s23041909.

Conference papers

- R. Bishnoi *et al.*, **W. Fabre**, and F. Chersi, "Efficient Deep Learning Platforms for Next-Generation Embedded Edge-AI Systems," *Proc. DATE*, 2026 (accepted, to appear).

Posters and presentations

- **W. Fabre**, H. Krichene, R. Prasad, and C. Andriamisaina, "Flexible DNN Mapping Model for NoC-Based Dataflow Architectures," poster, *EEAI*, Cagliari, Italy, Oct. 21–23, 2024.
- **W. Fabre**, H. Krichene, R. Prasad and C. Andriamisaina, "ML-Based PPA Estimation for Next-Generation Edge-AI Accelerators," poster, *EEAI*, Naples, Italy, Oct. 20–22, 2025.

Submitted

- Co-author manuscript submitted to *Design Automation Conference (DAC)*.

Résumé

La vision par ordinateur embarquée a atteint des niveaux de précision élevés grâce aux réseaux neuronaux profonds. Toutefois, ces gains s'accompagnent de coûts importants en calcul et en transferts de données, souvent incompatibles avec des contraintes strictes de latence et de consommation énergétique. Le recours au cloud est fréquemment inadapté aux systèmes temps réel en raison de la latence, des contraintes de connectivité et des enjeux de confidentialité, ce qui motive l'inférence sur l'appareil, au plus près du capteur d'image. Dans ce contexte, les accélérateurs dataflow basés sur un réseau sur puce (NoC) constituent une option prometteuse, mais leur efficacité dépend fortement du couple architecture et mapping, ce qui fait de la conception et de l'évaluation de ces systèmes un goulot d'étranglement central.

Cette thèse propose PANDA, un cadre d'estimation des métriques de performance fondé sur l'apprentissage et entraîné à partir de données issues de l'implémentation, destiné aux accélérateurs CNN paramétriques reposant sur un NoC. PANDA combine une représentation compacte du mapping avec un flot automatisé qui génère des données d'apprentissage à partir de simulations RTL et de synthèses FPGA, puis entraîne des modèles capables d'estimer rapidement les métriques de performance d'un grand nombre de candidats matériel et mapping, sans recourir systématiquement à des évaluations bas niveau coûteuses. L'approche permet une estimation rapide et fidèle de la latence et de la surface, et fournit des premiers résultats pour l'estimation de la puissance. PANDA est conçu comme un moteur d'évaluation pour accélérer les études conjointes d'architecture et de mapping, et ouvre la voie à son intégration dans une boucle complète d'exploration de l'espace de conception.

Abstract

Embedded computer vision has reached high accuracy thanks to deep neural networks. However, this progress comes with substantial compute and data movement costs that conflict with tight latency and energy budgets. Cloud offloading is often unsuitable for real time systems due to latency, connectivity constraints, and privacy concerns, motivating on device inference close to the image sensor. In this context, NoC-based dataflow accelerators are a promising option, yet their efficiency depends strongly on both architectural parameters and CNN to hardware mappings, making the design and evaluation of these systems a central bottleneck.

This thesis proposes PANDA, a learning-based PPA estimation framework for parametric NoC-based CNN accelerators. PANDA couples a compact mapping representation with an automated flow that generates implementation grounded training data from RTL simulation and FPGA synthesis, and trains predictors that estimate PPA for architecture and mapping candidates at a fraction of the cost of low level evaluation. The approach enables fast and reliable estimation of latency and area, and provides initial results for power estimation. PANDA is designed as a practical evaluation backend to accelerate architecture and mapping studies and can be integrated into a complete design space exploration loop in future work.

Notation Summary

CNN Layer Parameters

O	Number of output channels (filters)
I	Number of input channels
X, Y	Output feature map spatial dimensions (width, height)
W, H	Input feature map spatial dimensions (width, height)
Kx, Ky	Kernel spatial dimensions
$\text{Stride}_x, \text{Stride}_y$	Convolution stride values
$\text{Pad}_x, \text{Pad}_y$	Padding values
$B(o)$	Bias for output channel o

Tensor Notation

$\text{IFMAP}(i, w, h)$	Input feature map tensor
$\text{OFMAP}(o, x, y)$	Output feature map tensor
$\text{KERNEL}(o, i, kx, ky)$	Convolution kernel (weights) tensor

Architecture Parameters

R	Number of PE rows
GRID	Total PE count ($\text{FILTER_WIDTH} \times \text{OUTPUT_WIDTH}$)
noc_size_x	NoC X dimension
noc_size_y	NoC Y dimension
data_width	Datapath bit-width
addr_width	Address width (buffer depth $= 2^{\text{addr_width}}$)
fifo_depth	Router FIFO depth

Tiling and Mapping

t_a	Tile size along axis a
cap_K	Buffer capacity for kernel weights
cap_I	Buffer capacity for input feature maps
cap_O	Buffer capacity for output feature maps
$X_{\mathrm{win}}(x_{\mathrm{tile}})$	Input window width for output tile x_{tile}
$Y_{\mathrm{win}}(y_{\mathrm{tile}})$	Input window height for output tile y_{tile}

Latency Model

T	Layer type (Conv, DW, PW)
S	Execution state or architectural module
c_s	State visitation count for state s
μ_s	Mean cycles per visit for state s
ℓ_s	Total latency for state s ($\ell_s = c_s \times \mu_s$)
ℓ_{layer}	Total layer latency ($\sum_{s \in S} c_s \times \mu_s$)

Evaluation Metrics

y_i	Ground-truth value for sample i
$\hat{y}_i$	Predicted value for sample i
$\bar{y}$	Sample mean of ground truth
$\bar{\hat{y}}$	Sample mean of predictions
n	Number of samples
n_+	Number of samples with strictly positive ground truth
s_{ref}	Reference scale for normalization
R_i	Rank of ground-truth value y_i
$\hat{R}_i$	Rank of predicted value $\hat{y}_i$
$\bar{R}, \bar{\hat{R}}$	Sample means of ranks
R^2	Coefficient of determination
r	Pearson correlation coefficient
ρ	Spearman rank correlation coefficient

Acronyms

ADC Analog-to-Digital Converter

ADP Adaptive Dataflow Pattern

AI Artificial Intelligence

AINoC AI Network-on-Chip

ALU Arithmetic Logic Unit

AMBA Advanced Microcontroller Bus Architecture

API Application Programming Interface

ASIC Application-Specific Integrated Circuit

AXI Advanced eXtensible Interface

BRAM Block RAM

CEA Commissariat à l'Énergie Atomique

CMOS Complementary Metal-Oxide-Semiconductor

CNN Convolutional Neural Network

COE Coefficient File Format

COTS Commercial Off-The-Shelf

CPU Central Processing Unit

CSC CIS Smart Controller

CSV Comma-Separated Values

CUDA Compute Unified Device Architecture

DCP Design Checkpoint

DNN Deep Neural Network

DRAM Dynamic Random-Access Memory

DSE Design Space Exploration

DSP Digital Signal Processing

DW Depthwise Convolution

FF Flip-Flop

FIFO First-In First-Out

FLOPs Floating-Point Operations

FP32 32-bit Floating Point

FPGA Field-Programmable Gate Array

FPS Frames Per Second

FSM Finite State Machine

GC Global Controller

GOP Giga Operations

GPU Graphics Processing Unit

HDR High Dynamic Range

HOG Histogram of Oriented Gradients

HPDP Horizontal Partial Dot Product

I2C Inter-Integrated Circuit

IC Integrated Circuit

IGB Input Global Buffer

IGBH Input Global Buffer Horizontal

IGBV Input Global Buffer Vertical

IMC In-Memory Computing

IoT Internet of Things

ISP Image Signal Processor

JSON JavaScript Object Notation

LUT Look-Up Table

MAC Multiply-Accumulate

MAE Mean Absolute Error

MAPE Mean Absolute Percentage Error

MIPI Mobile Industry Processor Interface

MLP Multi-Layer Perceptron

MPX Macropixel

NCX NE Control Executor

NE Neural Engine

NMAE Normalized Mean Absolute Error

NoC Network-on-Chip

NPU Neural Processing Unit

OGB Output Global Buffer

ONP Output Neuron Processing

PANDA PPA Estimation for Neural Dataflow Architectures

PCIe Peripheral Component Interconnect Express

PE Processing Element

PPA Performance, Power, Area

PSUM Partial Sum

PW Pointwise Convolution

QQVGA Quarter Quarter VGA

QVGA Quarter VGA

RF Register File

RGB Red Green Blue

RISC Reduced Instruction Set Computer

RMSE Root Mean Square Error

ROI Region of Interest

RTL Register-Transfer Level

SAIF Switching Activity Interchange Format

SDK Software Development Kit

SE Squeeze-and-Excitation

SIFT Scale-Invariant Feature Transform

SIMD Single Instruction Multiple Data

SoC System-on-Chip

SPI Serial Peripheral Interface

SRAM Static Random-Access Memory

SURF Speeded-Up Robust Features

TCL Tool Command Language

TOPS Tera Operations Per Second

TPU Tensor Processing Unit

TSV Through-Silicon Via

URAM UltraRAM

VCD Value Change Dump

VGA Video Graphics Array

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit

ViT Vision Transformer

List of Figures

2.1 Functional view of an integrated vision system with AI processing such as [21–23]. . . 25
2.2 Design space for AI vision systems, highlighting the trade-off between flexibility and power consumption across processing configurations [21, 24–45]. 26
2.3 A basic CNN architecture. (a) Shows the network's structure from input to output, visualizing data processing for predictions. (b) Details the mathematical operations, including weights, biases, and activation functions, that map inputs into outputs. 28
2.4 Representation of CNN models for embedded systems, showcasing memory requirements, computational demands, and accuracy on ImageNet-1K. 29
2.5 Representation of uniform and non uniform quantization (inspired by [76]). 33
2.6 Representation of structured and unstructured pruning. 34
2.7 Simplified architectural view of near-sensor AI vision systems. 36
2.8 Simplified architectural view of in-sensor AI vision systems. 37

3.1 DSE workflow overview. 40
3.2 Dataflow illustration of Row Stationary dataflow on a 2D PE array. 44
3.3 Six-dimensional nested loop formulation for convolution operations. 46

4.1 PANDA Framework Workflow for CNN Mapping to a NoC-based dataflow Architecture and PPA Estimation 59
4.2 Overview of the NoC Dataflow-based Architecture for CNN processing. 61
4.3 Finite State Machines of the two-level distributed control architecture: Global Controller FSM (left), PE Controller FSM (right) 63
4.4 Processing Element internal architecture. 64
4.5 Internal architecture of the AINoC router showing the five communication ports. 66
4.6 Distributed memory organization showing Global Buffers interfacing with the PE grid example for standard convolution with $stride = 2$. 67

4.7 ADP for depthwise convolution hidden layer: (a) Diagonal unicast for ifmap mapping from top-right to bottom left, (b) Horizontal multicast for filter distribution. These patterns scale automatically with synthesized grid dimensions. 69
4.8 Hierarchical loop structure with three levels: DRAM loops for temporal distribution, parallelism loops for spatial distribution, and buffer loops for on-chip reuse. 71
4.9 Unified recursive engine for dataflows and reuse (IS, OS, RS, ...) 72
4.10 Hardware architecture illustrating hierarchical dataflow from DRAM through global buffers to the processing element array for OS example. 73
4.11 Mapping model pipeline from simulation to event counters, diagnostics, and validation. 77
4.12 Overview of the PPA estimation methodology combining simulation, synthesis, and machine learning 82
4.13 CNN Dataset Generation Workflow 85
4.14 Supervised learning pipeline for PPA estimation. Feature selection operates within cross-validation folds to prevent data leakage. The best model is retrained on the full training set before final evaluation on a held-out test set. 89
4.15 Multi Stimuli RTL Generator workflow: transforms CNN layer descriptions and dataflow models into VHDL testbenches and parameterized memory initialization files 92

5.1 Target architecture instance: 3×64 PE grid with distributed memory organization (left) and execution visualization example (right). Color variation encodes either memory access count or MAC operation density. 101
5.2 Conv0 RTL simulation trace: memory transactions per bank (OGB_R/OGB_W top, IGB_V_R left, IGB_H_R bottom) and MAC operations per processing element (center matrix). Color variation indicates activity-level. 106
5.3 Execution time breakdown by state for all MobileNet v1 layers (logarithmic scale). . . . 109
5.4 Parameterized loop structure output for a 5×32 configuration: the mapping model supports flexible hardware configurations beyond the fixed 3×64 reference array . . . 110
5.5 Conv0 mapping model output: memory access pattern and PE utilization. Legend shows per element values: OGB_R (boolean), OGB_W (writes/column), IGB_V_R (multicast volume), IGB_H_R (elements/bank), PE (MACs/PE). 111
5.6 Distribution of $GRID = filter_width \times output_width$ (log scaled x axis). The dataset spans more than one order of magnitude in PE array size, ensuring coverage of diverse architectural configurations. 114

5.7 Median resource utilization per module and resource type ($\log_{10}(\text{median} + 1)$ scale). Logic resources concentrate in pe and ainoc_parallel, while memory primitives localize in buffer modules. 115
5.8 Pearson correlation between input features (rows) and resource targets (columns). GRID shows the strongest correlation with logic targets ($r \approx 0.27$), while STRIDE and PADDING have negligible impact. 117
5.9 Test R^2 for each module target pair. Text shows R^2 value and selected algorithm (ET: ExtraTrees, RF: RandomForest, LIN: Linear, CST: Constant). 118
5.10 Test NMAE (%) for each module target pair and selected algorithm (ET: ExtraTrees, RF: RandomForest, LIN: Linear, CST: Constant). 118
5.11 Execution profile for MN_Conv0: transition counts (left) and average latency (right). S denotes the controller state, T the layer type. 121
5.12 Empirical target distributions for PW_COMPUTE: (a) Count is strongly right skewed (mean$= 20{,}452$, median$= 2{,}880$), with outliers spanning roughly three orders of magnitude; (b) Mean Cycles is tightly concentrated around 12-13 cycles, with notably limited dispersion. 122
5.13 Pearson correlation matrix for PW_COMPUTE. Notable values: Total Macs shows moderate correlation with Mean Cycles ($r = 0.42$), while Count correlations remain weak ($r \leq 0.39$). 123
5.14 Feature space projections colored by target values for PW_COMPUTE. 123
5.15 Layer wise total cycles comparison: ground truth (solid) vs. predictions (hatched). Log scale reveals strong agreement across five orders of magnitude. Spearman rank correlation: $\rho = 0.986$ ($p < 0.01$). 125

List of Tables

3.1 Primary metrics for DNN accelerator evaluation and their common units. 42
3.2 Comparison of evaluation backends used in DNN accelerator DSE. 48
3.3 Common integration patterns, defined by the evaluator queried for each candidate during exploration. 52

4.1 Main RTL parameters exposed by the architecture. Ranges are indicative and should be refined from RTL constraints. 62
4.2 Evaluation metrics definitions. 90

5.1 $\mathrm{MobileNetV1}$ ($\alpha = 0.25,$ input 128×128) backbone layers executed in the experiments. 102
5.2 Design space summary per layer type for the 3×64 reference architecture. 103
5.3 Representative simulation costs of MobileNet v1 on the CEA architecture using QuestaSim. 105
5.4 RTL validation summary across representative layer types. All observed values match manual calculations within measurement precision ($< 0.1\%$ error from timing artifacts). 108
5.5 Mapping model validation: exact MAC correspondence and near-exact memory correspondence across representative layers 113
5.6 Time comparison between FPGA synthesis and ML-based area prediction. 119
5.7 Prediction errors on held-out MobileNet v1 layers (representative subset). For each layer, MAE and MAPE are averaged over all FSM states (COMPUTE, LOAD_filter, LOAD_ifmaps, LOAD_config, IDLE). 125
5.8 Execution time comparison across evaluation methods. Prediction achieves two to three orders of magnitude speedup over RTL simulation while maintaining single-digit MAPE on total cycle estimates. 126
5.9 Power estimation results across layers of varying complexity. Dynamic power shows less than 1% variation despite order-of magnitude differences in computational workload. 128

List of Algorithms

1 Multi stimuli RTL-level Simulation and Trace Collection 94
2 Multi-Config Logic Synthesis and Resource Characterization 96

Table of Contents

Acknowledgments 1

Publications 2

Résumé 3

Abstract 4

Notation Summary 5

Acronyms 7

List of Figures 12

List of Tables 13

List of Algorithms 14

1 Introduction 20

1.1 Embedded Vision and the Efficiency Gap . . . 20

1.2 The Performance, Power, and Area Estimation Bottleneck . . . 21

1.3 Research Objectives and Thesis Organization . . . 22

2 Background: Embedded AI Vision Systems 23

2.1 Introduction . . . 23

2.2 From classical vision to embedded AI vision systems . . . 24

2.3 Metrics and Characteristics . . . 26

2.3.1 System Pipeline Overview . . . 26

2.3.2 Convolutional Neural Networks . . . 27

2.3.3 Evolution of Embedded Convolutional Neural Networks . . . 28
2.3.3.1 Early Architectures and Complexity Reduction . . . 30
2.3.3.2 The SqueezeNet Family . . . 30
2.3.3.3 The MobileNet Family . . . 31
2.3.3.4 The EfficientNet Family . . . 32
2.3.3.5 Recent Advances: MobileOne . . . 32
2.3.3.6 Summary . . . 32
2.3.4 Complexity Reduction Techniques . . . 33
2.3.5 AI Processing Unit . . . 34
2.3.6 AI Vision System Metrics . . . 35
2.4 Near sensor AI Vision Systems . . . 35
2.5 In sensor AI Vision Systems . . . 36
2.6 Conclusion . . . 37

3 Related Work: DNN Hardware Modeling and Estimation Frameworks 39
3.1 Design Space Exploration : a Joint Hardware–Mapping Optimization Problem . . . 39
3.1.1 Combinatorial Complexity and Feasibility Constraints . . . 40
3.2 PPA Metrics and Associated Cost Components . . . 41
3.2.1 Evaluation Objectives and Metrics . . . 41
3.2.2 Cost Decomposition: From Workload Events to Technology Costs . . . 43
3.3 Architectural Modeling and Mapping Representations . . . 44
3.3.1 Hardware Modeling Strategies . . . 44
3.3.2 Mapping Formulations . . . 45
3.3.3 Scope and Generality Trade-offs . . . 47
3.4 Evaluation Backends and Modeling Frameworks . . . 48
3.4.1 Component-Level Building Blocks . . . 48
3.4.2 Analytical Evaluation and Cycle-Count Modeling . . . 49
3.4.3 Cycle Accurate and System-level Simulation . . . 50
3.4.4 Synthesis-based Estimation and Implementation Ground Truth . . . 50
3.4.5 Learned Surrogates and Hybrid Estimators . . . 51
3.4.6 From Backends to Full Frameworks . . . 51
3.5 Exploration Algorithms and Search Strategies . . . 53
3.5.1 Budget and Feasibility Constraints . . . 53
3.5.2 Black-Box Optimization . . . 53
3.5.3 Gradient-based and Differentiable Methods . . . 54

3.5.4 Learned Predictors and Surrogate Assisted Search . . . 54
3.5.5 Multi Fidelity Optimization . . . 54
3.5.6 Joint versus Nested Optimization . . . 55
3.5.7 Multi-Objective Considerations . . . 55
3.6 Synthesis and Research Gap . . . 56

4 Methodology: PANDA Framework for Learning-based PPA Estimation 58
4.1 PANDA's Approach and Contributions . . . 58
4.2 NoC-based Dataflow Architecture . . . 60
4.2.1 Parametric Organization and Scalability . . . 61
4.2.2 Distributed Control Methodology . . . 62
4.2.3 Processing Element Architecture . . . 64
4.2.4 AINoC Communication Infrastructure . . . 65
4.2.5 Memory Organization . . . 67
4.2.6 Adaptive Dataflow and Execution Model . . . 68
4.3 Mapping Model of CNN Layers . . . 70
4.3.1 Loop Nest Formulation and Hierarchical Decomposition . . . 70
4.3.2 Tiling Strategy and Memory Constraints . . . 73
4.3.3 Event Counters, Diagnostics, and Validation . . . 77
4.3.3.1 Memory Event Detection . . . 78
4.3.3.2 Validation and Consistency Checks . . . 78
4.3.3.3 Per PE Metrics and Export . . . 79
4.3.3.4 Diagnostics and Outputs . . . 80
4.4 Learning-Based PPA Estimation Methodology . . . 81
4.4.1 Proposed Methodology . . . 82
4.4.1.1 Data Construction Methodology . . . 83
4.4.1.2 Representation and Observability . . . 85
4.4.1.3 Learning Formulation . . . 87
4.4.1.4 Machine Learning Pipeline . . . 89
4.4.2 Implementation . . . 91
4.4.2.1 Stimuli Generation for CNN Inference . . . 92
4.4.2.2 RTL Simulation Infrastructure . . . 93
4.4.2.3 Logic Synthesis Infrastructure . . . 95
4.4.2.4 Area Estimation Implementation . . . 96
4.4.2.5 Latency Estimation Implementation . . . 97

4.4.2.6 Power Estimation Implementation 98

5 Experimental Results and Validation 99

5.1 Experimental Overview 99
5.2 Experimental Setup 100
5.2.1 Compute environment 100
5.2.2 Target Architecture 100
5.2.3 CNN Workloads 102
5.2.4 Toolchain, Stimuli Generation, and Computational Cost 104
5.2.4.1 Stimuli generation and correctness checks 104
5.2.4.2 Computational cost 104
5.3 Ground Truth Data Generation 105
5.3.1 Conv0: Standard Convolution Analysis 106
5.3.1.1 Input Feature Map Distribution 106
5.3.1.2 Filter Distribution 107
5.3.1.3 Output Accumulation and MAC Validation 107
5.3.2 Validation Summary Across Layer Types 108
5.3.3 Global Execution Characterization 109
5.4 Mapping Model Evaluation 110
5.4.1 Model Abstraction and Output Semantics 110
5.4.2 Correspondence Validation 111
5.4.2.1 Input Source Flag 112
5.4.2.2 Input Feature Map Distribution 112
5.4.2.3 Filter Distribution 112
5.4.2.4 Output Accumulation 112
5.4.2.5 Computational Workload 112
5.4.3 Validation Summary 113
5.5 PPA Model Evaluation 113
5.5.1 Area Model Evaluation 114
5.5.1.1 Dataset Characterization and Coverage 114
5.5.1.2 Resource Distribution Analysis 115
5.5.1.3 Feature-Target Correlation Analysis 116
5.5.1.4 Model Performance and Interpretation 117
5.5.1.5 Prediction Speed and Practical Utility 119
5.5.2 Latency Model Evaluation 120

5.5.2.1 Convolution Execution State Profiling . . . 120
5.5.2.2 Pointwise Target Distribution and Implications for Learning . . . 121
5.5.2.3 Pointwise Feature–Target Relationship Analysis . . . 122
5.5.2.4 Overall Model Performance and Generalization . . . 124
5.5.2.5 Prediction Speed and Accuracy Trade-off . . . 126
5.5.3 Power Characterization and Baseline Model . . . 127
5.5.3.1 Dataset and Methodology . . . 127
5.5.3.2 Results and Observed Limitations . . . 127
5.5.3.3 Module-Level Power Distribution . . . 128
5.5.3.4 Pipeline Limitations and Future Directions . . . 128
5.6 Discussion and Scope Boundaries . . . 129
5.6.1 Addressing the Accuracy-Speed Gap . . . 129
5.6.2 Decomposition as Organizing Principle . . . 131
5.6.3 The Mapping-PPA Abstraction Gap . . . 131
5.6.4 Summary . . . 132

6 Conclusion and Perspectives **133**
6.1 Conclusion . . . 133
6.2 Perspectives . . . 134
6.2.1 Power Estimation Refinement . . . 134
6.2.2 Design Space Exploration Integration . . . 135
6.2.3 Generalization and Transferability . . . 135

Bibliography **148**

Chapter 1

Introduction

1.1 Embedded Vision and the Efficiency Gap

Embedded systems increasingly rely on computer vision to perceive and react to their environment, enabling applications ranging from robotics and autonomous navigation to medical and industrial inspection. Over the past decade, deep neural networks (DNNs) and in particular convolutional neural networks (CNNs) have become the dominant approach for recognition, detection, and segmentation, largely due to their accuracy and robustness across diverse visual conditions.

This accuracy comes at a substantial computational and data movement cost. Even compact CNNs require large numbers of multiply-accumulate operations (MAC) per inference and must repeatedly move activations and weights through the memory hierarchy. On resource constrained platforms, these demands compete directly with tight power and latency budgets.

Offloading inference to the cloud is often not a viable alternative. Communication introduces latency and bandwidth constraints, and it raises privacy and reliability concerns. Many embedded vision scenarios require millisecond scale response under limited or intermittent connectivity, which motivates local (on device) inference and pushes compute closer to the image sensor.

Local inference, however, exposes a central tension. Power budgets in embedded vision span from a few milliwatts to a few watts depending on the deployment context, while workloads are heterogeneous even within a single model. Efficient CNN architectures such as MobileNet combine standard convolutions, depthwise convolutions, and pointwise convolutions, each with different arithmetic intensity and dataflow behavior. Hardware optimized for one execution mode may underutilize resources in another: compute bound designs can become memory bound on depthwise layers, while memory bound designs may waste compute on pointwise layers. Achieving good end-to-end efficiency therefore requires architectures and mappings that perform well across heterogeneous

layer types under strict performance, power, and area (PPA) constraints.

This context has driven the development of specialized CNN accelerators. Dataflow architectures organized as regular arrays of processing elements connected by on-chip fabrics often implemented as a Network-on-Chip (NoC) are particularly attractive because they expose parallelism and enable structured data reuse. However, their efficiency depends critically on how the CNN is mapped onto the architecture: tiling, loop ordering, spatial assignment, and communication patterns can change performance and energy by large factors for the same hardware template.

Navigating this coupled hardware-mapping design space requires fast and reliable performance, power, and area estimation. Without such estimators, architecture decisions must rely either on slow, cycle accurate simulation and synthesis flows, or on analytical models that may miss implementation dependent effects that become first order in distributed NoC-based designs. This estimation bottleneck is the central problem addressed in this thesis.

1.2 The Performance, Power, and Area Estimation Bottleneck

Accurate PPA estimation for CNN accelerators is difficult for two interrelated reasons.

First, CNN to hardware mappings are combinatorial and highly constrained. A convolutional layer can be expressed as a nested loop computation over spatial dimensions, channels, and kernel coordinates. A mapping must choose tiling factors, loop ordering, and where data reside in the memory hierarchy. These choices determine reuse, bandwidth pressure, and utilization. Prior work has shown that mapping decisions alone can induce order of magnitude differences in energy and latency for identical hardware, which makes systematic mapping representations and efficient feasibility checks essential.

Second, PPA evaluation spans a wide spectrum of cost and fidelity. Analytical models enable rapid evaluation and broad exploration, but they often simplify effects such as routing delays, bank conflicts, protocol overhead, and congestion. At the other extreme, cycle accurate simulation and implementation flows capture these effects but can take minutes to hours per configuration, which sharply limits the number of candidates that can be evaluated. This accuracy cost trade off is particularly acute on FPGAs, where timing closure, routing congestion, and resource fragmentation can dominate realized performance and area.

These constraints motivate estimation methodologies that combine the throughput of fast evaluators with sensitivity to implementation-level effects. In particular, an effective approach should preserve candidate ranking across the design space (to guide search), while keeping marginal evaluation cost low enough to support exploration over thousands of candidates.

1.3 Research Objectives and Thesis Organization

This thesis develops a learning-based PPA estimation framework for NoC-based CNN accelerators with the objective of enabling rapid evaluation of hardware mapping configurations. While power estimation is investigated, the validated contributions primarily concern latency and area prediction. The estimation models are designed to serve as evaluation backends for design space exploration algorithms.

Chapter 2 surveys embedded AI vision systems and positions near sensor and in sensor computing approaches, clarifying the constraints that motivate efficient on device inference. Chapter 3 reviews existing modeling and estimation frameworks for CNN accelerators and synthesizes how mappings, evaluators, and search strategies interact in design space exploration. Chapter 4 introduces PANDA (Performance, Power and Area estimation framework for Neural Dataflow Accelerators), including the hardware model of the target NoC-based dataflow architecture, the mapping abstraction, the automated RTL/synthesis data generation pipeline, and the learning methodology. Chapter 5 validates PANDA on MobileNet v1 targeting a Xilinx Virtex UltraScale+ FPGA, reporting accuracy for latency and area prediction and discussing current limitations for power. Chapter 6 concludes and outlines future directions, including power model refinement and integration into end-to-end exploration workflows.

The main contributions of this thesis are:

- A mapping model that derives memory access patterns of CNN workloads on the target architecture.
- An ML-driven DNN-HW modeling methodology for fast and scalable PPA estimation.
- A framework enabling efficient dataset generation for ML-driven DNN-HW modeling

Chapter 2

Background: Embedded AI Vision Systems

2.1 Introduction

In the age of artificial intelligence, computer vision leverages edge computing [1] to analyze complex visual data in real-time, transforming machine environment interactions with unprecedented precision [2].

Vision systems capture images and transfer them to processing units, where the raw visual data are transformed into usable information. This process is used in applications such as autonomous navigation, augmented reality, mobile phones, and drones, where the rapid analysis of visual data enables seamless interactions and real-time decisions.

However, deploying modern vision algorithms on embedded platforms remains challenging due to constraints on latency, energy, bandwidth, and privacy. This chapter, based on our previously published survey [3], examines architectural solutions that bring computation closer to the image sensor, and compares near-sensor and in sensor paradigms.

This chapter is organized as follows. Section 2.2 provides a brief background, from classical computer vision to embedded AI vision systems. Section 2.3 introduces the fundamental metrics and characteristics of AI vision systems. Sections 2.4 and 2.5 review near-sensor and in-sensor approaches, respectively. Section 2.6 summarizes the key findings and positions the contributions of this thesis within the landscape.

2.2 From classical vision to embedded AI vision systems

Before the advent of AI-based techniques, image processing relied on custom algorithms that required considerable manual tuning and expertise. Methods such as SIFT (Scale Invariant Feature Transform) [4] and SURF (Speeded-Up Robust Features) [5] were used for feature detection and object recognition but required precise parameter adjustments and struggled to generalize to diverse datasets. HOG (Histogram of Oriented Gradients) [6] was widely used for pedestrian detection due to its ability to capture gradients but was sensitive to variations in lighting and perspective. These traditional algorithms were time consuming to develop and often failed to adapt to new environments, making them labor intensive and limiting their effectiveness in dynamic scenarios.

The limitations of traditional image processing methods, which heavily rely on manual feature engineering and lack of adaptability, have paved the way for more sophisticated approaches. Approaches based on deep neural networks, particularly convolutional neural networks, have emerged as groundbreaking solutions [7–9]. This shift is largely due to advances in GPU technology, data storage, and the availability of large datasets [10, 11]. Unlike customized algorithms, CNNs automatically learn hierarchical features from raw data. This ability allows them to excel at complex tasks such as object detection, recognition, and segmentation [12–14]. This capability allows CNNs to generalize to diverse datasets and adapt to dynamic environments, thus overcoming the limitations of previous methods.

CNNs are generally structured into two main components: the backbone and the head. The backbone is responsible for extracting spatial features from images using convolutional layers, while the head often uses a multilayer perceptron (MLP) for tasks such as image classification [15]. This architecture enables CNNs to efficiently process and analyze image data, making them an essential component of modern computer vision applications.

While recent advances in Pure-MLP networks [16, 17] and attention-based models, such as Transformers [18, 19] demonstrate significant potential in processing complex vision tasks, they are often accompanied by substantial computational requirements. For example, the Vision Transformer (ViT-L/16) requires approximately 190 billion FLOPs and 307 million parameters [18], illustrating the high resource requirements. Although new models of vision processing have emerged, CNNs maintain a prominent position in artificial intelligence solutions due to their effectiveness and extensive refinement over the years.

However, deploying CNNs introduces significant challenges in the context of embedded systems:

- Image sensors generate substantial amounts of data, which can lead to processing and communication bottlenecks.

- Real-time processing requirements often exceed the latency capabilities of cloud computing.
- Computation and communication require high energy consumption [20].
- Cloud-based processing raises privacy concerns when handling sensitive visual data.

To address these challenges, efforts are being made to integrate complex processing directly into embedded vision systems, enabling autonomous image capture as well as analysis. This approach reduces the need for communication with the cloud and the use of remote cloud computing resources, optimizing overall energy consumption and processing latency.

By integrating advanced processing capabilities into these systems, AI embedded vision systems can achieve greater autonomy and efficiency. Such a system typically comprises a camera for initial data capture and rudimentary image processing, combined with a processing unit that refines image quality and performs advanced analysis, ultimately transmitting the analyzed visual information (as illustrated in Figure 2.1).

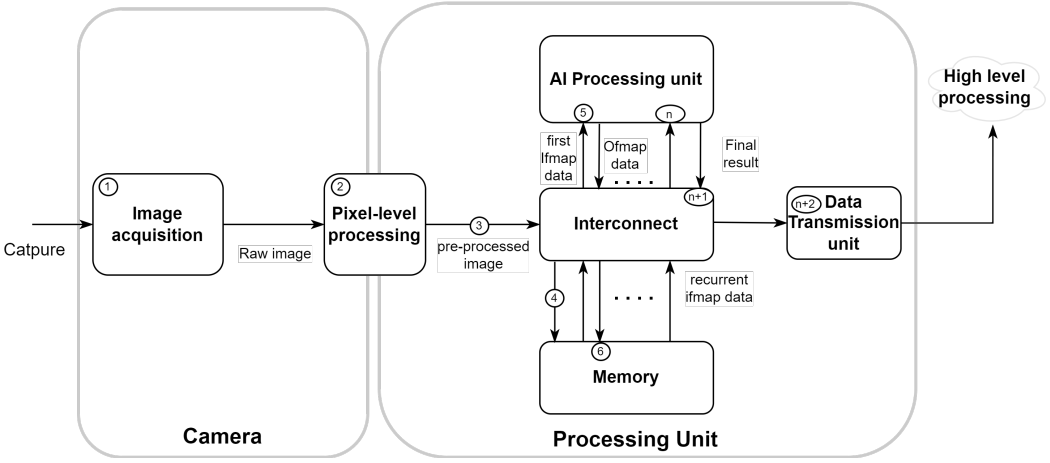


Figure 2.1: Functional view of an integrated vision system with AI processing such as [21–23].

The main challenge for embedded AI vision systems is balancing flexibility against power consumption and silicon area, as illustrated in Figure 2.2. These systems span a wide spectrum: from cloud-based solutions offering high flexibility at significant energy cost, to ultra low power devices that achieve energy efficiency within tight area budgets but with limited adaptability.

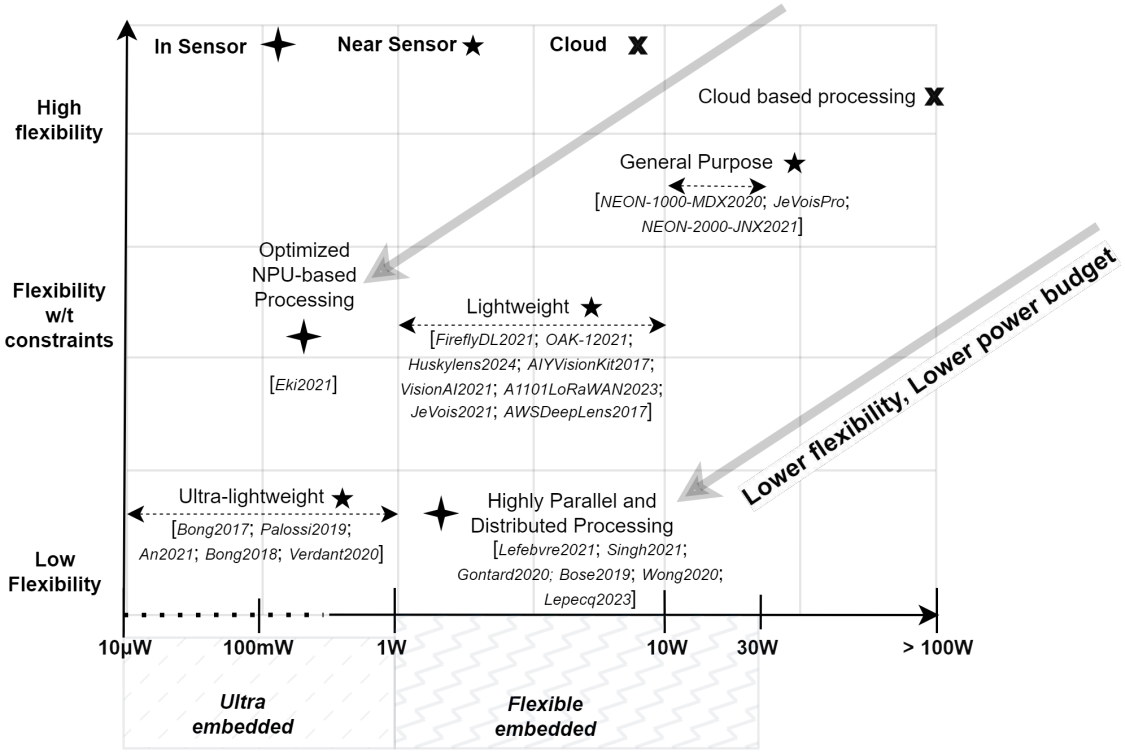


Figure 2.2: Design space for AI vision systems, highlighting the trade-off between flexibility and power consumption across processing configurations [21, 24–45].

A key architectural distinction concerns the physical placement of computation relative to the image sensor: processing may occur near the sensor (near-sensor) or directly within the sensor itself (in-sensor), each approach presenting distinct tradeoffs in performance, power, and design complexity.

2.3 Metrics and Characteristics

This section delves into the metrics and characteristics of AI integrated computer vision systems, offering a comprehensive analysis from image capture to CNN processing.

2.3.1 System Pipeline Overview

An embedded AI vision system pipeline typically comprises several stages (Figure 2.1). Image acquisition performs sensor readout and produces a digital pixel stream, which is then processed by an Image Signal Processor (ISP) through low level pixel domain operations such as normalization,

resizing, and region of interest (ROI) extraction. The resulting feature stream is transferred through the memory hierarchy to a CNN accelerator for inference.

End to end data movement depends both on the network structure and on the compute organization. In folded accelerators, computation is time multiplexed over a limited set of processing elements (PEs), which often implies repeated transfers of weights and intermediate activations through the memory system across layers, depending on buffering capacity and reuse opportunities. In contrast, spatially parallel (dataflow) accelerators exploit PE arrays by mapping loop dimensions spatially to increase on-chip reuse and reduce external traffic. Some architectures additionally organize computation as a multi stage pipeline, streaming data across stages to further limit intermediate feature map storage and transfers [46].

Each stage transition in this processing pipeline requires data transfer, which frequently constitutes the dominant performance bottleneck. The sensor readout and the sensor to ISP interface bandwidth constrain the maximum attainable frame acquisition rate. Concurrently, the digital interconnect bandwidth among the ISP, the memory subsystem, and the CNN accelerator limits the achievable inference throughput. Consequently, efficient orchestration of both analog and digital bandwidth resources is essential for reducing end-to-end latency.

These bandwidth limitations impose stringent constraints on memory subsystem design and capacity. During inference, the system must not only store the CNN parameters (weights and bias) but also buffer intermediate data, including partial sums and feature maps. Inadequate scheduling or dataflow orchestration can cause intermediate activations to be produced more rapidly than they are consumed, leading to excessive buffering demands. In such scenarios, available memory capacity becomes a hard constraint, potentially limiting system performance and scalability.

2.3.2 Convolutional Neural Networks

A CNN (illustrated in Figure 2.3a) is a powerful type of neural network for analyzing image data. Its execution is known as the inference process. During the inference process, it uses filters called kernels, which perform multiplication and accumulation operations, that are fundamental to the extraction of features by the convolution process. Each filter is applied across different channels of the input, such as the color channels in an RGB image, to extract distinct features. The CNN structure includes convolutional layers with filtering kernels and activation functions that introduce non linearity, allowing the network to learn complex visual patterns from multichannel inputs.

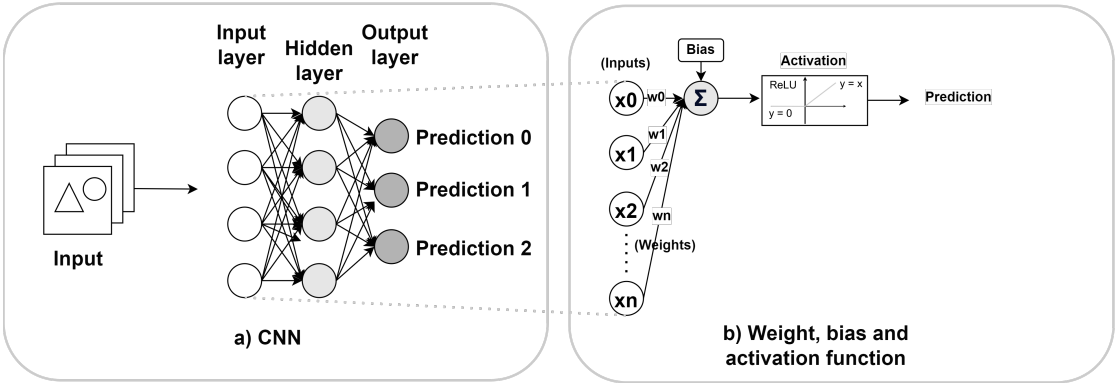


Figure 2.3: A basic CNN architecture. (a) Shows the network's structure from input to output, visualizing data processing for predictions. (b) Details the mathematical operations, including weights, biases, and activation functions, that map inputs into outputs.

In a CNN, parameters primarily refer to the weights and biases associated with each kernel (illustrated in Figure 2.3b). During training, conducted through gradient descent [47], the network iteratively adjusts parameters to enhance model performance based on the provided training data. The quantity of these parameters can significantly influence both the performance and computational intensity of the model. A higher number of parameters often means greater representational power, which can lead to more precise feature extraction and accuracy. However, it also increases the model's computational requirements and memory use, which can be critical when deployed at the edge.

2.3.3 Evolution of Embedded Convolutional Neural Networks

Over time, the integration of CNNs into embedded devices has been driven by the need to balance model complexity and performance, leading to numerous innovations in CNN model design. We focus specifically on how CNN topology influences performance: this includes the organization and interaction of layers within the backbone and head, as well as the operations performed within these layers. Accuracy [48] serves as the primary metric for comparison.

Figure 2.4 presents a representation of CNN models for embedded systems, showcasing memory requirements, computational demands, and accuracy for classification tasks on ImageNet-1K [10]. Each bubble represents a model, with size indicating memory requirements and position reflecting computational requirements and model accuracy, following the visualization approach of [48] and using representative models from the literature [15, 49–58].

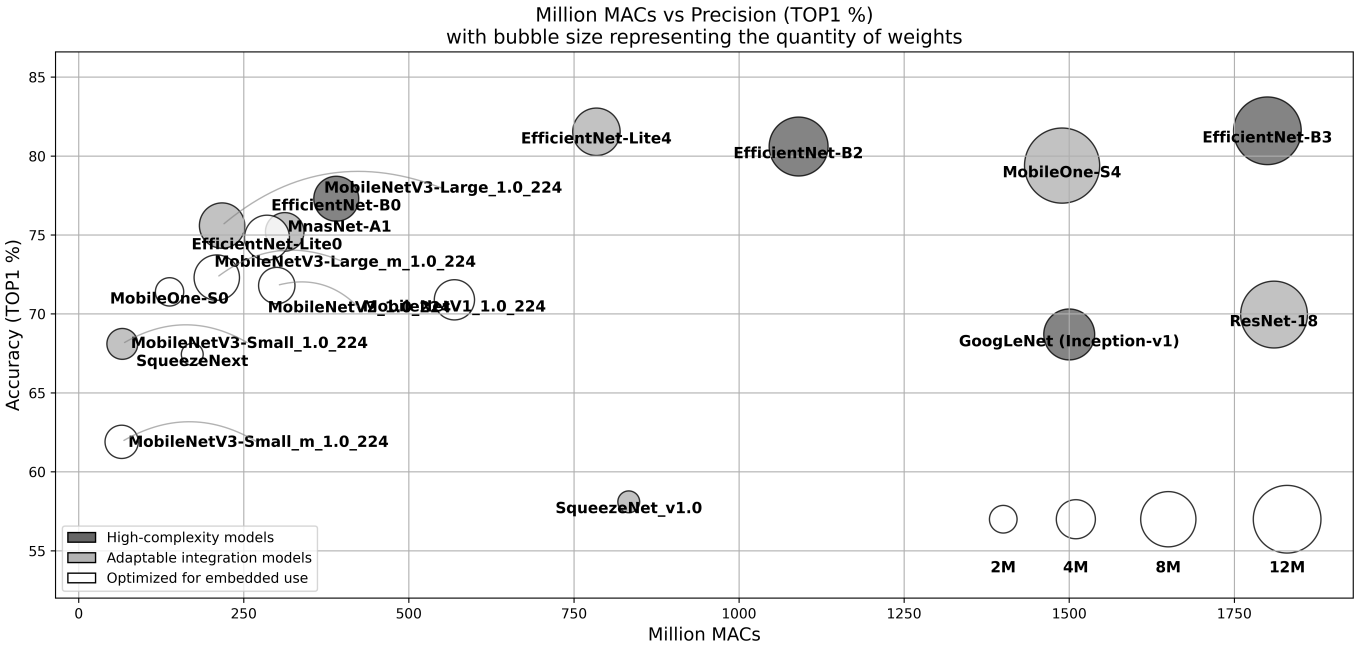


Figure 2.4: Representation of CNN models for embedded systems, showcasing memory requirements, computational demands, and accuracy on ImageNet-1K.

We classify CNN models optimized for embedded vision systems into three distinct groups, each representing a different balance between ease of implementation, computational requirements, number of parameters, and accuracy:

- High Complexity Models (dark gray): These models feature advanced topology and excellent accuracy but require significant layer modifications to adapt to embedded vision systems. These adaptations may include modifying or removing specific components such as Squeeze and Excite modules [52] or Swish activation [59].

- Adaptable Integration Models (light gray): This category includes designs that achieve high accuracy and maintain a manageable computing footprint. However, these models often require adaptations due to the large number of parameters and the computing power they require.

- Optimized for Embedded Use (white): These models are designed specifically for embedded systems, with an emphasis on ease of integration and minimal computational requirements. They use simple computation methods, such as ReLU activation, convolution factorization, and depth separable convolutions [60].

2.3.3.1 Early Architectures and Complexity Reduction

The introduction of VGGNet (2014) [61] is the first step towards complexity reductions. VGGNet simplified the topology of AlexNet (2012) [12] by using convolution factorization, allowing it to exclusively use 3 $\times$ 3 convolution filters. With approximately 15.5 billion MACs and 138 million parameters, VGGNet falls under the category of High Complexity Models.

GoogLeNet (2014) [49] marked a turning point in the optimization of embeddability by using inception modules paralleling multiple convolution sizes. By combining convolution factorization and 1 $\times$ 1 convolutions to reduce channel dimensionality, GoogLeNet achieved slightly better accuracy than VGGNet, but with approximately 1.5 billion MACs and 6.8 million parameters.

ResNet [15] introduced residual connections that facilitate the training of deeper models and improve accuracy. In practice, ResNet variants cover a wide range of compute and memory footprints, enabling different embedded trade-offs. Residual connections are shortcuts that skip one or more layers, improving deep network learning by facilitating gradient backpropagation. ResNet-50, with 1.7 billion MACs and 25.6 million parameters, outperforms VGG16 with a top-5 accuracy of 92.8% on ImageNet. Its reduced computational requirements classify ResNet as an Adaptable Integration Model. Additionally, there are numerous ResNet variants like ResNet-18 and ResNet-8 that offer easier implementation with different trade-offs in terms of computational demands and accuracy.

2.3.3.2 The SqueezeNet Family

SqueezeNet (2016) [51] is a compact architecture built around Fire modules (a squeeze 1×1 stage followed by an expand stage) to reduce parameters and compute. Squeeze and Excitation (SE) blocks were introduced later as a separate mechanism to reweight channels [52]. This module dynamically adjusts the importance of different features in a layer, allowing the most important features to be strengthened and the less important ones to be reduced. SqueezeNet's lightweight architecture offers accuracy comparable to AlexNet, but with only 860 million MACs and 1.25 million parameters, achieving an accuracy of 58.1%. SqueezeNet is categorized as an Adaptable Integration Model due to its low computational requirements and ease of integration.

SqueezeNext (2018) [54] is the evolution of SqueezeNet, designed to be even more efficient with fewer parameters and lower computational cost. SqueezeNext achieves this through further optimization of the fire modules by simplifying their structure and incorporating bottleneck layers. For instance, the SqueezeNext 1.0-SqNxt-23 model has 285 million MACs and 1.30 million parameters, while the SqueezeNext 1.0-SqNxt-23v5 model has 348 million MACs and 1.82 million parameters. SqueezeNext is categorized as Optimized for Embedded Use due to its extremely low computational

requirements.

2.3.3.3 The MobileNet Family

The MobileNet family [53, 57, 58] of networks is targeted for mobile and highly embedded systems, and represents a key benchmark for embedded CNN accelerators.

MobileNetV1 (2017) [58] uses depthwise separable convolutions introduced by [60]. This method separates the convolution into two layers: a depthwise convolution, applying a single filter per input channel, and a pointwise convolution that combines these outputs, effectively reducing computational complexity and model size. It also uses scaling factors to further lighten the model, considering the size of the input image and the number of parameters. With an input size of 224 $\times$ 224, the MobileNetV1_1.0_224 model has 569 million MACs and 4.2 million parameters, while a smaller MobileNetV1_0.5_224 model has 150 million MACs and 1.3 million parameters. Both fit into the Optimized for Embedded Use category.

MobileNetV2 (2018) [57] also uses depthwise separable convolutions. It reduces computational requirements and increases accuracy by using inverted residual connections and linear bottlenecks. The process involves an initial 1 $\times$ 1 convolution layer to increase the representation (expansion), followed by a 3 $\times$ 3 depthwise convolution to extract information, and a final 1 $\times$ 1 convolution layer (projection) to reduce the data volume. With an input size of 224 $\times$ 224, the MobileNetV2_1.0_224 model has 300 million MACs and 3.4 million parameters, while a smaller MobileNetV2_0.5_224 model has 97 million MACs and 1.95 million parameters [62]. Both belong to the Optimized for Embedded Use category.

MobileNetV3 (2019) [53] combined the lessons learned from previous MobileNet versions and MnasNet [55], using Neural Architecture Search techniques [63] to optimize the architecture and SE modules to enhance accuracy. With an input size of 224 $\times$ 224, the MobileNetV3-Large_1.0_224 model has 155 million MACs and 4.0 million parameters, while the MobileNetV3-Small_1.0_224 model has 66 million MACs and 2.9 million parameters [64]. MobileNetV3 models are categorized under Adaptable Integration Models due to their advanced topology and lightweight architecture.

The depthwise (DW) and pointwise (PW) layers of MobileNet architectures have specific requirements [65]: DW layers are more memory intensive and PW layers are more processing intensive. This distinction is important for accelerator design, as it influences dataflow strategies and resource allocation.

2.3.3.4 The EfficientNet Family

EfficientNet (2020) [50] provides a method for coordinately scaling network dimensions (channels, number of kernels, and input resolution) using scaling factors. It also uses depthwise separable convolutions, inverted residual connections, and SE modules. With a range of 390 million MACs and 5.3 million parameters (EfficientNet-B0) up to 24 billion MACs and 66 million parameters (EfficientNet-B7), EfficientNet fits into the High Complexity Models category due to its advanced topology and high number of MACs and parameters for EfficientNet B2 to B7.

Optimized for embedded vision systems, EfficientNet-Lite (2020) [66] and MobileNetV3-minimalistic (2019) [56] simplify EfficientNet [50] and MobileNetV3 [53], respectively. These adaptations reduce multiply-accumulate operations by removing features such as Swish activation [59] and SE modules, which are incompatible with the quantization methods [66] used to simplify complexity. Instead, they use the ReLU6 activation function [67] to maintain efficiency under quantization constraints. EfficientNet-Lite models include EfficientNet-Lite0 with 285 million MACs and 5.3 million parameters, and EfficientNet-Lite4 with 784 million MACs and 5.9 million parameters.

2.3.3.5 Recent Advances: MobileOne

MobileOne (2023) [68] introduced a unique architecture that separates its structure during training and inference, using specialized blocks for improved accuracy and efficiency during training that is simplified through a reparameterization process for better embedded performance during inference. This separation allows MobileOne to maintain high performance during training and efficient deployment during inference, making it highly suitable for embedded applications. The different versions of MobileOne mostly belong to the Optimized for Embedded Use category. For instance, MobileOne-S1 has 285 million MACs and 5.3 million parameters, while MobileOne-S4 has 1.5 billion MACs and 14.8 million parameters.

2.3.3.6 Summary

The integration of CNNs into embedded devices has evolved to find multiple balances between model complexity and accuracy. These innovations are driven by the need to reduce computing and memory requirements while maintaining acceptable accuracy levels. This evolution has resulted in a generation of CNNs optimized for embedded systems, ranging from highly complex models requiring significant modifications for efficient integration, to models specifically designed for embedded use, emphasizing simplicity and lightness.

2.3.4 Complexity Reduction Techniques

Complexity reduction techniques, such as quantization [69–71] or pruning [72, 73], enable the co-design and optimization of CNNs while taking system capabilities into account.

Quantization adjusts data representation dynamics, facilitating an algorithm-hardware co-design suited to constrained architectures (illustrated in Figure 2.5). This approach results in smaller operators and allows for more efficient use of memory as data size is decreased. Quantization can transform FP32 (IEEE 754 single precision floating point format) values into integer representations and can further reduce them to binary representations [74, 75]. The process converts full precision weights and activations to lower bit-width representations, reducing the model size and computational requirements while preserving accuracy.

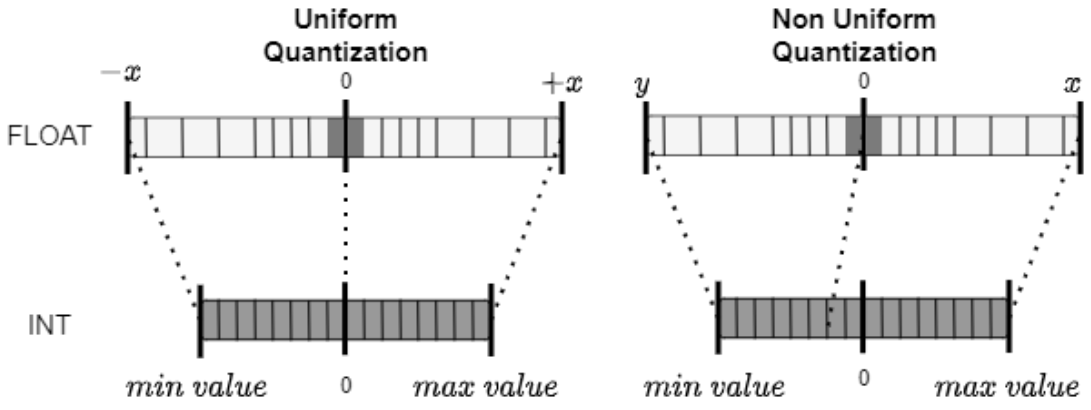


Figure 2.5: Representation of uniform and non uniform quantization (inspired by [76]).

Pruning is a model compression technique that leverages sparsity to reduce computational complexity. It does so by pruning weights, activations, or both (as illustrated in Figure 2.6). Structured pruning refers to the pruning of entire filters or layers, while unstructured pruning removes individual weights or activations. Unstructured pruning removes individual weights across filters, allowing for fine grained sparsity. Structured pruning removes entire channels or filters, resulting in a more regular pruned architecture that can be more efficiently implemented in hardware.

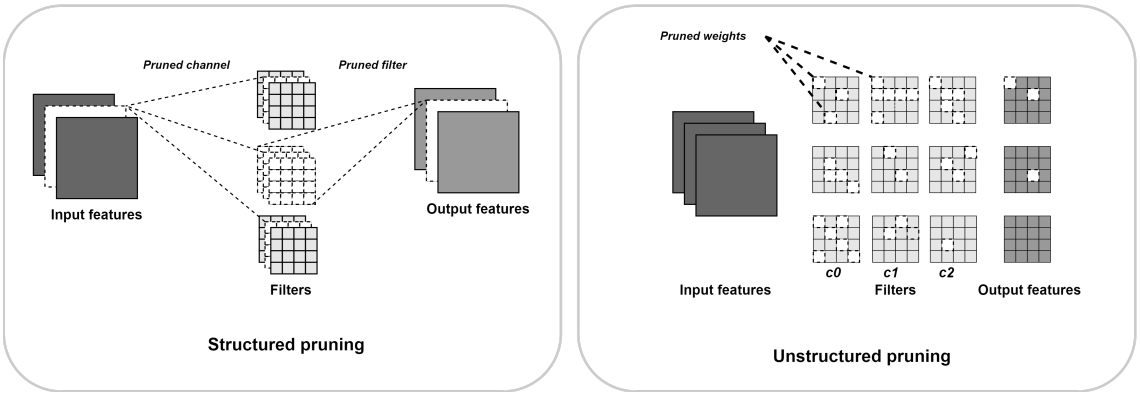


Figure 2.6: Representation of structured and unstructured pruning.

2.3.5 AI Processing Unit

The adaptability, efficiency, and effectiveness of CNNs are influenced by several critical aspects: flexibility, programmability, the software stack used and the types of AI processing units deployed.

The flexibility of a system is measured by its ability to adapt to different computational requirements, memory needs, CNN model types, and CNN layer types. We distinguish three levels: high flexibility approaches that impose few restrictions on CNN choice; moderate flexibility approaches focusing on specific CNN subsets; and low flexibility approaches designed for particular CNN models.

We categorize processing units for CNNs into three levels based on their flexibility (as shown in Figure 2.2):

- General purpose systems: Using architectures such as CPUs, GPUs, and FPGAs, these systems deliver high performance across a wide range of AI processing tasks with great flexibility, at the cost of increased power consumption and larger size.
- Lightweight systems: These systems employ specialized computing architectures like NPUs that focus on energy efficiency for particular tasks, achieving a balance between performance and power consumption.
- Ultra lightweight systems: These systems employ specific computing architectures generally designed for a single task, maximizing power efficiency at the expense of flexibility.

These categories reflect the fundamental tradeoffs governing embedded AI vision systems: performance, power, area, and flexibility cannot be simultaneously optimized. Improving one dimension typically degrades another : high flexibility demands larger, more power hungry architectures, while stringent performance, power and area constraints force specialization toward narrow application domains.

2.3.6 AI Vision System Metrics

The tradeoffs outlined above are quantified through a set of complementary metrics:

- Throughput : measured in frames per second (FPS), quantifies the system's capability to process image data and exhibits a direct relationship with end-to-end latency in real-time applications. Within this context, latency is interpreted as the Performance dimension in the PPA metrics of this thesis.
- Power Consumption (Watts): the energy drawn during operation, determining thermal constraints and battery life.
- Area (mm^2): the silicon footprint of the integrated circuit, constraining both cost and integration in size limited deployments.
- Energy Efficiency (TOPS/W): the ratio of computational throughput to power consumption. Maximizing this metric typically requires optimizing dataflow to increase data reuse [8]. It must be reported with arithmetic precision and operation definition.
- Utilization Rate: the fraction of computational resources actively engaged during inference, reflecting how well the workload maps to the hardware.

No single metric suffices to characterize a system: a high TOPS/W figure is meaningless if throughput is insufficient for the target application, and minimal area is irrelevant if power constraints are violated. Evaluating embedded AI vision systems therefore requires jointly considering these metrics against application specific requirements. The following sections survey existing architectures through this lens, distinguishing near-sensor implementations (Section 2.4) from in-sensor approaches (Section 2.5).

2.4 Near sensor AI Vision Systems

Near sensor architectures physically separate the image sensor from the processing unit, typically interconnected via standard interfaces (MIPI, SPI). This modularity enables rapid system integration using commercial off the shelf components, but incurs data transfer overhead between capture and inference.

Figure 2.7 illustrates representative implementations across the flexibility spectrum defined in Section 2.3. General purpose platforms such as NVIDIA Jetson [77] deliver high throughput (up to 32

TOPS) at significant power cost (10–30 W), supported by mature software stacks (TensorRT, CUDA). Lightweight systems based on dedicated NPUs such as Intel Movidius [78] reduce consumption to 1–5 W by restricting supported network topologies. Ultra lightweight implementations targeting sub watt operation [37, 79] further constrain flexibility, often requiring quantized models and fixed dataflows.

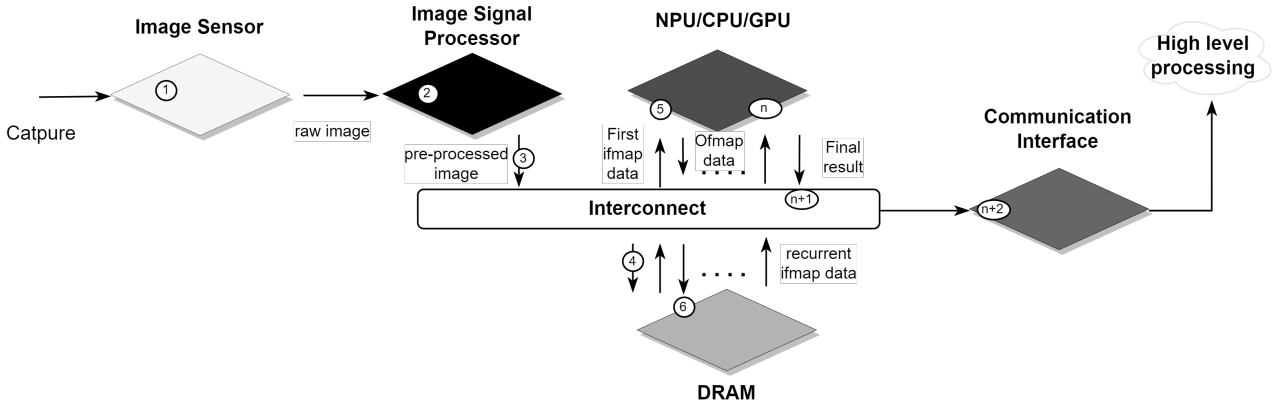


Figure 2.7: Simplified architectural view of near-sensor AI vision systems.

The principal limitation of near-sensor systems lies in the memory bandwidth required to transfer pixel data from sensor to processor: for high resolution streams, this data movement dominates both latency and energy consumption, motivating tighter integration approaches.

2.5 In sensor AI Vision Systems

In sensor architectures address the data movement bottleneck by integrating processing capabilities within or immediately beneath the pixel array. This proximity eliminates the sensor to processor interface, enabling highly parallel access to pixel data with minimal transfer energy.

Two integration strategies have emerged (Figure 2.8). The first leverages 3D stacking technologies [80] to place NPU circuits directly beneath the pixel array on separate silicon tiers. Systems such as [40] demonstrate that this approach achieves near-sensor throughput at ultra lightweight power levels (sub-400 mW), effectively shifting the PPA tradeoff frontier. The second strategy implements computation within the focal plane itself [81, 82], distributing processing elements across the pixel array for maximum parallelism. However, the limited silicon area available per pixel constrains both memory capacity and computational complexity.

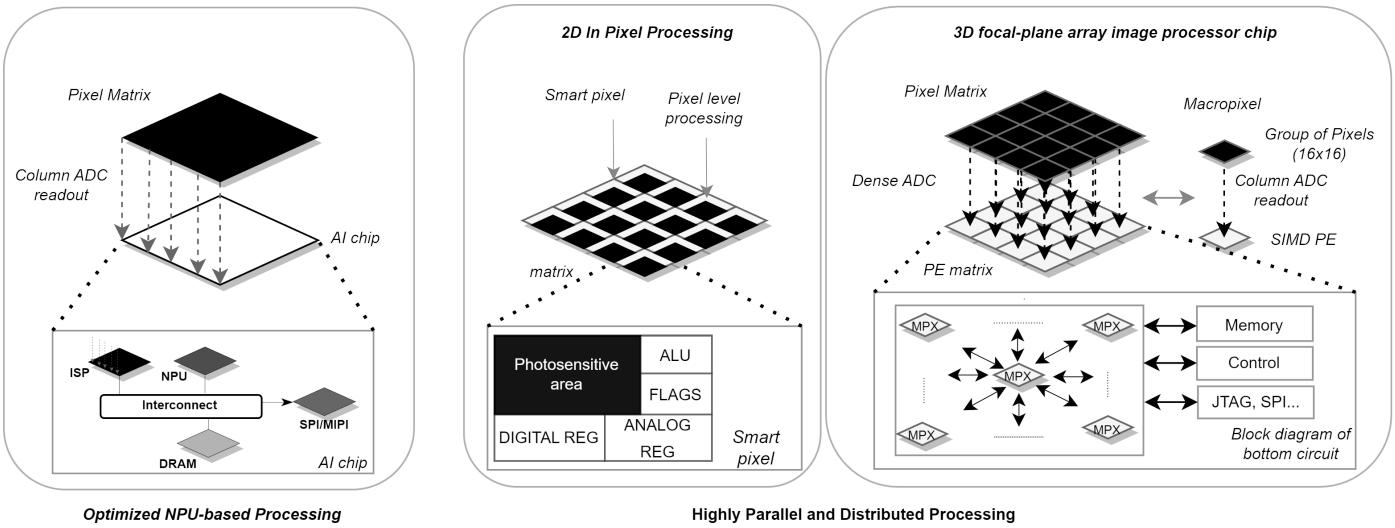


Figure 2.8: Simplified architectural view of in-sensor AI vision systems.

These architectural choices directly impact the design space explored in this thesis: as processing moves closer to the sensor, the accelerator becomes increasingly specialized, and accurate PPA estimation must account for the tight coupling between pixel array characteristics and inference hardware.

2.6 Conclusion

This chapter surveyed embedded AI vision systems, distinguishing near-sensor architectures that leverage commercial processing units from in-sensor approaches exploiting 3D integration to minimize data movement. Across both paradigms, the fundamental tradeoff between performance, power, area, and flexibility governs architectural choices: general purpose platforms offer adaptability at significant energy cost, while ultra lightweight implementations achieve efficiency through specialization.

The evolution of CNN architectures toward embedded optimized variants, from early complexity reductions through depthwise separable convolutions and quantization techniques, has been a key enabler for edge deployment. These innovations also multiply the design space: the combination of network topologies, hardware configurations, and dataflow strategies creates a vast landscape of possible implementations, each exhibiting distinct PPA characteristics.

Systematic exploration of this design space to identify configurations that comply with application specific constraints requires the adoption of structured exploration methodologies. Fundamentally, such methodologies depend on the ability to rapidly and accurately estimate the performance,

power consumption, and silicon area of candidate architectures without resorting to computationally expensive synthesis flows. The following chapter provides a comprehensive review of existing techniques for modeling and estimating CNN accelerators, thereby establishing the methodological and conceptual foundation for the contributions presented in this thesis.

Chapter 3

Related Work: DNN Hardware Modeling and Estimation Frameworks

3.1 Design Space Exploration : a Joint Hardware–Mapping Optimization Problem

The previous chapter motivated embedded intelligence close to the image sensor and reviewed near-sensor in-sensor architectural approaches to reduce power consumption and latency [3]. Translating this motivation into an actual design choice requires selecting a DNN accelerator suited to near-sensor or in-sensor deployment under workload and platform constraints. Because accelerator architectures span a wide spectrum of power, performance, and area (PPA) trade-offs, this selection cannot be reduced to a single criterion and naturally motivates design space exploration (DSE).

For a given workload, we define a design point as the pair $(\text{hardware}, \text{mapping})$. The hardware instantiates compute resources, memory hierarchy, and communication mechanisms. The mapping specifies how the workload is scheduled onto these resources through tiling, loop ordering, and parallelization choices. These decisions are interdependent. Hardware constraints restrict feasible mappings, while the mapping determines utilization and data movement. Consequently, DNN accelerator optimization must be formulated over the joint hardware-mapping space rather than optimizing each in isolation [83–85].

A typical DSE workflow takes three inputs, a hardware model, a workload, and a candidate mapping, and iterates between mapping, evaluation, and search, as shown in Figure 3.1. The mapping model proposes a schedule for a candidate architecture. The resulting hardware-mapping pair is evaluated by a PPA evaluator. The exploration algorithm uses these evaluations, together with

feasibility checks, to select the next candidates to test [83, 84, 86, 87].

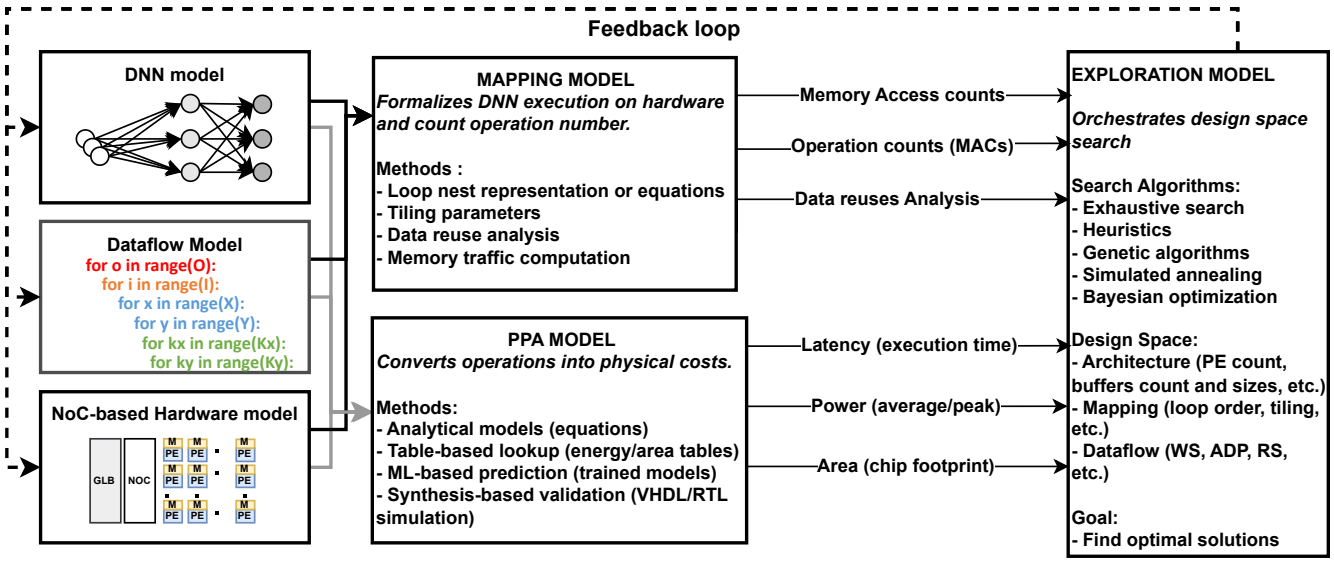


Figure 3.1: DSE workflow overview.

The remainder of this chapter assumes this coupled view. It focuses on what the evaluator must predict and on how hardware and mapping must be represented so that these quantities can be estimated consistently across design points.

3.1.1 Combinatorial Complexity and Feasibility Constraints

The main difficulty of DSE is the combinatorial growth of candidates induced by the product of architectural and mapping choices. Even when the architecture is characterized by a small set of parameters and the mapping is restricted to legal transformations, the number of valid hardware-mapping pairs grows multiplicatively with the available choices [83, 84]. Feasibility constraints further complicate exploration. Large regions of the space are invalid due to insufficient buffer capacity, bandwidth limitations, or poor resource utilization. As a result, a substantial fraction of evaluations can be spent rejecting infeasible candidates, making efficient identification of valid regions a challenge in itself.

A second practical issue is modeling scope. Several frameworks operate at the layer-level because loop nest structure is explicit and the mapping space is already large [83, 86]. Full-CNN evaluation introduces cross-layer effects such as activation lifetimes, weight reloads, fusion, and

global scheduling. Capturing these effects increases modeling complexity and evaluation cost. Layer-level modeling is therefore often used as a pragmatic compromise, while network-level modeling is adopted when system constraints such as real-time deadlines or tight bandwidth limits dominate.

3.2 PPA Metrics and Associated Cost Components

DSE for DNN accelerators is intrinsically multi-objective. Evaluating a candidate design yields metrics that characterize execution latency, power or energy consumption, and implementation feasibility. These quantities are used either as constraints or as objectives in a trade-off analysis. Under a limited evaluation budget, the evaluator must be explicit about what it predicts and under which assumptions, so that estimates remain consistent and comparable across design points.

A PPA model used for DSE is not intended to reproduce silicon validated results with full accuracy. Its role is to provide early stage estimates that preserve the relative ranking of candidates and expose dominant bottlenecks. In this setting, internal consistency across the explored space can be more valuable than high absolute accuracy at a single operating point.

The relevant scope follows the exploration objective. At the architectural-level, the estimator should capture scaling trends induced by PE array size, memory capacities, and bandwidth constraints. At the mapping-level, it should reflect the impact of tiling, execution ordering, and spatial assignment on reuse and utilization. At the system-level, it should reveal global bottlenecks such as off-chip bandwidth saturation and on-chip communication pressure, which can dominate latency and energy even when operation count is fixed.

3.2.1 Evaluation Objectives and Metrics

Most evaluation workflows report energy or power, latency or throughput, and area, as summarized in Table 3.1. For DSE, three properties matter. Metrics must be comparable across configurations, sensitive to mapping decisions, and interpretable enough to guide search.

Metric	Common units	Notes
Energy	pJ, nJ, μJ	Scope must be stated (per op, per layer, per inference)
Power	mW, W	Average over an inference window, dynamic plus leakage
Latency	cycles, ms, μs	If in time, the clock frequency must be stated
Throughput	GOP/s, TOP/s, FPS	Batch size and I/O conventions must be stated
Area (ASIC)	mm^2	Technology and toolflow dependent, often a constraint
Area (FPGA)	LUTs, FFs, BRAM, DSPs	Target device must be stated
Efficiency	GOP/J, TOP/s/W	Meaningful only when scope and assumptions match

Table 3.1: Primary metrics for DNN accelerator evaluation and their common units.

Energy and power are closely related but describe different aspects of consumption. Energy accumulates over the duration of an inference, whereas power reflects the instantaneous rate at which energy is being spent. This distinction matters because energy determines the cost per inference and battery lifetime, while power constrains thermal dissipation and the power delivery system, especially when consumption is irregular.

Both metrics are commonly decomposed into static and dynamic components. Static power mainly arises from leakage and is largely set by the hardware configuration, so its contribution to energy increases with execution time. Dynamic power depends on switching activity and can vary significantly across phases of execution. In many DNN accelerators, the energy cost is dominated by data movement rather than arithmetic, with memory accesses often costing orders of magnitude more than MAC operations [20]. A mapping that improves reuse therefore reduces energy primarily by lowering memory traffic, and it can also reduce leakage energy by shortening execution time. The effect on power is less direct: concentrating activity into a shorter interval can change how power varies over time during the inference and may increase or decrease average and peak power depending on how activity overlaps in time.

Latency and throughput depend on both compute parallelism and data supply. A common approach estimates cycle counts from the schedule and converts cycles to time using an assumed clock period. This is adequate when frequency is fixed, but it becomes fragile when architectural changes shift timing closure or when implementation constraints change the achievable frequency. For DSE, the key quantity is achieved latency under bandwidth and communication constraints, since these are frequent limiting factors in embedded systems [8].

Area can be both an objective and a feasibility constraint. In ASIC oriented work, it is a continuous footprint that supports cost reasoning. In FPGA oriented work, it is expressed as discrete resource usage, and feasibility depends on whether the design fits the target device. Area is therefore among the metrics for which fast estimators most often diverge from implementation outcomes.

3.2.2 Cost Decomposition: From Workload Events to Technology Costs

Fast PPA estimation commonly follows a two stage decomposition. First, the evaluator derives event counts implied by the mapping on the chosen architecture. These events include arithmetic operations, accesses at each memory-level, transfers across communication links, and delays induced by bandwidth limits or synchronization. Second, it converts event counts into physical metrics by applying calibrated costs, such as energy per access, delay per transfer, or area per storage bit and per compute unit. This separation keeps the event model stable while technology dependent costs are updated through calibration [83, 86, 88].

This decomposition is particularly effective for energy estimation, since energy is additive and can be expressed as the sum of event-level contributions, independent of when events occur. Since data movement often dominates compute cost, frameworks such as Timeloop and Accelergy track traffic per memory-level and apply component energy models calibrated through tools such as CACTI [83, 88, 89]. Power estimation is less direct because instantaneous power depends on the temporal overlap of activity, not only on totals. Capturing time resolved power therefore requires assumptions, or schedule-level information, about when events occur and which components are active concurrently.

Latency is not additive and is typically obtained by modeling a schedule under resource constraints and taking the resulting makespan. Cycle count is typically set by the limiting resource among compute, memory bandwidth, and communication [83]. Increasing reuse can reduce memory traffic but may also reduce parallelism or introduce synchronization, increasing cycles. Conversely, maximizing parallelism can saturate bandwidth and induce delays. Capturing these effects requires the evaluator to model concurrency and overlap, not only aggregate counts. Analytical models often approximate this through bandwidth constraints and coarse schedule reasoning, while cycle accurate simulators represent contention explicitly.

Area and frequency are the metrics for which decomposition is least straightforward. Event counting does not yield area directly, and implementation outcomes depend on memory, interconnect layout, and routing congestion. Analytical estimators can sum component models, but they often miss post place and route effects. This is why synthesis results are commonly used to calibrate fast estimators, or to train surrogates that map architectural parameters to area and frequency while preserving high evaluation speed [90]. A similar approach is often used for power when reliable switching activity is available only through expensive simulation traces.

3.3 Architectural Modeling and Mapping Representations

Cost decomposition relies on event counts. What can be counted, and where it is counted, is determined by the hardware and mapping representations. These representations define the design space, enable feasibility checks, and bound which effects the PPA evaluator can capture reliably.

3.3.1 Hardware Modeling Strategies

In DSE, hardware is modeled either as a fixed architecture with predetermined parameters or as a parametric template whose parameters are exploration variables. A fixed architecture commits to a specific topology, buffer structure, and interconnect, which enables detailed evaluation but limits structural exploration. A template model defines a class of accelerators through a controlled set of architectural variables, such as PE array dimensions, buffer capacities, bandwidth constraints, and interconnect organization [83, 86]. This representation is well suited to systematic exploration because candidates remain comparable under a shared abstraction and evaluation remains tractable.

Hardware models differ in how they encode computation and data movement through the chosen dataflow. A dataflow specifies how computations are assigned to processing elements and how operands are delivered and reused over time. Canonical styles such as weight stationary, input stationary, output stationary, and row stationary correspond to different loop schedules and data movement patterns, and are widely used to reason about energy and performance in convolution accelerators [91, 92]. Row stationary aims to balance reuse of inputs, weights, and partial sums by mapping convolution rows across neighboring PEs, as illustrated in Figure 3.2.

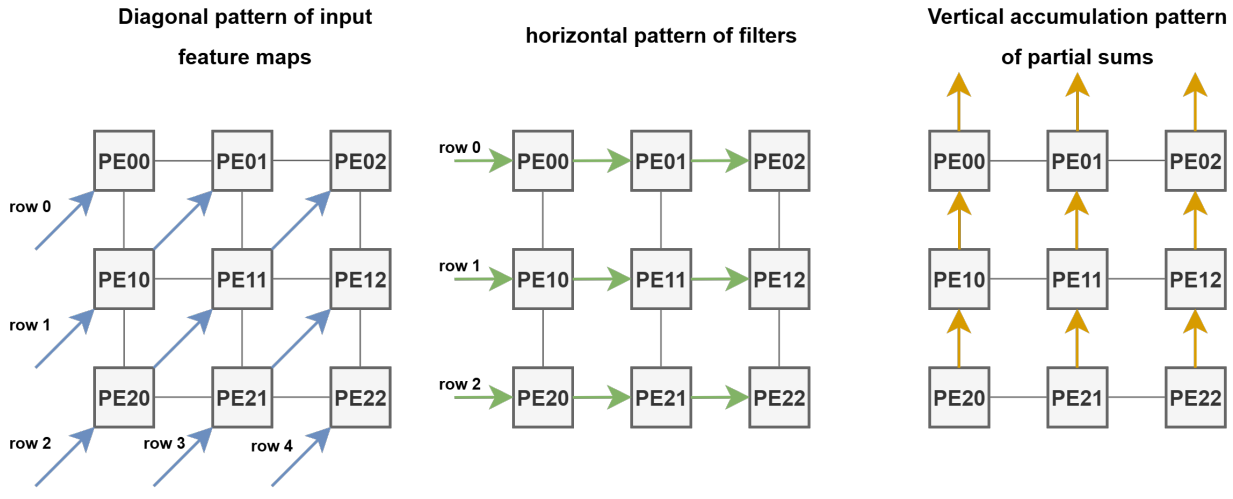


Figure 3.2: Dataflow illustration of Row Stationary dataflow on a 2D PE array.

Many accelerators realize these dataflows on a 2D PE array with local buffers and an on-chip interconnect for operand distribution and reductions. At this scale, the interconnect determines how operands are delivered to PEs and how partial sums are combined. A common abstraction is a Network-on-Chip (NoC), which offers flexible routing and supports a wide range of mappings at the cost of more complex control and contention [93]. Another approach is systolic style nearest neighbor communication, which uses fixed local links and a regular pipelined dataflow. This can deliver high utilization for structured operators, but it reduces mapping flexibility.

Hardware models also differ in how they represent the memory hierarchy and on-chip communication. Many analytical frameworks describe the hierarchy as a set of levels with capacity and bandwidth, and approximate communication through aggregate bandwidth limits or idealized multicast and reduction patterns [83, 86]. This abstraction captures reuse driven traffic effects and scales well, but it becomes less reliable when NoC behavior is a first order bottleneck. NoC designs introduce routing, arbitration, and congestion effects that are difficult to reduce to a single bandwidth value without losing key behavior [93].

Clustered and multi core designs require the model to represent communication across clusters, not only within a local PE group. Templates that include multiple compute clusters, shared and private buffers, and explicit links enable exploration of locality and scalability trade-offs, but they enlarge the parameter space and complicate feasibility checking. Several frameworks therefore restrict heterogeneity to structured variants, or delegate part of the evaluation to simulation or learned surrogates when modeling detail becomes too costly.

Finally, hardware models differ in timing granularity. Some represent resources through capacity and bandwidth constraints and estimate performance through overlap rules. Others approximate schedule execution more explicitly by modeling pipeline stages and synchronization. Higher fidelity models include cycle-level effects and can represent contention and memory pressure directly [87, 94]. This choice shapes which mapping degrees of freedom can be expressed naturally and which bottlenecks can be attributed to architectural parameters.

3.3.2 Mapping Formulations

Most mapping formulations in DSE can be related to a loop nest view of a DNN layer, as shown in Figure 3.3. They differ in which scheduling decisions are exposed explicitly, which constraints are built in, and which costs can be estimated efficiently.

```
# Convolution loops
for o in range(O):                    # Output Channel
  for i in range(I):                  # Input Channel
    for x in range(X):                # Position X
      for y in range(Y):              # Position Y
        for kx in range(Kx):          # Kernel X
          for ky in range(Ky):        # Kernel Y
            ofmap[o, x, y] += ifmap[i, x + kx, y + ky] * kernel[o, i, kx, ky]
```

Figure 3.3: Six-dimensional nested loop formulation for convolution operations.

A common family works directly with loop nest transformations. A layer is expressed as a nest of loops, and the mapping is defined by tiling factors, loop orderings, and assignments of loop dimensions to spatial or temporal execution. This representation is expressive for a wide range of dense operators and structured enough to support systematic search and analytical evaluation, which is why it is central to widely used DSE frameworks [83].

Data centric formulations [86] provide a complementary view. Instead of describing mappings through low level scheduling choices, they focus on how data move and where reuse occurs, making reuse patterns primary objects. This can simplify memory-traffic estimation and clarify reuse trade-offs, but it can abstract away timing detail that matters when contention and synchronization dominate.

Traditional loop-based mappings often impose a uniform tiling structure across operands. Inputs, weights, and outputs exhibit distinct reuse patterns, and a single tiling scheme cannot always exploit them fully. Operand decoupling relaxes this constraint by allowing each operand to adopt an independent tiling configuration. ZigZag [84] implements this approach systematically and shows that the expanded space can yield improved hardware-mapping co-design solutions, at the cost of increased exploration complexity and stricter requirements on evaluation accuracy.

Compiler driven schedules provide another interface. Domain specific languages such as Halide separate algorithm from schedule and express tiling, reordering, fusion, and parallelization at a high level [95]. Frameworks such as Interstellar leverage these constructs to analyze DNN accelerator behavior in a unified representation [96]. This approach supports schedule transformations across operators, but it requires a careful mapping between compiler-level schedules and hardware-level constraints, especially for memory and communication.

Some approaches restrict the mapping space to match a specific architecture and dataflow. Row stationary designs are a representative example, where schedules keep selected data local to maximize reuse [91]. Architecture specific frameworks can exploit this structure to reduce search and improve accuracy within their scope. Agamotto [97] follows this direction by optimizing mappings

under a row stationary style and targeting objectives aligned with that constraint. The downside is reduced generality and the need to revisit the representation when the hardware template changes.

Recent work also explores differentiable representations of mapping decisions. The idea is to relax discrete choices such as tiling and assignment into continuous variables and optimize a smoothed surrogate objective. DOSA [85] uses this strategy for one loop co optimization of mapping and architecture variables. This can reduce the number of evaluations needed, but it requires that the evaluator and constraints remain meaningful under relaxation and rounding.

Finally, some evaluation infrastructures assume fixed mappings or expose limited mapping choices. This is common when the goal is to study system-level behavior under a given execution style rather than to optimize mapping. End to end simulation frameworks such as SMAUG [87] exemplify this approach by prioritizing faithful execution for microarchitectural validation over mapping exploration. This highlights that mapping representation is tied to the modeling goal and to the available evaluation budget.

3.3.3 Scope and Generality Trade-offs

Hardware and mapping representations involve a trade-off between breadth and fidelity. Generic templates and loop-based mappings enable broad exploration across accelerator variants and workloads, and they pair naturally with analytical evaluation [83, 84, 86]. They provide a shared language for comparison by exposing common parameters and constraints. Their limitation is that they often simplify communication and implementation effects, which can create systematic gaps when interconnect behavior or place and route constraints become significant [90, 93].

Specialized representations encode domain knowledge and improve search efficiency within a restricted design space. Imposing a fixed dataflow style can shrink the mapping search space and concentrate exploration on structured variants [97]. This comes at the cost of reduced generality and weaker comparability across accelerator families.

Portability across technology nodes and implementation toolflows adds another constraint. Separating event counting from technology costing enables reuse through calibration, but only if the counted events remain meaningful across process nodes [83, 88, 89]. When new effects become dominant, such as NoC congestion or arbitration overhead, the representation must expose new event types. Calibration alone cannot recover effects that are not represented [93].

3.4 Evaluation Backends and Modeling Frameworks

Evaluation backends determine how hardware-mapping candidates are evaluated under a limited budget, as summarized in Table 3.2. Each backend makes different modeling choices, exposes different bottlenecks, and remains reliable in different operating modes. As a result, the backend strongly constrains which exploration strategies are practical and which architectural effects can be optimized with confidence.

Backend	**Runtime**	**Captures well**	**Limitations**
Analytical counting	ms to s	Reuse and memory-traffic trends, first order cycle models	Approximate overlap rules, limited sensitivity to NoC congestion, no physical effects
Cycle or system simulation	min to h	Contention, arbitration, queues, backpressure	Low speed, accuracy depends on microarchitectural assumptions
Synthesis and place and route	h to day	Area and timing closure in a specific toolflow	Few evaluated points, outcomes depend on toolflow and constraints
Learned surrogate or hybrid	ms to s	Fast prediction once trained, can reflect a target toolflow	Generalization errors outside the training domain, requires refresh and validation

Table 3.2: Comparison of evaluation backends used in DNN accelerator DSE.

3.4.1 Component-Level Building Blocks

Many evaluation stacks build system-level estimates from calibrated component models and combine them with event counts derived from the mapping. CACTI provides representative memory models. Given capacity, banking assumptions, and a technology node, it estimates access time, power, and area, which can be used to derive per access costs [89]. For compute and control, pre-RTL tools such as Aladdin provide architectural-level estimates from high level descriptions and report performance, power, and area at the model-level [98]. When higher fidelity is needed, RTL simulation can generate cycle accurate activity traces that feed downstream power analysis. Synthesis then maps designs to a target library and reports post-synthesis area, timing, and power. Place and route accounts for physical effects such as routing and congestion, and it is therefore a common reference for timing closure and implementation-level PPA. For FPGA targets, Vivado [99] reports resource

utilization and timing after synthesis and implementation. For ASIC targets, Design Compiler provides post-synthesis reports and PrimePower/PrimeTime [100, 101] are used for timing and power analysis after implementation.

These tools are best viewed as building blocks rather than complete DSE solutions. The key design choice is how a framework connects counted events to component costs, and which timing and communication effects are modeled explicitly before costs are applied. This compositional view keeps estimators extensible. New effects can be introduced by adding event types or refining components without rewriting the evaluation stack.

3.4.2 Analytical Evaluation and Cycle-Count Modeling

Analytical evaluators optimize for speed. Analytical evaluators optimize for speed. They translate a given mapping into event counts such as MAC operations, accesses at each level of the memory hierarchy, and on-chip or off-chip data transfers, then convert these counts into performance and power using component models or calibrated per-event costs. Timeloop [83, 102] is a representative example. Given a hardware template and a mapping expressed through loop transformations, it derives buffer occupancy requirements, access counts across the memory hierarchy, and a corresponding execution schedule from which cycle counts can be estimated. Accelergy [88] complements this flow by translating architectural events into energy estimates through a library of component models, commonly parameterized using CACTI [89] for memories. MAESTRO [86] follows a data centric formulation and uses reuse, bandwidth, and buffer constraints to predict performance and energy from dataflow driven traffic and a critical path schedule. ZigZag [84] extends the mapping space through operand decoupling while remaining compatible with fast analytical evaluation.

The main advantage of analytical backends is coverage. They can evaluate large numbers of candidates and therefore support joint hardware-mapping exploration. Their limitation is that some effects are difficult to express through event counting alone. Communication is often approximated through bandwidth constraints or idealized multicast and reduction patterns. Timing is commonly approximated through overlap rules. These abstractions become fragile when routing, arbitration, synchronization overhead, or congestion dominate. In such conditions, analytical models remain useful as a baseline, but they often require calibration or correction to remain predictive for a given architectural family.

3.4.3 Cycle Accurate and System-level Simulation

Simulation backends trade speed for behavioral fidelity. They are particularly valuable when the dominant effects are those that analytical models struggle to capture accurately, such as pipeline hazards, queueing, backpressure, cache behavior, and contention in interconnect and memory system. System simulators such as gem5 [94] provide a flexible platform to model full-system behavior, including memory hierarchies and interconnect, and can expose bottlenecks that emerge only at system scale. SMAUG [87] illustrates full-stack infrastructures that integrate system simulation and accelerator models to evaluate deep learning workloads end-to-end, with emphasis on cycle-level realism rather than large scale exploration. Even when simulation is not used as the per candidate evaluator, it is commonly used for validation and for generating labeled data to train or calibrate faster estimators.

The main limitation is runtime. When evaluation takes minutes or hours rather than milliseconds, broad exploration becomes infeasible. Simulation-based workflows therefore rely on targeted studies, selective sampling, or sample efficient optimization methods, often guided by faster evaluators.

3.4.4 Synthesis-based Estimation and Implementation Ground Truth

Synthesis-based evaluation provides toolflow grounded estimates by mapping designs to a specific technology and implementation flow. Post-synthesis reports provide early area and timing indicators, while post place and route results better reflect implementation-level timing closure and congestion effects. For FPGA targets, Vivado [99] reports resource utilization and timing after implementation, and power analysis can incorporate switching activity obtained from simulation traces. For ASIC targets, Design Compiler performs logic synthesis and technology mapping to produce area and timing numbers, while PrimePower/PrimeTime [100, 101] supports timing and power analysis on implemented netlists. Frameworks such as MAGNet [90] incorporate synthesis to anchor predictions in implementation reality and to assess feasibility under technology constraints. Architecture specific mapping approaches [97] targeting FPGA deployment similarly rely on synthesis because frequency and routing feasibility depend strongly on tool decisions.

The limiting factor is cost and infrastructure. Synthesis and implementation can take hours per point and depend on target specific libraries, resulting in tightly coupled evaluations that do not scale to broad DSE. In practice, these tools are most useful as sources of ground truth for calibration, validation, and supervised labeling of faster evaluators.

3.4.5 Learned Surrogates and Hybrid Estimators

Learning-based approaches use data to predict PPA metrics or to recommend promising hardware-mapping choices. Training data is typically produced by evaluating a limited number of design points with a higher fidelity backend, after which the model can be queried at low cost during exploration [90, 103].

A common pattern is the surrogate loop. A small set of candidates is evaluated using simulation or synthesis. A surrogate is fitted on these observations and then used to steer subsequent sampling. As exploration reaches new operating regions, additional high fidelity points are added to refresh calibration and reduce error where it matters most [90].

The main limitation is generalization outside the operating regions covered by the training data. When exploration proposes new mappings, different NoC operating points, or shifted implementation constraints, prediction quality can degrade. Feature design and validation strategy therefore become central, and periodic cross checks against higher fidelity backends are often needed to keep the proxy aligned with implementation behavior.

Hybrid estimators mitigate part of this limitations by combining an analytical backbone with a learned correction. The analytical component captures structural invariants such as operation counts and reuse driven traffic, while the learned component models residual effects that are difficult to express analytically, including routing overhead and contention. This combination preserves interpretability and scalability while improving accuracy in regions covered by calibration data.

3.4.6 From Backends to Full Frameworks

Complete modeling frameworks connect hardware and mapping representations to one or more evaluation backends and expose an interface that exploration algorithms can query. In practice, several integration patterns recur. They are defined by which evaluator is used for each candidate during exploration and by which higher fidelity source is used for calibration or validation, as summarized in Table 3.3.

Analytical stacks are the default choice when broad coverage of a coupled hardware-mapping space is required. They rely on structured representations that make reuse and memory traffic explicit and can therefore evaluate many candidates quickly.

Pattern	Per candidate evaluator	Typical use
Analytical stack	Analytical evaluator	Broad search and pruning, validate a small set of points with higher fidelity
Implementation grounded	Simulation or synthesis	Feasibility focused studies, highly selective exploration
Hybrid or amortized	Surrogate or corrected evaluator	Fast exploration with periodic labeling and calibration refresh

Table 3.3: Common integration patterns, defined by the evaluator queried for each candidate during exploration.

Timeloop [83] with Accelergy [88] exemplifies this approach by combining loop-based mappings with analytical access counting and component energy models. MAESTRO [86] follows a similar approach with a data centric formulation designed for scale. ZigZag [84] expands the expressible mapping space through operand decoupling while remaining compatible with rapid evaluation. These approaches are most reliable when reuse and memory hierarchy traffic dominate first order costs, and they become less predictive when detailed communication contention or implementation effects dominate.

Implementation grounded stacks sit at the other end of the spectrum. They prioritize accuracy by using simulation or synthesis during exploration, which exposes contention, backpressure, and toolflow specific timing and area [87, 90, 94]. The cost is that the evaluation budget collapses and exploration becomes inherently selective.

Hybrid and amortized stacks connect these extremes. High-fidelity runs provide labels or calibration points, while fast proxies restore speed for exploration [103]. This can be achieved with learned surrogates, with analytical models corrected by residual predictors, or with differentiable relaxations that enable one loop co-optimization under a smooth proxy objective [85]. The key question becomes trust. A fast proxy is useful only if it preserves the effects that matter over the regions of the design space that the search will visit.

Overall, the choice of backend and integration pattern sets the practical balance between coverage and accuracy. Analytical evaluation supports large scale exploration but can hide communication and implementation artifacts. Simulation and synthesis expose these effects but limit the number of points that can be tested. Hybrid workflows mitigate the gap, but they introduce requirements in calibration strategy, dataset coverage, and out of distribution monitoring, which become central to reliable exploration.

3.5 Exploration Algorithms and Search Strategies

Exploration algorithms address two practical constraints. First, the joint hardware-mapping space is large and highly constrained by feasibility and implementation constraints. Second, each candidate evaluation consumes a limited budget. Consequently, search procedures aim to minimize evaluations spent on infeasible points and to concentrate effort on regions that improve the objective or refine the trade-off surface. The discussion below groups strategies by how they use evaluations and how they handle feasibility.

3.5.1 Budget and Feasibility Constraints

Query budget shapes the exploration strategy. When candidate evaluations are cheap, search procedures can rely on broad sampling and iterative improvement [83, 86]. Conversely, when candidate evaluations are expensive, exploration becomes selective and emphasizes sample efficiency [87, 90]. In both contexts, feasibility remains a dominant issue. Buffer capacities, bandwidth limits, and alignment requirements exclude large regions of the space, so naive exploration wastes evaluations on invalid configurations. Effective methods therefore incorporate constraints directly into the parameterization and search operators, or rely on fast feasibility checks and pruning mechanisms. This challenge becomes more acute for NoC-based designs, where feasibility and performance depend not only on static resource capacities but also on contention patterns that may require deeper evaluation to characterize [84, 93].

3.5.2 Black-Box Optimization

Black-box methods treat the evaluator as an oracle and make few assumptions about smoothness or convexity. This is well matched to hardware-mapping DSE, where the space is mixed discrete and small parameter changes can trigger abrupt changes in feasibility or cost. In practice, the key differentiator is often constraint handling rather than the optimizer family.

Evolutionary search is widely used because it handles discrete parameters and multi-objective trade-offs, but generic genetic operators, such as mutation and crossover, can often generate infeasible designs when constraints are not respected. GAMMA [86, 104] illustrates how evolutionary search can be coupled to fast evaluators for mapping exploration. DiGamma [105] shows the benefit of domain-aware operators that bias mutations and recombinations toward constraint-satisfying candidates, increasing the fraction of valid samples and making joint exploration more practical.

Bayesian optimization targets sample efficiency by fitting a probabilistic surrogate and selecting new points through acquisition functions. This is attractive when each evaluation is expensive, for example in synthesis grounded workflows such as MAGNet [90]. Reinforcement learning provides another route by learning policies over sequential decisions. It can be effective when evaluation is fast enough to support training and when decision structure transfers across workloads [86, 106].

3.5.3 Gradient-based and Differentiable Methods

Differentiable approaches reduce the number of evaluations by using gradient information to guide updates. Because tiling factors and architectural choices are discrete, these methods rely on relaxations that introduce continuous proxies and smooth constraint penalties. DOSA [85] is representative of this direction and performs joint optimization over relaxed mapping and architecture variables in a single loop. The main difficulty is the gap between the relaxed formulation and the discrete design space. Rounding can reintroduce constraint violations or shift bottlenecks, so practical utility depends on whether the relaxation preserves the structure that determines latency and energy.

3.5.4 Learned Predictors and Surrogate Assisted Search

Learning-based approaches can recommend promising configurations or approximate objective values, which are then refined using a small number of high-fidelity evaluations. AIrchitect [103] follows this strategy by learning near optimal decisions from data generated from fast evaluators, enabling low-cost inference at deployment time. However, prediction robustness remains a challenge, as changes in architecture templates, interconnect regimes, or constraints can significantly degrade accuracy.

A more conservative use of learning is surrogate-assisted optimization. In this paradigm, a learned surrogate replaces an expensive evaluator within the exploration loop, while periodic validation against simulation or synthesis keep the proxy aligned. This approach is particularly relevant when synthesis is the reference, because the budget of direct synthesis calls limits unguided search [90].

3.5.5 Multi Fidelity Optimization

Multi-fidelity optimization combines evaluators with different cost and fidelity. Low-fidelity models support broad ranking and pruning, while higher fidelity simulation or synthesis is invoked selectively to validate candidates, recalibrate model parameters, or generate training labels for learned surrogates. Polaris [107] exemplifies this approach by coupling fast analytical estimates with a limited

set of RTL-level measurements and applying transfer learning to predict high fidelity metrics from low fidelity features. The central difficulty lies in determining when low fidelity predictions remain trustworthy and when escalation to higher fidelity evaluation is required. This decision is particularly difficult for communication dominated designs, where contention and arbitration effects can dominate end-to-end cost while remaining weakly reflected by coarse abstractions.

3.5.6 Joint versus Nested Optimization

A structural choice is whether to optimize hardware and mapping jointly or via nested loops. Nested strategies fix one dimension and optimize the other, for example by selecting a hardware candidate and then searching for its best mapping. This decomposition reduces complexity and can be effective when one subproblem is small or when evaluations are inexpensive enough to support an inner mapping search. However, nested approchaes can miss co-design effects where small hardware changes enable different mapping strategies and shift bottlenecks. Joint strategies search the coupled space directly, using constraint aware search operators [105] (mutation and crossover) or differentiable formulations [85]. While this enables richer co-design, it also increases sensitivity to representation quality and to evaluator fidelity.

3.5.7 Multi-Objective Considerations

Exploration is inherently multi-objective because energy, latency, and area trade-off. Some approaches reduce the objective to a scalar metric such as energy-delay product or weighted sums, which simplifies optimization but requires careful weighting [83]. Others treat some metrics as constraints, for example enforcing an area or power budget while optimizing latency. Pareto-based reasoning provides a direct view of trade-offs by identifying non dominated solutions. It generally requires many evaluations to reconstruct a diverse frontier, which favors analytical or surrogate-based evaluation [90, 103]. When evaluations are expensive, Pareto reasoning is often approximated by focusing on a small number of operating points.

Overall, search procedures must match the available evaluation budget while remaining constraint aware. Fast evaluators enable broad exploration, while expensive evaluators favor sample efficient and guided strategies. Hybrid and multi fidelity workflows connect these approaches, but they require calibration strategies and validation protocols that maintain reliability as exploration moves across the design space.

The effectiveness of search algorithms is tightly coupled to evaluator behavior. Noisy, biased, or low-fidelity estimators can mislead optimization, particularly in multi-objective settings where small

ranking errors accumulate across iterations.

3.6 Synthesis and Research Gap

This chapter reviewed design space exploration pipelines for DNN accelerators under a coupled hardware-mapping formulation. Hardware choices restrict which mappings are feasible, while mapping choices determine whether execution is limited by computation, memory movement, or on-chip communication. At scale, analytical approaches enable fast, structured evaluation across large spaces [83, 86, 88]. Broadening the mapping space can expose improved hardware-mapping pairs, but it also increases evaluation cost and raises the burden on the search itself [84]. Detailed simulation and implementation flows capture effects that analytical abstractions often omit, but they are too expensive to serve as the main evaluator inside large combinatorial searches [87, 90, 94].

The central trade-off is speed versus implementation relevance. Analytical models scale because they omit fine grained timing and physical implementation effects. This is acceptable when first order costs are driven by memory hierarchy traffic and operand reuse. It becomes less reliable in distributed architectures, where effects that are minor in centralized designs can become prominent. This shift is increasingly common as accelerators move toward distributed scratchpads and on-chip fabrics that behave like networks. In NoC architectures, arbitration and congestion depend on both mapping decisions and network parameters. These mechanisms can change latency and power substantially, yet they are difficult to capture with bandwidth only models without losing speed that makes analytical evaluation attractive. Calibration can reduce error for a fixed template, but it typically relies on expensive reference runs and may not transfer when the architecture or mapping strategy changes.

Portability across targets introduces another source of mismatch. Even with a tuned analytical evaluator, changing the implementation platform can shift achievable frequency and area because timing closure, routing overhead, and resource fragmentation change the physical outcome. Post-implementation results therefore remain the most reliable reference for feasibility and timing, but their cost rules out using them directly to drive exploration. This motivates workflows that keep a fast evaluator in the loop and invoke high fidelity tools selectively for validation and periodic recalibration. It also motivates learned surrogates that amortize expensive reference evaluations [103]. Differentiable and single loop optimization approaches strengthen the same requirement. The PPA evaluator model must remain fast and smooth enough to support iterative updates [85].

These points expose a specific gap for NoC-based dataflow architectures. Current analytical frameworks provide strong abstractions for reuse and memory hierarchy reasoning, but they offer

limited support for contention, arbitration, and implementation driven timing while keeping high-speed evaluation capabilities. High fidelity simulation can represent these mechanisms, but it does not scale to broad joint exploration. Post-implementation evaluation provides the closest view of feasibility, but it is prohibitive for iterative search. What is missing is a pipeline that connects these fidelity levels in a controlled way, so that exploration stays fast while remaining sensitive to the effects that dominate in distributed, communication heavy designs.

More generally, analytical models, simulation, and implementation flows are often viewed as separate sources of information rather than as parts of one estimator. In many workflows, fast models drive the search and a small number of detailed runs are used for spot checking, but calibration is rarely updated systematically as the search moves to new regions of the design space. This is most problematic for communication dominated designs, where contention and schedule interactions can dominate latency and energy while remaining weakly represented by coarse abstractions. Purely learned predictors face a related issue. Reliability degrades when explored designs differ from the training distribution.

The next chapter develops a hybrid direction. It collects implementation-level measurements across an architectural family and uses them to train predictive models that can be queried at near analytical speed. The goal is to retain the speed of analytical exploration while incorporating NoC and implementation effects that are otherwise hard to model directly. Simulation and implementation are employed as reference data sources for calibration and learning, not as the PPA evaluator used for every candidate. This pipeline aims to narrow the speed versus fidelity gap for NoC accelerator exploration and to support subsequent joint hardware and mapping optimization at scale.

Chapter 4

Methodology: PANDA Framework for Learning-based PPA Estimation

4.1 PANDA's Approach and Contributions

The PANDA framework addresses the challenges of modeling and optimizing the mapping of CNNs onto NoC-based dataflow architectures. Our framework consists of two main components (as shown in Figure 4.1): a mapping model and a ML-based PPA estimation model. The design space exploration engine is left as future work as it is still a work in progress.

PANDA takes three primary inputs: a DNN model imported from the N2D2 [108] tool (NEUROKIT2E project [109]), which provides both the network topology and trained weights, along with input feature map data. A dataflow specification that defines the high level loop ordering and determines which data remains stationary in memory (as shown in subsection 3.3.1). A target architecture based on the NoC-based dataflow architecture modeled by CEA [110], with dimensions adapted to the size of the DNN model. This parameterizable architecture serves as a research prototype for exploring multiple design points. This NoC-based dataflow architecture is a completely synthesizable VHDL implementation and currently supports multiple convolutional layers (standard convolution, depthwise convolution, and pointwise convolution), fully connected layers, activation function, and pooling operations.

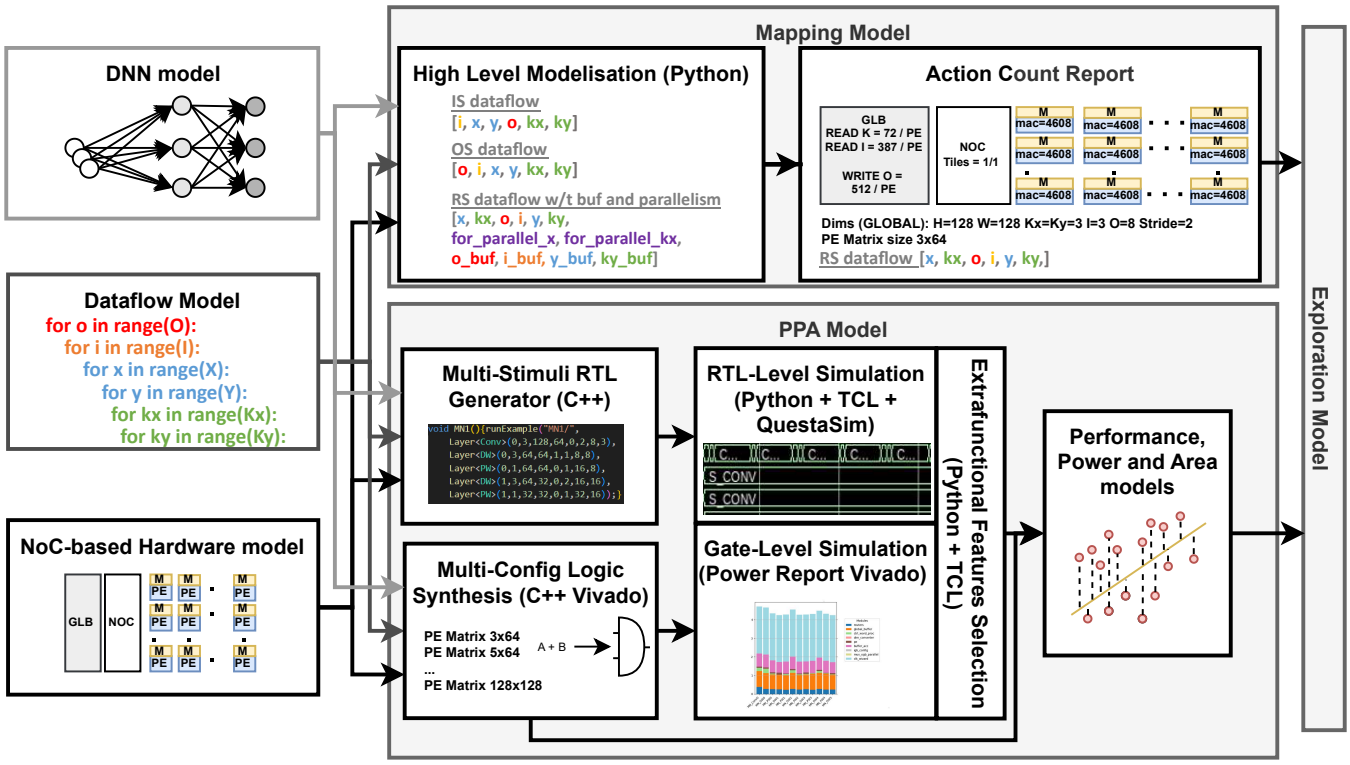


Figure 4.1: PANDA Framework Workflow for CNN Mapping to a NoC-based dataflow Architecture and PPA Estimation

The mapping model operates on the basis of high-level abstractions and the user specifies only theoretical parameters such as memory capacity, degree of parallelism, and dataflow order. No description of register transfers or explicit operand movements is necessary. The model provides a simple interface for specifying mappings through dimension vectors, such as $[o_chan, i_chan, i_x, i_y, k_x, k_y]$ for an output-stationary schedule (as shown in Figure 3.3), or $[o_x, k_x, o_chan, i_chan, o_y, k_y]$ for a row-stationary style.

In this thesis, we target a NoC-based dataflow architecture whose execution model is expressed through the Adaptive Dataflow Pattern (ADP) abstraction [110]. We therefore use ADP as a reference dataflow specification, which consolidates multiple canonical dataflow.

From these specifications, the model generates comprehensive action count reports that quantify MAC and accumulation operations, memory accesses, and per PE data requirements. These reports serve multiple purposes: benchmarking the mapping efficiency, validating the correctness of proposed mappings against RTL simulations.

The PPA model is trained using an automated data generation pipeline that amortizes the cost of high-fidelity evaluation into an offline phase, enabling fast surrogate-based scoring of candidate hardware mapping configurations. PANDA implements a C++ based multi stimulus RTL generator

that creates testbenches and produces memory initialization files containing properly formatted DNN data, which are then loaded into the architecture's block RAMs according to the specified dataflow. This generator supports both single layer and multi layer configurations, handling both input and hidden layers appropriately. Pre synthesis simulations in QuestaSim, automated through Python and TCL scripts, enable rapid generation of cycle accurate latency data and optional VCD dumps for detailed timing analysis. Power is also investigated through activity-based reporting in Vivado [99]; however, limitations in trace to netlist reconciliation can reduce workload dependent variation, which is discussed in Chapter 5. In parallel, a Vivado-based automated synthesis flow generates area metrics (LUTs, BRAMs, DSPs, . . .) for numerous architectural configurations. Together, these tools produce datasets for performance (latency) and area modeling (and power reports where available), which are then used to train ML-based models.

The core innovation of PANDA lies in narrowing the accuracy speed gap in latency and area estimation for NoC-based accelerators through ML-based models trained on implementation-level data from RTL simulations and logic synthesis workflows. The framework delivers evaluation speeds approaching analytical methods [83, 84, 86] while accounting for implementation dependent phenomena including routing delays, resource contention, and FPGA specific constraints typical of synthesis driven approaches [90, 97]. Rather than directly predicting optimal configurations [103], PANDA provides an evaluation backend that can be coupled to iterative exploration strategies, investing upfront in automated dataset generation to enable low marginal cost per candidate across the current target DNN family (MobileNet).

4.2 NoC-based Dataflow Architecture

This section introduces the fully parametric NoC-based dataflow architecture developed by the CEA, illustrated in Figure 4.2. This experimental platform serves as the foundation of the PANDA framework [93].

Designed for CNN inference on resource-constrained embedded vision systems, the architecture features a PE grid connected via a mesh NoC, integrated routers, distributed I/O buffers, and dedicated blocks for activation and pooling, all orchestrated by a global controller. Its modular design enables PANDA's automatic generation flow to produce a wide range of architectural instances. By varying RTL parameters such as grid size and memory configuration (see Table 4.1), PANDA systematically explores the design space to match performance, area, and power constraints while maintaining support for diverse neural network layers without architectural redesign.

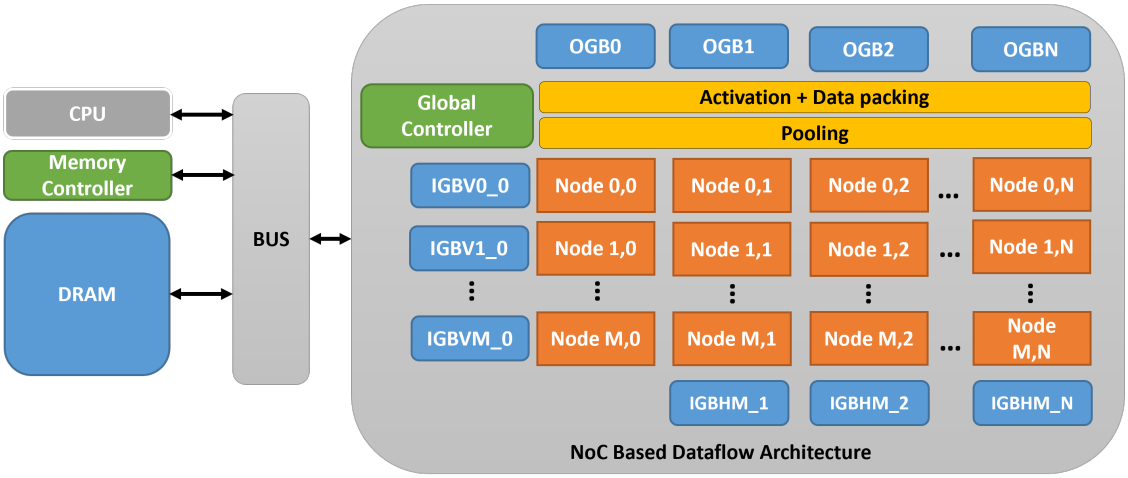


Figure 4.2: Overview of the NoC Dataflow-based Architecture for CNN processing.

4.2.1 Parametric Organization and Scalability

The architecture and its modules dimensions are defined by a set of parameters, as shown in Table 4.1. It comprises a two-dimensional grid of nodes, where each node couples a processing element with a router (as shown in Figure 4.2). In the current RTL, the grid dimensions are tied to layer spatial parameters through noc_size_x = filter_width and noc_size_y = output_width. This choice fixes the number of nodes along X and Y, and therefore directly shapes spatial parallelism as well as communication distances across the fabric. The inter-router connectivity is selected through the mesh/torus/ring encoding, which impacts latency and congestion behavior depending on the workload.

Input Global Buffers (IGB) and Output Global Buffers (OGB) constitute another parametric subsystem whose capacity and spatial distribution are set at generation time (as shown in Figure 4.2). Output banking is controlled by the output_width parameter, which can be extended to account for padding. On-chip buffer depths are configured using address width parameters: addr_width_n for top-side memories (OGBX), addr_width_h for bottom-side memories (IGBHX), and addr_width_v for left-side memories (IGBVX). Data packing is managed via pack_en and pack_size, enabling multiple data items to be grouped per transfer to reduce NoC transactions, at the cost of extra packing logic.

Each router uses FIFOs to buffer traffic at its interfaces (see Figure 4.5). The FIFO depth is configured with fifo_depth and can be adjusted based on burst workloads and expected contention to reduce memory pressure (as shown in Figure 4.2).

RTL parameter	Range	Definition
filter_width	$\{3, 5, 7\}$	NoC X dimension via noc_size_x
output_width	$[8, 256]$	NoC Y dimension via noc_size_y
data_width	$\{4, 8, 16, 32, 64\}$	Datapath bit width for computation and NoC transfers
addr_width_n, addr_width_h, addr_width_v	$[1, 24]$	Address widths that set on-chip buffer depths (depth $= 2^{addr_width}$)
fifo_depth	$[1, 4096]$	FIFO depth at router interfaces
noc	mesh, torus, ring	NoC topology selection
pack_en	$\{0, 1\}$	Enables data packing per transfer
pack_size	$\{0, 1, 2, 3\}$	Number of data items packed per transfer

Table 4.1: Main RTL parameters exposed by the architecture. Ranges are indicative and should be refined from RTL constraints.

4.2.2 Distributed Control Methodology

The architecture employs a two-level distributed control scheme, combining a global controller with locally autonomous modules. This design enables asynchronous local execution while maintaining global coherence through well placed synchronization mechanisms. The control methodology remains consistent across all synthesized instances, ensuring that the architecture's configurable nature does not compromise control integrity, regardless of the configuration generated by the software.

At the highest level, the Global Controller (GC) orchestrates architecture wide operations by decoding layer specific configuration parameters stored in a dedicated memory. This parameter driven approach enables runtime layer wise reconfiguration without hardware modification, supporting diverse CNN topologies.

The GC (Figure 4.3 (left)) operates as a Finite State Machine (FSM) with five states. Upon receiving a trigger signal, it transitions from IDLE to LOAD_config, where it loads configuration words from the Input Global Buffer Configuration (IGB_config). These words contain essential information about the current layer including ifmap width, filter width, number of filters, number of ifmap channels, stride, padding, and shift values. The GC then sequences through LOAD_filters and LOAD_ifmaps states to coordinate data distribution across the architecture before entering the COMPUTE state. It remains in this state until all Processing Elements (PE) complete their calculations, after which it will go back to either LOAD_filters, LOAD_ifmaps to process further in the layer or LOAD_config to process the next layer.

Each PE incorporates a local FSM that operates independently while maintaining synchronization with the global control flow (as shown in Figure 4.3 (right)). This controller continuously monitors its register files through a five state FSM. When valid operands are available and a start signal is received from the GC, it autonomously sequences through filters and ifmaps loading phases. The asynchronous execution model enables computation to begin immediately upon data availability, effectively hiding data movement latency. During the PSUM_ACC state, each PE sends partial sums to its northern neighbor while accumulating results from southern PEs, creating an event driven dataflow. The last active PE in specific array positions signals the GC for global state transitions, ensuring synchronized progress across the entire PE array.

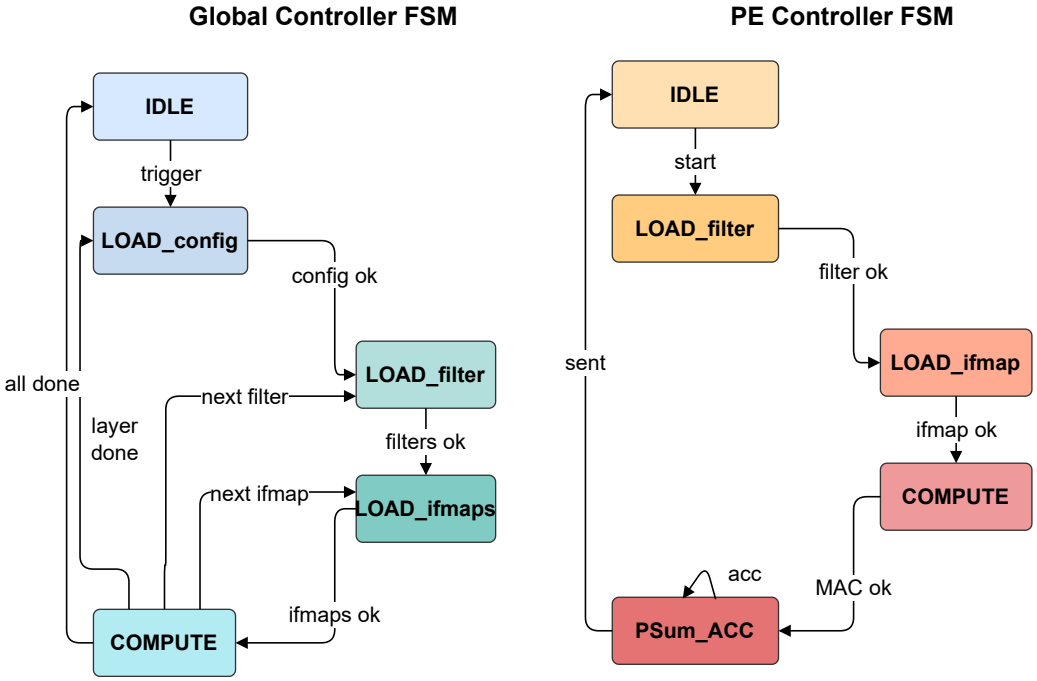


Figure 4.3: Finite State Machines of the two-level distributed control architecture: Global Controller FSM (left), PE Controller FSM (right)

Within each router, control is realized through distributed port local logic rather than an additional monolithic state machine. For each input port (North, East, West, South, Local), a compact combinational routing function inspects the header flit, including destination coordinates (X/Y), the control word, and boundary and connectivity conditions, and generates requests for the feasible output directions. Each output port then applies a local arbitration policy (e.g., round robin) to resolve contention, with re-arbitration performed for each control/data pair. As a result, router operation is predominantly combinational and naturally supports concurrent communication across all five ports.

The synergy between these control mechanisms enables a highly efficient execution model. The GC provides high level coordination, while PE controllers and router local control logic operate with substantial autonomy, reducing the need for frequent global synchronization. This combination of centralized orchestration and distributed decision making allows the architecture to scale to larger arrays while preserving the deterministic behavior required for real-time operation.

4.2.3 Processing Element Architecture

The PE is the basic compute tile of the accelerator and implements a self-contained datapath for convolution (as shown in Figure 4.4). It combines local operand storage, an 8-bit multiply-accumulate unit, partial sum accumulation, and a lightweight controller implemented as a finite state machine (PE Controller FSM). During execution, the PE receives activations and weights, performs MAC operations, accumulates partial results, and outputs computed values.

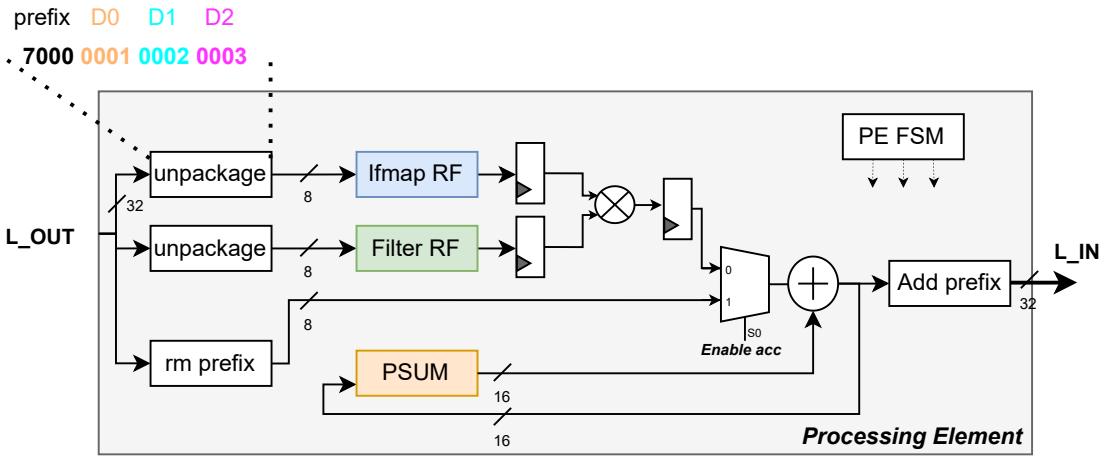


Figure 4.4: Processing Element internal architecture.

At the array-level, the accelerator is organized as a grid of these PEs. Each PE is paired with a node router, so computation and communication are co located at the tile granularity. This organization supports distributed dataflow execution, where data and partial results move through the array as the computation progresses.

Inside each PE, operands are staged in two dedicated register files. The Filter RF stores convolution weights, while the Ifmap RF stores input feature map values. Keeping both operands in local storage reduces off-PE transfers and enables the data reuse patterns typical of convolutional workloads.

These locally stored operands feed the compute engine, which is centered around an 8-bit integer

MAC unit. Products are accumulated into a dedicated partial sum register (PSUM). The PSUM supports both intra-PE accumulation for locally computed convolutions and inter-PE accumulation when a computation is partitioned across multiple PEs, enabling flexible mappings across different layer shapes and parallelization strategies.

To sustain this compute flow at the array scale, each tile relies on its paired node router to exchange data with neighboring tiles. The node router provides five bidirectional ports (North, South, East, West, and Local) and connects the PE to the NoC. Transfers use data_width bit flits grouped into fixed size packets composed of two flits. The first flit is a routing control word that identifies the communication primitive and carries the destination coordinates. The second flit carries payload data and reserves its most significant byte as a compact control prefix.

The packet fields are parametric and depend on the instantiated array dimensions and packing configuration. The routing flit contains a 4-bit operation code (OP_C), followed by the destination coordinates xCoord and yCoord, and the remaining bits set to zero. The coordinate widths are derived from the NoC dimensions, with $xLen = \lceil \log_2(\text{filter_width}) \rceil$ and $yLen = \lceil \log_2(\text{output_width}) \rceil$. In the payload flit, data packing is controlled by pack_en and pack_size. When packing is enabled, up to pack_size 8-bit elements are packed into the payload field; otherwise, a single 8-bit element is transmitted and unused payload bytes are set to zero. Arbitration resolves contention at each router output port, and the routing logic performs the required control prefix handling when injecting and receiving packets in the NoC.

This organization enables efficient layer execution across the PE array. Data loading in one PE can overlap with computation in another and with result forwarding in a third, improving overall throughput while preserving the flexibility required to map diverse neural network layers.

4.2.4 AINoC Communication Infrastructure

The AINoC [111] is designed to support a wide range of layer dependent communication patterns. Unlike fixed systolic arrays, it is not confined to a single predetermined dataflow. The architecture instantiates a 2D array of routers organized as a noc_size_x $\times$ noc_size_y grid. Each node router is paired with a PE and exposes dedicated bidirectional local links (L_IN/L_OUT). This organization lets computation proceed while data is exchanged across the network.

Each router uses a packet switched organization with five independent ports: Local, North, South, East, and West (as shown in Figure 4.5). Upon reception, packets are parsed to extract routing information from the control prefix. They are then forwarded to the selected output direction. When multiple inputs contend for the same output, a local arbitration stage resolves conflicts before transmission. This port local distributed control enables concurrent activity on all ports. It also

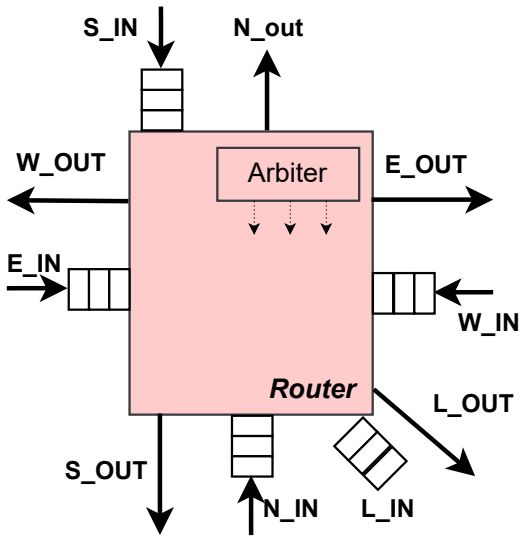


Figure 4.5: Internal architecture of the AINoC router showing the five communication ports.

preserves deterministic routing behavior, which bounds communication latency during inference.

Connectivity between routers is selected at synthesis time through a parametric topology choice. The AINoC supports three options: mesh, torus, and ring. These topologies trade off wiring cost, path diversity, and traffic balance:

- Mesh connects each router to its cardinal neighbors. Routing follows an X-first policy to avoid cyclic dependencies. External I/O ports are exposed at the grid boundaries through the directional links (L_IN/L_OUT(Local PE interface), N_IN/N_OUT(North), S_IN/S_OUT(South), E_IN/E_OUT(East), and W_IN/W_OUT(West)).
- Torus extends the mesh with wrap around connections between opposite edges. Boundary routers connect to their counterparts on the other side. This increases path diversity and reduces edge effects. It can improve load balance under uniform traffic.
- Ring forms unidirectional circulation paths along rows. Multiple rings operate concurrently. This supports streaming style communication and can efficiently pipeline sequential transfers.

On top of the selected topology, the router supports communication primitives that match common DNN data movement patterns. Unicast provides point to point transfers. Multicast enables one to many distribution along horizontal, vertical, and diagonal directions. This is useful for distributing shared operands such as weights across PE rows or columns. Broadcast supports one to all transfers when global availability of a value is required.

This combination of parametric topology selection and flexible communication primitives allows the architecture to adapt the NoC to the target workload and deployment constraints. The mapping framework can explore these options during design space exploration and select the configuration that minimizes communication latency and power for the chosen dataflow.

4.2.5 Memory Organization

The memory architecture employs a hierarchical approach with distributed storage elements (as shown in Figure 4.6). Global buffers are strategically positioned around the PE grid to match the dominant dataflows: Input Global Buffers Vertical (IGBV) on the left side primarily store filter weights for reuse, Input Global Buffers Horizontal (IGBH) at the bottom store input feature maps, and Output Global Buffers (OGB) at the top collect computed results. The OGB are also reused as an input feature maps buffer for the CNN hidden layers.

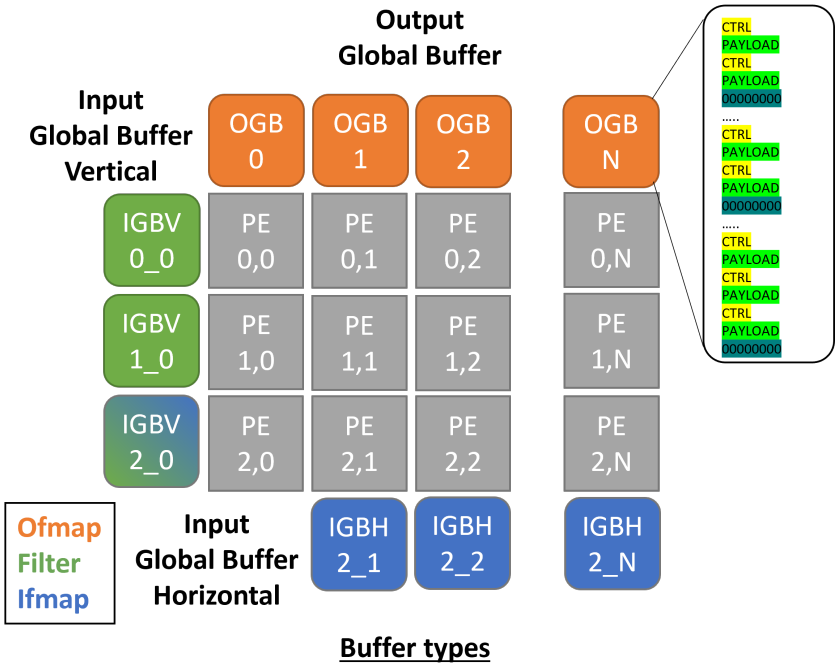


Figure 4.6: Distributed memory organization showing Global Buffers interfacing with the PE grid example for standard convolution with $stride = 2$.

In addition, the left side buffers are not all identical in role. While most IGBV instances are dedicated to filters (green in the figure), the last IGBV is shared between filters and ifmaps, which is why it is rendered with a green to blue transition (as shown in Figure 4.6). This shared buffer is used to increase locality for workloads where the boundary between weight reuse and activation reuse is

advantageous.

Distributed memories interface with the PE grid exclusively through edge nodes. With this placement strategy, data distribution remains efficient and predictable: vertical buffers enable, for example, horizontal multicast of filter weights (and when needed ifmaps) across rows, while horizontal buffers support vertical or diagonal movement patterns depending on the operation type. As a result, each data element fetched from external memory is maximally utilized across multiple computations before being replaced.

To support these transfers, data is handled as packet streams (as shown in subsection 4.2.4). Each transfer is encoded as a fixed size packet made of two 32-bit words: a CTRL word (routing control, primitive ID, destination coordinates) followed by a PAYLOAD word (data, with its most significant byte reserved as a compact control prefix). The trailing `0000000` words correspond to explicit alignment/padding in memory, used to preserve burst alignment and/or delimit packet sequences depending on the protocol (as shown in Figure 4.6 top-right).

Finally, the parametric memory sizing allows design space exploration to find optimal configurations balancing storage capacity against area overhead. The exact memory reduction achieved depends on the synthesized grid dimensions and allocated register file sizes, with larger local storage enabling greater data reuse at the cost of increased silicon area. This flexibility enables PANDA to produce accelerator instances tailored to specific application memory access patterns and constraints.

4.2.6 Adaptive Dataflow and Execution Model

The architecture employs ADP [110] strategies that adjust communication and memory access patterns based on layer type, stride configuration, and padding configuration. These adaptations represent the configuration implemented in hardware enabling the PANDA to optimize data generation for each specific layer of the CNN.

Filter weights are systematically stored in the left side vertical global buffer memories, enabling horizontal parallel multicast across PE rows (as shown in Figure 4.7(b)). This consistent filter distribution strategy applies across all layer types, maximizing weight reuse within each row.

For example, in ADP mode for a depthwise convolution hidden layer with stride-1 and padding-0 (as shown in Figure 4.7(a) and Figure 3.2), input feature maps stored in the upper horizontal memories are distributed using a diagonal southwest multicast. This diagonal dataflow maximizes spatial reuse while each PE processes its own spatial location. With stride-2, the distribution shifts to vertical north to south multi-unicast, reflecting the reduced overlap between neighboring computations of this layer.

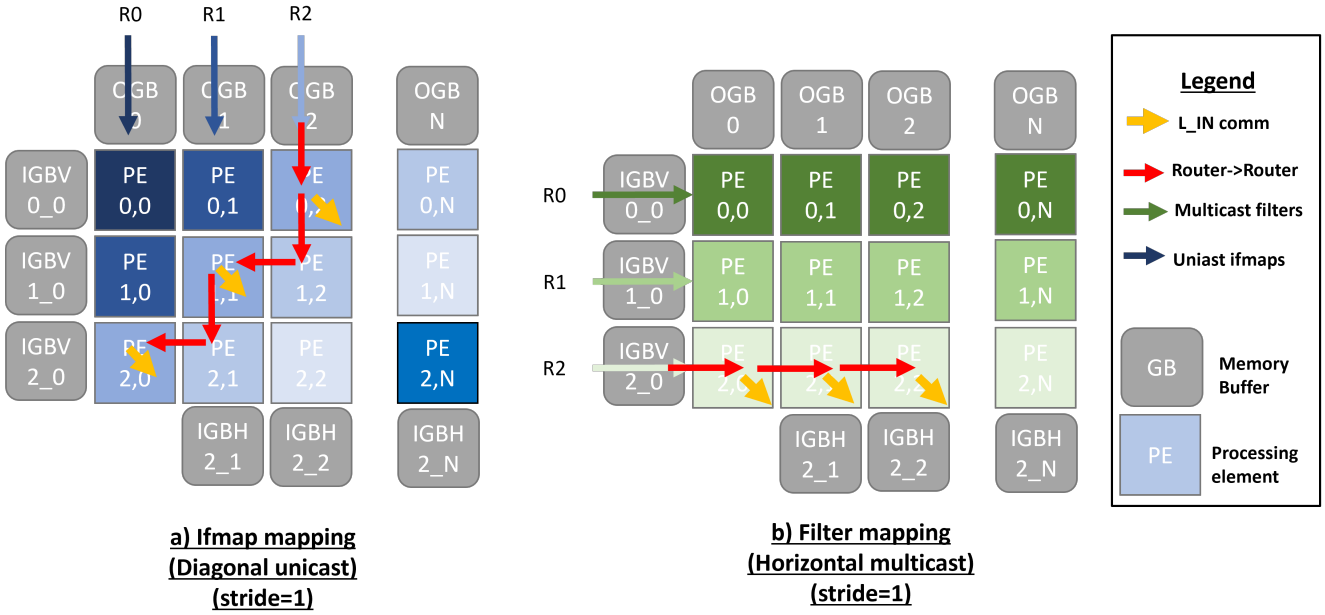


Figure 4.7: ADP for depthwise convolution hidden layer: (a) Diagonal unicast for ifmap mapping from top-right to bottom left, (b) Horizontal multicast for filter distribution. These patterns scale automatically with synthesized grid dimensions.

Standard convolutional layers follow similar patterns: stride-1 padding-0 operations use diagonal multicast (either northeast for first layers or southwest for hidden layers), while stride-2 employs vertical multi-unicast transfers. Pointwise convolutions optimize for channel wise operations using vertical multi-unicast to maximize parallelism across all channels simultaneously. Fully connected layers reuse the pointwise convolution mapping strategy, processing flattened input vectors as single pixel feature maps (1×1 spatial dimensions). However, due to the absence of spatial reuse, FC layers process inputs in sequential batches: each batch is distributed across the PE array, local multiply-accumulate operations execute in parallel, then results are reduced through a two phase accumulation. This batch-iterative approach reuses the PW hardware mapping while accommodating the fully connected topology.

Across these layer cases, ADP can be seen as a parameterized execution template that reconfigures NoC transfers and buffer access patterns to match the reuse opportunities induced by stride and padding.

This reconfigurability allows PANDA to support standard, depthwise, pointwise, and fully connected layers on a single PE array without structural modifications. The trade-off is a substantially enlarged design space that couples DNN choices (layer shapes, strides), mapping decisions (dataflow variants), and architectural parameters (grid dimensions, buffer sizes). Exploring this space via exhaustive RTL simulation would be prohibitively expensive. To address this, we combine two complementary

evaluation components: an analytical mapping model that estimates memory traffic and compute demand analytically, and an ML-based PPA estimator trained from synthesis and RTL simulation results. Together, they enable fast design space exploration.

4.3 Mapping Model of CNN Layers

Efficient design space exploration requires fast and trustworthy feedback on how a given CNN layer and schedule will exercise the hardware: not only how many operations are performed, but also when data must move, where reuse is created or lost, and which resources become the bottleneck. Our mapping model is built to provide this feedback with minimal assumptions, starting from the loop nest semantics that define correctness and using them to reason about tensor shapes and reference compute counts. Rather than representing execution as a flat set of loops, the model mirrors the hardware hierarchy and exposes where tiling boundaries and spatial distribution force loads, reloads, and reductions. This is especially critical for input feature maps, whose true buffer footprint is governed by sliding window halos under stride and kernel support. By instrumenting the resulting execution with explicit counters and consistency checks, the model produces explainable metrics (global and per PE) that can be validated systematically and reused downstream for performance, and power estimation.

4.3.1 Loop Nest Formulation and Hierarchical Decomposition

Convolution operations can be expressed as a six-dimensional nested loop structure (as shown in Figure 3.3). The loop indices (o, i, x, y, kx, ky) iterate over the corresponding extents (O, I, X, Y, Kx, Ky), which represent, respectively, output channels (O), input channels (I), output spatial coordinates (X, Y), and kernel spatial coordinates (Kx, Ky):

$$\mathrm{OFMAP}(o,x,y) = \sum_{i=0}^{I-1} \sum_{kx=0}^{Kx-1} \sum_{ky=0}^{Ky-1} \mathrm{IFMAP}\big(i,\ x{\cdot}\mathrm{Stride}_x{+}kx,\ y{\cdot}\mathrm{Stride}_y{+}ky\big){\times}\mathrm{KERNEL}(o,i,kx,ky){+}B(o), \tag{4.1}$$

where IFMAP describes the input feature maps, KERNEL the kernel weights, OFMAP the output feature maps, and $B(o)$ the bias associated with output channel o (as shown in subsection 2.3.2). This loop nest serves as the semantic reference: any sequential or parallel dataflow must preserve the data dependencies encoded by this structure.

To parameterize this reference loop nest and enable consistency checks in the mapping model,

we first derive the output spatial extents from the layer configuration. We denote the input feature map spatial dimensions by $W \times H$ (width $\times$ height), the kernel size by $Kx \times Ky$, and the strides by $(\mathrm{Stride}_x, \mathrm{Stride}_y)$. The output dimensions are then computed as:

$$X = \left\lfloor \frac{W - Kx}{\mathrm{Stride}_x} \right\rfloor + 1, \qquad Y = \left\lfloor \frac{H - Ky}{\mathrm{Stride}_y} \right\rfloor + 1. \tag{4.2}$$

These expressions are used throughout the mapping model to derive buffer footprints and to support consistency checks. In particular, they provide a closed-form estimate of the total compute workload:

$$\mathrm{MAC}_{\mathrm{theoretical}} = O \times I \times X \times Y \times Kx \times Ky. \tag{4.3}$$

This quantity serves as a validation reference by comparison with the MAC count accumulated during the simulated traversal.

While the six loop formulation is sufficient to define correctness, practical accelerators cannot keep full tensors on-chip. Consequently, the reference loop nest must be decomposed to match the memory and parallel hierarchy of the target architecture (as shown in Figure 4.8).

```
# DRAM loops
for o_buf in range(0, O, DRAM_o) :                  # o_end = min(o_buf + DRAM_o, O)
  for i_buf in range(0, I, DRAM_i):                 # i_end = min(i_buf + DRAM_i, I)
    for x_buf in range(0, X, DRAM_x):               # x_end = min(x_buf + DRAM_x, X)
      for y_buf in range(0, Y, DRAM_y):             # y_end = min(y_buf + DRAM_y, Y)
        for kx_buf in range(0, Kx, DRAM_kx):        # kx_end = min(kx_buf + DRAM_kx, Kx)
          for ky_buf in range(0, Ky, DRAM_ky):      # ky_end = min(ky_buf + DRAM_ky, Ky)

# Parallelism loops
          A) for_parallel o in range(o_buf, o_end):  # Parallel Output channels
          B) for_parallel i in range(i_buf, i_end):  # Parallel Input channels
# Buffer loops
            for x in range(x_buf, x_end):           # Buffer position X
              for y in range(y_buf, y_end):         # Buffer position Y
                for kx in range(kx_buf, kx_end):    # Buffer Kernel X
                  for ky in range(ky_buf, ky_end):  # Buffer Kernel Y
                    ofmap[o, x, y] += ifmap[i, x + kx, y + ky] * kernel[o, i, kx, ky]
```

Figure 4.8: Hierarchical loop structure with three levels: DRAM loops for temporal distribution, parallelism loops for spatial distribution, and buffer loops for on-chip reuse.

For an architecture with DRAM, 2D parallelism, and on-chip buffer memory, we model three loop-levels. If additional memory-levels are present, we extend the hierarchy by inserting the corresponding

loop-levels. In the example Figure 4.8, at the outer-level, loops iterate over tiles transferred from DRAM to integrated buffers which sets the granularity of off-chip traffic (#DRAM loops). At the intermediate-level, loops distribute these tiles across processing elements, defining spatial work assignment and load balance (#Parallelism loops). At the inner-level, loops control buffer accesses within each processing element, where temporal reuse is exploited before a reload from DRAM becomes necessary (#Buffer loops). Together, these levels constrain the execution schedule through buffer capacity, available parallelism, and reuse opportunities induced by loop ordering.

However, encoding this hierarchy directly as explicit nested loops becomes difficult to maintain as the number of levels grows. Counter management for tile offsets, loop indices, and PE assignments becomes tedious, and event detection (e.g., reload boundaries or MAC counts) requires coordinating state across multiple levels. As a result, extending the model to new memory hierarchies or parallelism organizations does not scale: it tends to require invasive changes and becomes error prone.

We address this by unifying all levels into a single recursive execution engine (as shown in Figure 4.9). Each hierarchical-level follows the same pattern: iterate over the tiles at the current level, then recursively descend to the next level.

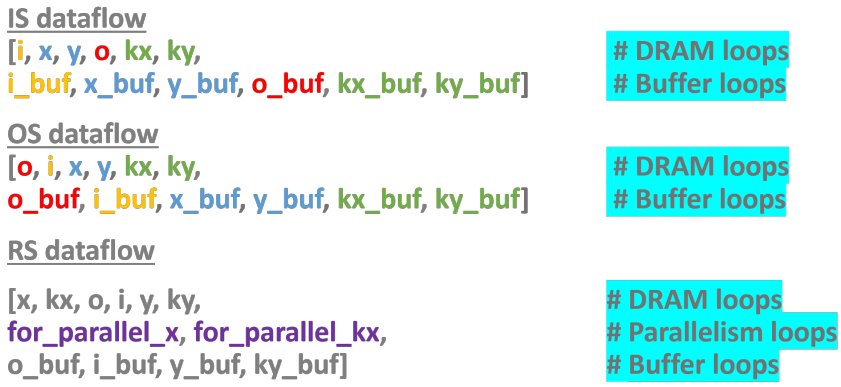


Figure 4.9: Unified recursive engine for dataflows and reuse (IS, OS, RS, ...)

We encode a mapping as an ordered vector of loop indices, read from left (outermost) to right (innermost), and grouped by hierarchy: indices without suffix correspond to the upper-level tiling loops (e.g., DRAM-level traversal), while indices with the _buf suffix correspond to the finer-grain loop nest executed within the on-chip buffer tile. In the RS example, we additionally include explicit for_parallel_* entries to denote loops spatially distributed across processing elements; notably, the IS and OS examples are purely temporal and therefore contain no parallel loops.

Different dataflows are then captured simply by reordering this vector while preserving the same semantic loop set (o, i, x, y, kx, ky) from the reference convolution loop nest. Concretely, the IS

ordering prioritizes the input channel loop i at the outer-level, promoting input activation reuse, whereas OS starts with the output channel loop o, keeping partial sums local for longer. RS interleaves spatial and kernel dimensions and explicitly spatializes x and kx across PEs (purple), which is why those dimensions do not reappear as _buf loops in the RS vector. Context flows naturally through the recursion stack, allowing events to be detected at the appropriate hierarchical-level without cross checking several counters.

This approach preserves the underlying traversal complexity: it enumerates the same loop iterations as a traditional nested loop formulation. At the same time, implementation becomes simpler and more flexible. Adding a new memory-level requires only one additional recursion-level, and different parallelism organizations are expressed by changing parallelism factors and loop ordering rather than rewriting the core logic. The same engine therefore supports tiling, PE distribution parallelism, and on-chip reuse within a compact and extensible tool (as shown in Figure 4.10).

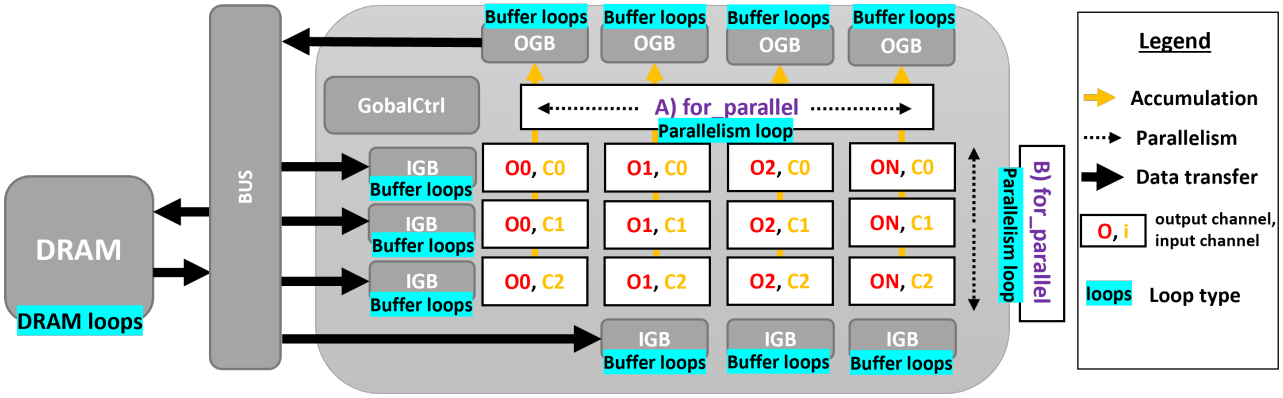


Figure 4.10: Hardware architecture illustrating hierarchical dataflow from DRAM through global buffers to the processing element array for OS example.

4.3.2 Tiling Strategy and Memory Constraints

The recursive execution engine unifies how we traverse the hierarchy, but it does not determine how much work is executed per recursion-level. This is the role of the tiling strategy. Because on-chip storage has finite capacity, we split the reference convolution loop nest into tiles so that, at each on-chip descent (i.e., at the level of the _buf loops), the logical working sets of the three tensors KERNEL, IFMAP, and OFMAP can be staged in the physical on-chip memories (IGBs and OGBs). In our architecture model we use a fixed tensor to buffer binding in the baseline where KERNEL and IFMAP are staged in the IGBs, while OFMAP are staged in the OGBs. To keep the policy deterministic and simple, we assume a fixed allocation (e.g., static partitioning or separate banks) that yields per tensor capacity budgets expressed as maximum numbers of elements: cap_K for

KERNEL tiles, cap_I for IFMAP tiles (both derived from the IGB allocation), and cap_O for OFMAP tiles (derived from the OGB allocation). A mapping is valid only if the chosen tile sizes satisfy the corresponding capacity constraints. The strategy described here is intentionally basic: it provides a simple and deterministic baseline, it can be improved substantially, and it is not claimed to be optimal; we nevertheless adopt it as our default reference policy.

A dataflow induces an ordered list of loop indices. For each tensor, we restrict this global order to the indices that actually address it, yielding a tensor specific axis order that describes how reuse is exposed while the tensor resides on-chip in its assigned physical buffer. This order is then used by the tiling heuristic: intuitively, it determines which axes are kept more inner to maximize immediate reuse. We adopt a flush and reload baseline for buffer management: when the iteration space advances to a new tile, the on-chip contents are assumed to be fully replaced. Consequently, reuse is only credited when it occurs within the lifetime of a single on-chip tile, and we do not model incremental retention mechanisms (e.g., line buffers or sliding window buffering) that could exploit overlap across consecutive tiles. This choice keeps mapping and capacity checks simple and deterministic, at the cost of potentially underestimating reuse.

This assumption naturally separates tensors that admit disjoint tilings in their native index spaces from inputs, whose footprint is induced by the convolution window and therefore typically overlaps across neighboring output tiles. We accordingly classify $\mathrm{KERNEL}(o, i, kx, ky)$ and $\mathrm{OFMAP}(o, x, y)$ as the disjoint case, and handle IFMAP separately with a window/halo aware footprint model.

For the disjoint tensors (KERNEL and OFMAP), we use a simple greedy splitter that allocates the available per tensor capacity budget to axes from innermost to outermost. Let $(a_1, \ldots, a_m)$ be the tensor specific axis order from outermost to innermost, with extents $\mathrm{extent}(a_j)$. We compute tile sizes by scanning right to left, maintaining the current tile volume V (initialized as $V = 1$). For each axis a_j, we set

$$t_{a_j} \;=\; \min\Big(\mathrm{extent}(a_j),\; \Big\lfloor \frac{\mathrm{cap}}{V} \Big\rfloor\Big), \qquad V \leftarrow V \cdot t_{a_j}.$$

By construction, the final volume satisfies $V \leq \mathrm{cap}$, hence the tile fits in the tensor's allocated on-chip budget. In this work, the axis orders derived from the schedule and the greedy policy above are considered as fixed defaults to provide a clear, deterministic baseline.

The importance of keeping the right axes "inner" can be seen on a simple illustrative example for kernel weights. Consider the loop nest restricted to weights:

$$\texttt{for } o\texttt{: for } i\texttt{: for } kx\texttt{: for } ky\texttt{:}\ \text{access}\ \mathrm{KERNEL}(o, i, kx, ky).$$

Assume $O = 8$, $I = 3$, $Kx = 3$, $Ky = 3$, so the full tensor contains $O \cdot I \cdot Kx \cdot Ky = 216$ elements.

Let the weight tile budget be $\mathrm{cap}_K = 9$. Any weight tile $(o_{\text{tile}}, i_{\text{tile}}, kx_{\text{tile}}, ky_{\text{tile}})$ must satisfy

$$o_{\text{tile}}\, i_{\text{tile}}\, kx_{\text{tile}}\, ky_{\text{tile}} \le 9.$$

If we choose $(o_{\text{tile}}, i_{\text{tile}}, kx_{\text{tile}}, ky_{\text{tile}}) = (1, 1, 3, 3)$, the on-chip budget holds the full $Kx \times Ky$ kernel for a fixed (o, i). As the inner loops traverse kx and ky, each weight is loaded once and then reused from on-chip storage across consecutive inner iterations. In contrast, if we choose $(o_{\text{tile}}, i_{\text{tile}}, kx_{\text{tile}}, ky_{\text{tile}}) = (1, 3, 3, 1)$, the on-chip budget holds all (i, kx) combinations but only one position along ky. Covering the full $Ky = 3$ then requires three ky tiles; since ky is innermost, these tiles are consumed one after another, forcing reloads each time we advance to the next ky tile. Intuitively, capacity is spent on axes that are not aligned with the tightest inner reuse axis under this loop order, which increases reload frequency.

Inputs require special care because producing an output tile of size $(x_{\text{tile}}, y_{\text{tile}})$ touches a larger IFMAP region due to the sliding window. Ignoring borders for the moment, the required input patch dimensions are

$$X_{\text{win}}(x_{\text{tile}}) = \mathrm{Stride}_x\,(x_{\text{tile}} - 1) + Kx, \qquad Y_{\text{win}}(y_{\text{tile}}) = \mathrm{Stride}_y\,(y_{\text{tile}} - 1) + Ky.$$

If the input tile spans i_{tile} channels and serves an output tile $(x_{\text{tile}}, y_{\text{tile}})$, the physical IFMAP footprint (in elements) is

$$\mathrm{cost}(i_{\text{tile}}, x_{\text{tile}}, y_{\text{tile}}) = i_{\text{tile}} \cdot X_{\text{win}}(x_{\text{tile}}) \cdot Y_{\text{win}}(y_{\text{tile}}),$$

and the IFMAP on-chip budget constraint is

$$\mathrm{cost}(i_{\text{tile}}, x_{\text{tile}}, y_{\text{tile}}) \le \mathrm{cap}_I.$$

This cost is non decreasing in each argument, which makes greedy enlargement well defined. The halo aware splitter starts from $(i_{\text{tile}}, x_{\text{tile}}, y_{\text{tile}}) = (1, 1, 1)$ and, following an axis order for the IFMAP tensor, increases one axis at a time to the largest value that still fits in capacity. For a chosen axis, the largest feasible tile size can be found by binary search over its full extent because the cost is monotone.

The guiding objective in this baseline is alignment with the OFMAP tiling: an IFMAP tile should ideally cover the output tile it feeds, without paying excessive extra halo. This strategy strongly adapts to the CEA architecture implemented dataflows.

Let $(x_{\text{tile}}^{(O)}, y_{\text{tile}}^{(O)})$ be the spatial tile sizes chosen for the OFMAP tensor (and therefore for its OGB-resident working set), and let a candidate input order produce $(i_{\text{tile}}^{(I)}, x_{\text{tile}}^{(I)}, y_{\text{tile}}^{(I)})$. We score candidates

deterministically using three priorities: first, maximize spatial match through $\min(x_{\text{tile}}^{(I)}, x_{\text{tile}}^{(O)})$ and $\min(y_{\text{tile}}^{(I)}, y_{\text{tile}}^{(O)})$; second, minimize overshoot beyond the OFMAP tile via $(x_{\text{tile}}^{(I)} - x_{\text{tile}}^{(O)})_+$ and $(y_{\text{tile}}^{(I)} - y_{\text{tile}}^{(O)})_+$; third, among comparable candidates, prefer larger logical tiles (fewer tiles, fewer DRAM transactions) while keeping the physical IFMAP footprint

$$i_{\text{tile}}^{(I)} \cdot X_{\text{win}}(x_{\text{tile}}^{(I)}) \cdot Y_{\text{win}}(y_{\text{tile}}^{(I)})$$

as small as possible. These priorities are encoded as a lexicographically compared score tuple, making the choice fully deterministic and cheap at mapping time.

The IFMAP window model above characterizes the input region required by a given OFMAP tile and is exact for tiles that lie fully in the interior of the feature map. In practice, tiles close to the spatial boundaries may require input coordinates that fall outside the original IFMAP domain. Padding and border handling must therefore be made explicit, both to preserve correctness and to reason consistently about IFMAP footprints near the edges.

In our implementation, padding is handled by materializing a padded input tensor. Let the original IFMAP have spatial size $W \times H$. We build an extended IFMAP $\mathrm{IFMAP}_{\mathrm{pad}}$ of spatial size $(W + 2\mathrm{Pad}_x) \times (H + 2\mathrm{Pad}_y)$, where the central region contains the original input and the surrounding border is filled with zeros. The convolution is then evaluated by indexing $\mathrm{IFMAP}_{\mathrm{pad}}$ without any out-of-range accesses:

$$\begin{aligned} \mathrm{OFMAP}(o, x, y) \;=\; & \sum_{i=0}^{I-1} \sum_{kx=0}^{Kx-1} \sum_{ky=0}^{Ky-1} \mathrm{IFMAP}_{\mathrm{pad}}\Big(i,\; x \cdot \mathrm{Stride}_x + kx,\; y \cdot \mathrm{Stride}_y + ky\Big) \\ & \times \mathrm{KERNEL}(o, i, kx, ky) \;+\; B(o). \end{aligned} \tag{4.4}$$

With this convention, the IFMAP window sizes $X_{\text{win}}(\cdot)$ and $Y_{\text{win}}(\cdot)$ apply uniformly to all tiles (including at the borders), since every accessed coordinate is valid in $\mathrm{IFMAP}_{\mathrm{pad}}$. As a result, the same footprint and capacity checks can be applied without introducing special-case boundary logic.

These modeling choices define a complete and deterministic baseline mapping policy. The next subsection describes how this policy is executed within the mapping model pipeline and how we extract metrics and validate the recursive traversal.

4.3.3 Event Counters, Diagnostics, and Validation

The quality of the mapping model depends on its ability to faithfully represent hardware execution and deterministically account for memory traffic, parallelism effects, and computational work. To connect the loop model to hardware-relevant metrics, we implement a unified pipeline that executes the hierarchical recursive loop model and collects event counters during execution (as shown in Figure 4.11).

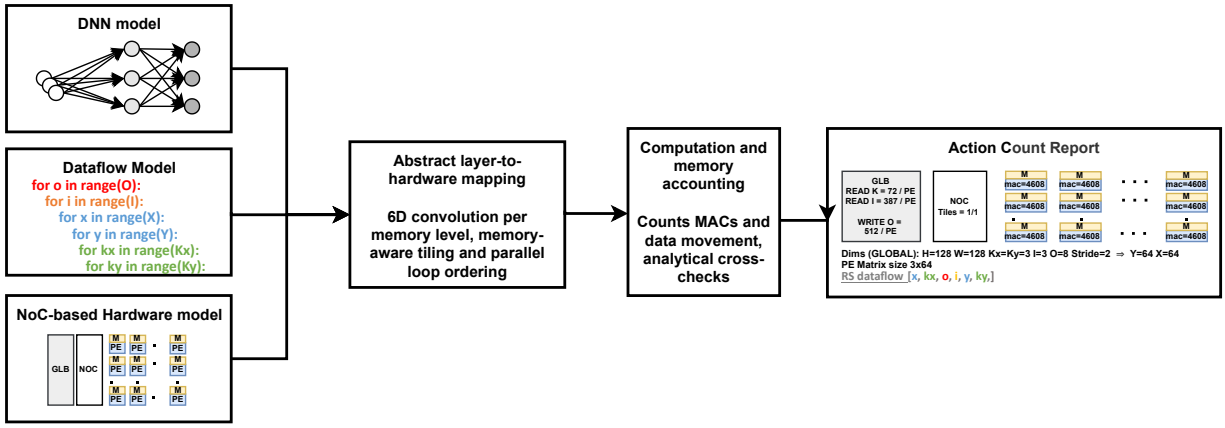


Figure 4.11: Mapping model pipeline from simulation to event counters, diagnostics, and validation.

The mapping model takes as inputs CNN layer dimensions, memory capacities, loop ordering, and parallelism factors. The engine runs the recursive loop execution framework, which unfolds the nested loops while maintaining an execution context that tracks the current tile, PE assignments, and buffer states. Because tiling is computed upfront, the simulator advances at tile granularity rather than iterating element by element, aggregating MAC and memory access counters per tile. This avoids per element overhead and substantially accelerates evaluation. Events are counted at the time they occur, enabling time-ordered traces without reconstruction, and memory events are detected deterministically by tracking buffer state changes at the appropriate level of the hierarchy.

The collected event counters can be combined with an architecture specific PPA cost model, parameterized by per event latency and power costs, to derive end-to-end latency and power estimates for a given workload and mapping. This requires the mapping model to expose events at the same granularity and with the same definitions as the cost primitives assumed by the PPA model, so that each counter maps unambiguously to a unit cost.

In the remainder of this section, we first describe how memory events are detected and classified, then present the validity checks used to verify counter consistency, and finally describe the per PE metrics and the diagnostic outputs produced by the pipeline.

4.3.3.1 Memory Event Detection

The mapping model detects and counts DRAM-related events for each buffer, including reads to fill on-chip storage and writes for produced outputs. These events vary across the buffer hierarchy because tiling and reuse are buffer dependent. In particular, reload events depend on which loop axes are only partially tiled for a given buffer and on the buffer axis-ordering policy.

For weights and input feature maps, a reload occurs when execution advances along the outermost axis (according to the buffer policy order) for which the buffer is only partially tiled. Intuitively, once the execution steps to a tile index that exceeds what is resident under the buffer's partial tiling axis, the current contents are no longer valid and must be fetched again. More precisely, let $\mathrm{part_axes}(B)$ denote the set of axes of buffer B whose chunk size is smaller than the full extent. Let a_B be the first axis in $\mathrm{part_axes}(B)$ when axes are ordered by the buffer policy from outer to inner loops. A reload is triggered when the tile index along a_B changes between two consecutive tiles, in which case the current buffer contents become invalid and must be fetched again from DRAM.

We distinguish the first load, which occurs on the first tile, from subsequent reloads. This distinction is architecturally significant because first loads capture the baseline traffic required to initialize computation, while reloads capture recurring traffic induced by the tiling strategy. The volume of each load event is the product of tile sizes along the buffer dimensions, with special handling for input feature maps to account for spatial overlap via the W and H window calculation (as shown in subsection 4.3.2).

For outputs, the transaction model differs. We assume partial sums remain in the on-chip output buffer during inner loop execution. Output tiles are produced and written back when an output tile is completed, and the model does not assume DRAM reads for outputs during inner loop execution. As recursion proceeds, all memory events are accumulated in the execution context, which maintains current counts of first loads, reloads, writes, and total element transfers. Having defined the memory event taxonomy, we next validate that the executed loop schedule and the resulting counters are self consistent.

4.3.3.2 Validation and Consistency Checks

The primary validation check compares the total MAC count accumulated during mapping model execution against the analytical expectation derived from the layer parameters:

$$\mathrm{MAC}_{\mathrm{theoretical}} = O \times I \times X \times Y \times Kx \times Ky,$$

where (X, Y) are computed from the input dimensions as

$$X = \left\lfloor \frac{W - Kx}{\text{Stride}_x} \right\rfloor + 1, \qquad Y = \left\lfloor \frac{H - Ky}{\text{Stride}_y} \right\rfloor + 1.$$

After every run, a consistency check compares the total simulated MACs against $\text{MAC}_{\text{theoretical}}$. Any mismatch flags an inconsistency in the mapping model implementation and, when validation is enabled, raises an exception.

Beyond the total MAC count, validation also checks that the simulator consistently separates accumulator initializations from accumulate add updates. For each output element $\text{OFMAP}(o, x, y)$, the inner product spans $I \times Kx \times Ky$ multiplications. Under the standard accumulation convention, exactly one multiplication initializes the accumulator, and the remaining $I \times Kx \times Ky - 1$ multiplications contribute accumulate add updates.

The mapping model detects initializations by checking whether the current compute step corresponds to the first visit of the (i, kx, ky) space for the current (o, x, y) tile (equivalently, the first tile of the input channel and kernel loops). Under this convention, the expected number of initialization operations is

$$\text{MAC}_{\text{inits}} = O \times X \times Y,$$

and the expected number of accumulate add operations is

$$\text{ACC}_{\text{adds}} = O \times X \times Y \times (I \times Kx \times Ky - 1) = \text{MAC}_{\text{total}} - \text{MAC}_{\text{inits}}.$$

After establishing global consistency, we refine the view by attributing work and traffic to individual processing elements.

4.3.3.3 Per PE Metrics and Export

In addition to memory events, the recursive execution engine collects compute and traffic metrics at multiple granularities. Global summary counters, such as total MACs and total traffic, are accumulated in the counters dictionary. Per processing element metrics are accumulated by associating each leaf compute step with a PE coordinate derived from the parallelism loops and by recording the corresponding tile parameters. For example, under 2D parallelism over output rows and kernel rows, a PE may have coordinate $(Y = 2, Ky = 1)$, indicating responsibility for output row stripe 2 and kernel row stripe 1.

For each PE, we record the tile dimensions $O_{\text{tile}}, I_{\text{tile}}, X_{\text{tile}}, Y_{\text{tile}}, Kx_{\text{tile}}, Ky_{\text{tile}}$ and buffer access volumes for weights, outputs, and inputs, including both logical and physical input elements when

accounting for IFMAP spatial halo. The IFMAP corresponding W and H values are stored because they determine the effective input footprint required to produce each output tile. These metrics are accumulated across all leaf compute steps involving the same PE.

After execution completes, the aggregated per PE data is exported to a pandas DataFrame [112], with one row per unique PE coordinate and one column per metric. This representation supports downstream analysis and visualization, including load balancing studies and heatmaps of memory traffic to identify PEs with disproportionate compute or access patterns. With the per PE and global counters available, the pipeline finally emits a set of diagnostic artifacts designed for both debugging and quantitative analysis.

4.3.3.4 Diagnostics and Outputs

To support both debugging and quantitative analysis, the mapping model emits several complementary artifacts at the end of execution. First, an execution trace can be printed to standard output, where indentation follows recursion depth and each step reports the active loop, tile size and index, and the corresponding buffer state transitions, including reload points. When parallelism is enabled, the trace additionally reports the PE coordinates performing each compute step and their local work assignments, which is useful for validating scheduling decisions and pinpointing inconsistencies.

Second, the per PE metrics can be serialized to CSV. This tabular view contains PE coordinates, tile parameters, MAC counts, buffer access volumes, and window sizes, enabling downstream post processing in scripts or spreadsheets. In addition, the pipeline produces compact summary reports in JSON format that capture aggregate statistics such as total ingress and egress traffic, per buffer first load and reload counts, auto derived tile sizes, and the outcome of enabled consistency checks.

Finally, the pipeline can generate lightweight spatial diagnostics to provide an immediate view of imbalance. In particular, ASCII heatmaps summarize MAC activity and memory traffic across the PE array for weights, input feature maps, and output feature maps, making it easy to identify skewed work distribution or concentrated access patterns that suggest opportunities for improved reuse or parallelism rebalancing. Because all output feature maps are emitted in standard, tool friendly formats, they integrate naturally with external visualization tools and automated analysis pipelines.

Beyond diagnostics, the same event counters also provide the interface to cost modeling. Counts of primitive actions such as loads, writes, MAC operations, and synchronization events can be combined with per action latency and power costs in the PPA framework to compute end-to-end performance and power estimates, and to support automated design space exploration.

Although the event counters precisely describe the executed workload and expose the interface required for cost evaluation, translating these events into physical latency, power, and area sees the

model depend on technology and implementation specific effects. The next section introduces a learning-based PPA estimation methodology that leverages RTL simulation and synthesis data to capture these effects and to enable fast PPA prediction for new mappings and configurations.

4.4 Learning-Based PPA Estimation Methodology

Efficient evaluation of the design space covering DNN models, dataflow architectures, and hardware parameters requires accurate yet computationally feasible modeling of performance, power, and area characteristics. The mapping model proposed in the previous section determines the number and nature of high-level operations to be executed. However, these operations must be validated and characterized through measurable physical metrics.

Obtaining such metrics through exhaustive simulation and synthesis for every candidate design point is prohibitively expensive. The core difficulty lies in the non linear interactions between workload parameters (layer dimensions, stride values, etc.), architectural choices (PE array size, buffer capacity, etc.), and physical implementation effects (routing overhead). A layer configuration that saturates memory bandwidth on one architecture may become compute bound on another, and synthesis optimizations can shift resource utilization unpredictably. These interactions make analytical modeling unreliable without empirical calibration across the design space.

To address this challenge, we propose a learning-based PPA estimation methodology (shown in Figure 4.12) that decomposes the problem into tractable subproblems while preserving the ability to capture nonlinear interactions between workload, architecture, and implementation effects. Instead of exhaustively characterizing all configurations, we construct training data through synthetic layer generation, trading completeness for systematic coverage of representative operating points. This enables controlled analysis of how workload parameters affect hardware behavior under a feasible characterization budget.

The measurement strategy is metric dependent. Performance (latency) is characterized using compact, state-level observables (performance counters and FSM derived statistics) that expose execution bottlenecks at low instrumentation cost. Area is obtained from synthesis utilization reports and is independent of runtime traces. Power is investigated using activity-based reporting derived from switching traces. However, this requires full (or near-full) signal activity and can be further constrained by trace to netlist reconciliation limiting the number of configurations that can be characterized. Overall, this metric-dependent strategy balances fidelity with data collection cost across PPA.

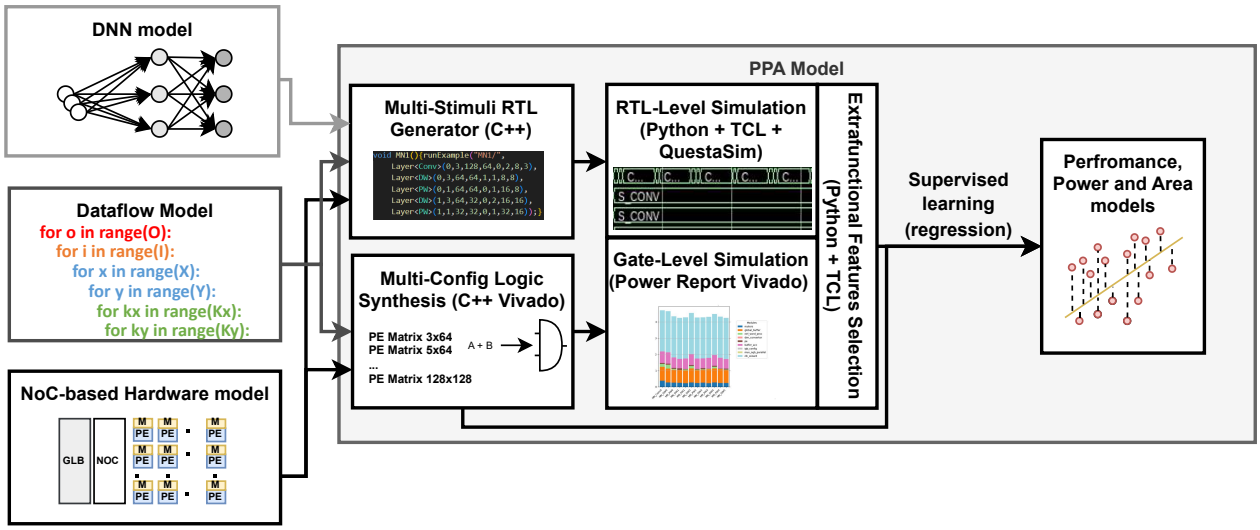


Figure 4.12: Overview of the PPA estimation methodology combining simulation, synthesis, and machine learning

Finally, because different layer types and stride patterns exhibit distinct execution modes on the target NoC-based dataflow architecture, we train separate models per execution context rather than forcing a single global predictor to fit heterogeneous behaviors.

The remainder of this section develops these ideas in two stages. We begin by presenting the methodological principles: how the PPA problem is formulated, what training data is constructed, and how learning is structured to exploit execution phase decomposition. We then describe the concrete implementation that instantiates these principles through simulation workflows, synthesis automation, and supervised learning pipelines.

4.4.1 Proposed Methodology

PPA estimation for DNN accelerators requires predicting how workload parameters, architectural choices, and physical implementation effects interact. This methodology decomposes the problem into context-specific ML-based regression tasks, each targeting a distinct execution modes defined by layer type, stride pattern, and dataflow behavior.

This decomposition is necessary because different layer types exhibit fundamentally different execution characteristics on the target NoC-based dataflow architecture. Convolution layers with stride 2 generate dense, bursty memory traffic during input loading phases, creating bandwidth bottlenecks. Depthwise convolution layers with stride 1 exhibit more uniform memory access patterns but reduce arithmetic intensity, shifting the bottleneck from compute to memory latency. Pointwise convolution layers shift between compute bound and memory bound modes depending on channel

count. Within each layer type, execution unfolds through distinct phases (loading feature maps, loading weights, computing partial sums, writing results), each with different resource demands. A single global model that pools these heterogeneous behaviors would either overfit to the dominant execution mode or underfit the minority cases. Training separate models for each execution mode enables specialization on narrower distributions while preserving interpretability.

Implementing this decomposition rests on three methodological principles. First, training data must cover representative execution modes without exhaustive enumeration. Synthetic data generation allows systematic parameter variation while keeping the characterization effort manageable. Second, the level of observability must match the metric being predicted. Some metrics can be estimated from compact high-level statistics, while others require detailed signal traces. This approach balances modeling accuracy with data collection cost. Third, the target metrics are decomposed to reflect how hardware execution actually behaves. By isolating independent sources of variation, ML-based regression models become easier to train and can, where possible, be combined with analytical estimates.

These principles structure the methodology into four components. Data construction defines how synthetic layers are generated to cover representative operating modes while respecting hardware constraints. Representation and observability determine which aspects of hardware execution are instrumented and how they are transformed into modeling features. The learning formulation explains how the estimation problem is decomposed into context specific ML-based regression tasks and how target metrics are separated to simplify modeling. The machine learning pipeline describes how these elements are used within a supervised learning workflow to produce predictive models. The following subsections detail each component, first presenting the methodological foundation and then the corresponding implementation choices.

4.4.1.1 Data Construction Methodology

Exhaustively characterizing all feasible hardware-mapping configurations is computationally impractical. Throughout this study, we therefore use a 3×64 PE reference instance. It is composed by a PE array organized as 3 rows by 64 columns (192 PEs total). Even for this single hardware instance, the feasible design space already contains nearly 4 million valid mapping configurations. Architectural constraints restrict the admissible spatial unrolling factors (bounded by the PE-array dimension, in particular the width of 64) and enforce dataflow-dependent execution modes that differ across layer types (Conv, DW, PW). For dataset generation, we further scope the exploration to channel depths up to 256 and select representative combinations of stride, padding, and kernel dimensions that cover the operational boundaries relevant to efficient CNN families such as MobileNet.

Within our toolchain, RTL simulation requires 2 to 40 minutes per layer depending on complexity and instrumentation depth, while our FPGA synthesis typically requires 20 minutes to 6 hours per configuration. At these rates, exhaustive enumeration of the full 4 million configurations would require on the order of 15 to 300 CPU-years for simulation alone, excluding synthesis and repeated measurements across metrics.

We address this limitation through synthetic layer generation with stratified sampling across execution modes. This approach differs from using reference networks (e.g., ResNet, MobileNet) not in the nature of the configurations, but in how they are sampled. Reference networks provide realistic layers, yet they form dependent sequences: consecutive layers share activations and memory allocation decisions depend on full network structure, which prevents processing layers as independent training samples for supervised ML-based. Moreover, coverage is uncontrolled: layer parameters arise from high-level architecture choices rather than systematic variation, making it difficult to guarantee that boundary conditions (resolution limits, channel extremes, stride transitions) are sufficiently represented.

Synthetic generation resolves both issues (Figure 4.13). Each generated layer is independent (no cross-layer dependencies), and sampling is controlled to ensure systematic coverage. Concretely, all synthetic layers respect the hardware constraints: kernels are fixed to 3×3 (Conv/DW) or 1×1 (PW), strides are restricted to $\{1, 2\}$, output spatial dimensions are bounded to 64×64, and channel counts are restricted to powers of two (8 to 256). This controlled sampling enables isolating the impact of individual parameters on performance and resource usage, which is difficult in reference networks where multiple parameters co-vary.

Stratification is performed across the three operation classes because they exhibit distinct execution modes: standard convolutions tend to be memory bound, depthwise convolutions are communication-intensive (multicast-heavy), and pointwise convolutions shift toward compute bound behavior as channels increase. This is particularly important because pointwise layers dominate the feasible configuration space, yet the ML-based models require sufficient examples from all execution modes to learn latency variations across the full behavioral spectrum. We therefore allocate samples explicitly rather than sampling uniformly.

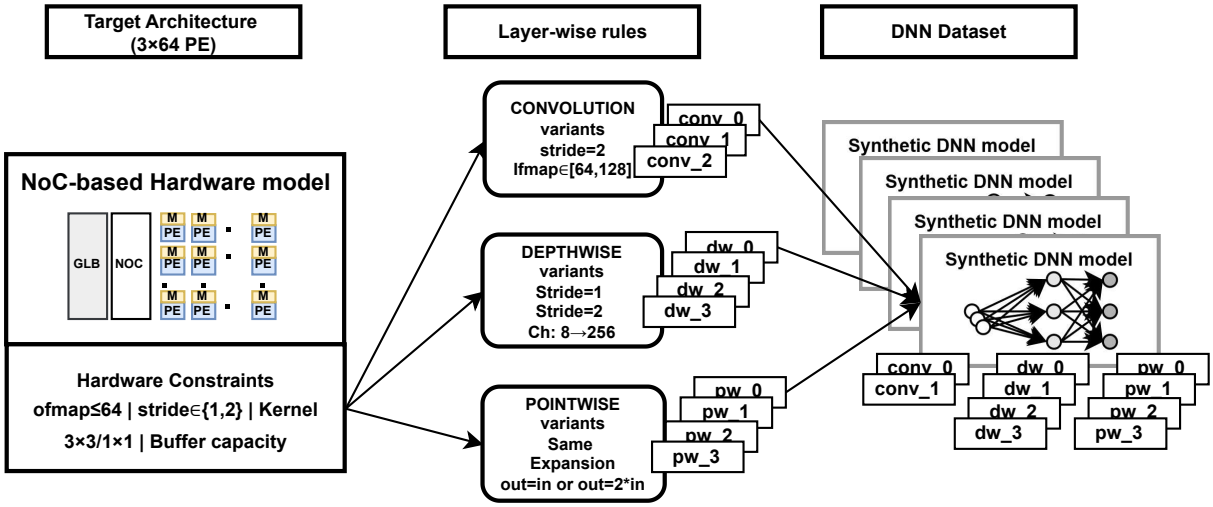


Figure 4.13: CNN Dataset Generation Workflow

The resulting synthetic training dataset contains 300 layers and represents approximately 0.008% of the feasible space while preserving execution modes coverage. It is composed of: (i) 50 standard convolutions split between stride 1 and stride 2, covering input resolutions from 8×8 to 128×128 (subject to the output bound); (ii) 150 depthwise convolutions (75 per stride value), spanning channels 8 to 256; and (iii) 100 pointwise convolutions split evenly between same channel and channel expansion configurations. Duplicate configurations are removed during generation, and all retained layers are validated against buffer-capacity and executability constraints before inclusion. For power characterization, we further restrict the workload to a convolution-only subset (augmented beyond the 300 layer set) to keep full-VCD simulation tractable.

4.4.1.2 Representation and Observability

Accurate PPA prediction requires observing hardware execution at a granularity that captures dominant bottlenecks without incurring prohibitive instrumentation cost. The appropriate observability-level depends on the target metric and the underlying execution structure, and the constraints of the environment in which the predictive models are used. Latency depends on how execution time is distributed across distinct phases (loading feature maps, loading weights, computing partial sums, writing results, etc.), each dominated by different resource constraints. Power depends on switching activity across all nets in the design, as dynamic power accumulates contributions from individual toggle events. Area depends on resource allocation decisions made during synthesis, independent of runtime execution. These differing dependencies motivate a metric dependent instrumentation strategy that balances fidelity with data collection cost.

For latency estimation, we represent hardware execution as a sequence of execution states corresponding to distinct phases in the dataflow. Each state captures a coherent execution mode with homogeneous resource utilization: loading phases, computations, and result writebacks. This phase-based representation aligns naturally with the control flow implemented in hardware controllers, which transition between states according to the mapping-induced schedule. By tracking state transitions and measuring cycle durations within each state, we obtain compact observables that expose execution structure without requiring full signal-level visibility. State visitation counts reveal how the mapping distributes computation across execution phases, while mean cycle durations per state capture microarchitectural effects such as buffer contention and NoC arbitration delays.

This representation enables a natural factorization of latency into two components: state visitation count (determined by mapping decisions through loop structure, tiling, and spatial assignment) and mean cycles per state (determined by microarchitectural timing effects). The factorization has two advantages. First, it reduces ML-based regression complexity by isolating these sources of variation into separate prediction targets. Second, it establishes a path toward convergence with the analytical mapping model: state counts follow deterministically from the mapping formulation and could eventually be derived analytically, while cycle durations depend on implementation specific effects that require empirical calibration. This convergence objective motivates the choice of observables: we instrument quantities that have analytical counterparts in the mapping model, enabling progressive replacement of learned components with analytical predictions as the mapping model matures.

Full signal-level visibility is unnecessary for latency estimation because execution bottlenecks manifest at the level of state transitions and resource saturation, not individual net toggles. Instrumenting every signal in a multi-PE NoC-based design would generate trace files orders of magnitude larger than state-based counters, with minimal additional information for latency prediction. The phase-based representation already captures the dominant timing effects and improves interpretability: per state predictions can be validated against known hardware constraints, such as minimum transfer times dictated by bandwidth limits or maximum throughput bounded by PE utilization. These sanity checks would be obscured in monolithic black box predictions that lack execution phase structure.

Power and area estimation follow different instrumentation strategies but share the same decomposition principle. Power depends on net-level switching activity that cannot be inferred from coarse execution states, requiring complete signal traces to capture toggle rates across the design hierarchy. Area depends on synthesis time and resource allocation. However, both metrics are decomposed by architectural module (PEs, buffers, controllers, interconnect, etc.) rather than by execution state. For power, module-level decomposition reveals which components dominate dynamic consumption and guides targeted clock gating or voltage scaling. For area, per module resource consumption

(LUTs, flip-flops, BRAMs, DSPs, etc. for each PE, buffer, controller, etc.) exposes which components dominate silicon footprint and guides resource allocation decisions during hardware design. This module-level structure mirrors the phase-based structure used for latency: both decompositions align predictions with hardware organization rather than modeling the accelerator as a monolithic black box.

This metric-dependent observability strategy achieves three objectives. First, it reduces instrumentation and data collection costs by tailoring observables to what each metric requires. Second, it exposes a structure that aligns with hardware organization, enabling interpretable predictions that can be validated against known constraints. Third, this approach helps connect the PPA model with the analytical mapping model. Some quantities, such as state counts or resource usage, can already be computed analytically. This allows learned components to be gradually replaced by analytical estimates, so that only implementation-specific timing and physical effects need empirical modeling. Structured observability is therefore a methodological choice that matches the modular execution of dataflow accelerators and supports a gradual transition from learned to analytical modeling.

4.4.1.3 Learning Formulation

The PPA estimation problem is formulated as a collection of supervised ML-based tasks, each targeting a specific prediction objective within a well-defined execution or architectural context. A naive global approach using a single model would pool heterogeneous execution behaviors and obscure structured dependencies between predictions and hardware organization.

We instead decompose the estimation problem into context specific ML-based tasks indexed by layer type T and context S. Layer type T distinguishes the three fundamental operation classes: convolution, depthwise convolution, and pointwise convolution. Context S captures the operational modes within each type. For latency estimation, S identifies execution states corresponding to distinct dataflow phases (loading feature maps, loading weights, computing partial sums, writing results, etc.). For power and area estimation, S identifies architectural modules (PEs, buffers, controllers, interconnect, etc.). Each combination (T, S) defines an independent regression bucket with its own training data, feature space, and prediction target. Execution behaviors within a bucket are more homogeneous than across buckets: layers of the same type executing in the same state exhibit similar resource utilization patterns, memory access characteristics, and bottleneck structures, enabling more effective learning on narrower distributions.

Latency estimation uses a dual-head ML-based model. Each bucket (T, S) learns two quantities: state visitation count c_s (how many times execution state s is entered) and mean cycles per visit μ_s (average duration each time state s executes). Total latency for state s is reconstructed as $\ell_s = c_s \times \mu_s$,

and layer latency is the sum over all states. This factorization aligns prediction structure with execution phase decomposition. State counts are determined primarily by layer geometry (resolution, channels, stride) and mapping decisions (tiling, loop ordering), while cycle durations are determined primarily by microarchitectural effects (buffer contention, NoC arbitration, etc.). By learning these quantities separately, each regression target depends on a more focused subset of input parameters, improving sample efficiency and generalization. Moreover, state counts can eventually be derived analytically from the mapping model, leaving only cycle durations to be learned from hardware measurements, supporting the longer term convergence objective.

Power and area estimation follow a simpler structure. Each bucket (T, S) learns a single target: the resource usage or power consumption attributable to architectural module S when executing layer type T. Power predictions capture dynamic power consumption per module, enabling identification of high cost components for targeted optimization (clock gating, voltage scaling, etc.). Area predictions capture resource utilization per module (LUTs, flip-flops, BRAMs, DSPs, etc. for each PE, buffer, controller, etc.), exposing silicon footprint bottlenecks and guiding resource allocation. Unlike latency, power and area do not decompose into count and cost components because they depend on aggregate switching activity or synthesis time resource packing. However, module-level decomposition achieves a similar benefit: it isolates variation across architectural components, enabling each regression task to specialize on a single module's scaling behavior rather than modeling the entire accelerator monolithically.

This formulation improves learning robustness and generalization through three mechanisms. First, it reduces task complexity by narrowing the distribution of behaviors each model must capture. A model trained exclusively on depthwise stride 1 convolution in the weight loading state faces a more constrained input space than a global model spanning all layer types and states. Second, it enables sample efficient learning: even with limited training data, each bucket achieves strong accuracy within its execution modes because relevant input features are more strongly correlated with narrower targets. Third, it preserves interpretability through structured predictions aligned with hardware organization. Per state latency predictions can be validated against bandwidth limits and arithmetic intensity bounds. Per module power and area predictions can be checked against known scaling laws (PE power scales linearly with array size, buffer area scales with capacity, etc.). These sanity checks are only possible when predictions decompose along interpretable dimensions.

The formulation defines what is learned at each level. At the architectural-level, the methodology learns how hardware parameters (PE count, buffer capacity, interconnect organization) scale PPA metrics across layer types. At the mapping-level, it learns how workload parameters (resolution, channels, stride) and mapping decisions (tiling, ordering) affect execution phase durations and resource utilization. At the microarchitectural-level, it learns implementation specific timing effects

(contention, arbitration, etc.). By structuring the problem into context specific regression tasks with factorized targets, the formulation aligns supervised learning with the modular structure of dataflow accelerators.

4.4.1.4 Machine Learning Pipeline

Supervised learning implements the proposed formulation by instantiating ML-based models for each context-specific bucket. The pipeline follows standard supervised learning practices, which include feature construction and selection, choice of model families, model training, and systematic model selection and evaluation (Figure 4.14). Within this structure, the machine learning workflow produces predictive models consistent with the decomposition principles established in the previous subsections.

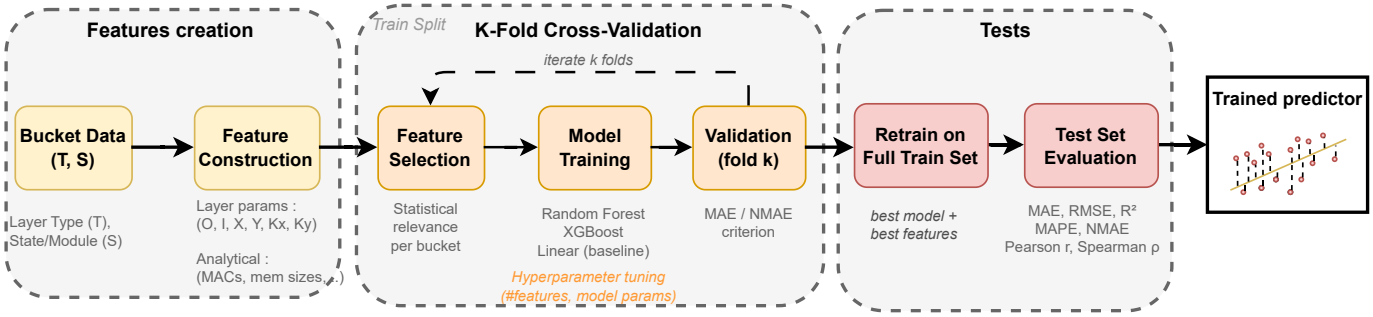


Figure 4.14: Supervised learning pipeline for PPA estimation. Feature selection operates within cross-validation folds to prevent data leakage. The best model is retrained on the full training set before final evaluation on a held-out test set.

Feature construction relies on two main sources of information. First, structural layer parameters describe the intrinsic properties of each operation, including kernel size, stride, padding, input and output feature map dimensions, and channel counts. These parameters determine the computational workload and memory footprint independently of the target architecture. Additional analytical quantities derived from these parameters provide further information. For example, the total number of multiply–accumulate operations reflects arithmetic intensity, feature map sizes indicate memory traffic, and weight volume represents parameter storage requirements. For latency estimation, state-specific features further refine predictions by highlighting the parameters most relevant to each execution phase. Weight loading phases depend mainly on filter dimensions and channel counts, whereas computation phases depend more on output resolution and arithmetic intensity. Overall, this feature construction incorporates domain knowledge of CNN computation and hardware execution, providing direct information about resource demand instead of requiring the model to infer these relationships from raw parameters alone.

Training employs k-fold cross-validation to obtain robust performance estimates [113]. Feature selection operates inside each fold: statistical relevance scores identify the most predictive parameters, and the number of retained features is tuned jointly with model-specific hyperparameters. This design prevents data leakage since the validation fold never influences feature selection. Within each fold, candidate models are trained and the configuration minimizing the validation criterion (MAE or NMAE) is retained.

The pipeline evaluates tree-based methods suited to tabular data with nonlinear interactions [114]: random forests [115], which average multiple decision trees to reduce variance, and gradient boosting [116, 117], which builds trees sequentially to correct residual errors. Linear regression serves as a baseline.

After cross-validation identifies the best model and feature subset, the selected estimator is retrained on the full training split, then evaluated once on a held-out test set. Complementary metrics (as shown in Table 4.2) quantify predictive performance: absolute error (MAE, RMSE), explained variance (R^2), scale-invariant error (MAPE, NMAE), linear correlation (Pearson r) and rank correlation (Spearman ρ).

Metric	Definition
MAE	$\mathrm{MAE} = \frac{1}{n}\sum_{i=1}^{n}\lvert\hat{y}_i - y_i\rvert$
RMSE	$\mathrm{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}$
R^2	$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$
MAPE	$\mathrm{MAPE} = \frac{100}{n_+}\sum_{i:\, y_i>0}\left\lvert\frac{\hat{y}_i - y_i}{y_i}\right\rvert$
NMAE	$\mathrm{NMAE} = \frac{100}{s_{\mathrm{ref}}}\mathrm{MAE}$
Pearson r	$r = \frac{\sum_{i=1}^{n}(y_i - \bar{y})(\hat{y}_i - \bar{\hat{y}})}{\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}\sqrt{\sum_{i=1}^{n}(\hat{y}_i - \bar{\hat{y}})^2}}$
Spearman ρ	$\rho = \frac{\sum_{i=1}^{n}(R_i - \bar{R})(\hat{R}_i - \bar{\hat{R}})}{\sqrt{\sum_{i=1}^{n}(R_i - \bar{R})^2}\sqrt{\sum_{i=1}^{n}(\hat{R}_i - \bar{\hat{R}})^2}}$

Table 4.2: Evaluation metrics definitions.

In Table 4.2, y_i denotes the ground truth target value for sample i, $\hat{y}_i$ the corresponding prediction, and n the number of test samples. $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$ is the sample mean of the ground truth. MAE

(Mean Absolute Error) measures the average absolute deviation in the natural units of the target, making it directly interpretable. RMSE (Root Mean Squared Error) uses squared errors and therefore penalizes occasional large mistakes more strongly than MAE. The coefficient of determination R^2 quantifies the fraction of variance explained relative to a constant baseline predictor equal to $\bar{y}$.

Percentage-based metrics provide scale-invariant comparisons across targets. MAPE (Mean Absolute Percentage Error) averages absolute relative errors, but is undefined when $y_i = 0$ and can become unstable when y_i is close to zero; we therefore compute it only over samples with strictly positive ground truth, where n_+ is the number of such samples. To obtain a more stable normalized error when small values may occur, we report NMAE (Normalized Mean Absolute Error), which normalizes MAE by a reference scale s_{ref} (e.g., the median of positive ground truth values computed on the training split for the considered target), expressed as a percentage.

Correlation coefficients complement error-based metrics by quantifying the strength and direction of association between predictions and ground truth. Pearson's correlation coefficient r measures linear dependence, where $\bar{\hat{y}} = \frac{1}{n}\sum_{i=1}^{n} \hat{y}_i$ denotes the sample mean of predictions; values close to $+1$ indicate strong positive linear agreement. Spearman's rank correlation ρ applies the same formula to ranks rather than raw values: R_i and $\hat{R}_i$ denote the ranks of y_i and $\hat{y}_i$ respectively within their samples, with $\bar{R}$ and $\bar{\hat{R}}$ their means. Spearman's ρ captures monotonic relationships without assuming linearity, making it robust to outliers and nonlinear transformations.

The resulting pipeline is modular and extensible. Each bucket trains independently, enabling parallel execution and straightforward addition of new layer types or execution states as the architecture evolves. Feature construction remains consistent across buckets, simplifying interpretation and enabling transfer of domain knowledge across contexts. The separation between problem formulation (data construction, observability, target factorization) and model instantiation (supervised learning workflow) ensures that methodological insights remain independent of specific algorithmic choices, and that improvements to either layer can proceed without disrupting the other.

4.4.2 Implementation

This subsection describes the concrete realization of the measurement infrastructure used to collect ground truth PPA data. The implementation instantiates the data construction and observability strategies defined in the previous subsection through specific simulation and synthesis workflows. These workflows transform synthetic layer specifications into labeled training data: RTL simulation produces latency measurements indexed by execution state, logic synthesis produces area measurements indexed by architectural module, and power analysis produces power estimates indexed by module when trace to netlist matching is sufficient. The machine learning pipeline that consumes

this data follows the protocol described in Section 4.4.1.4.

4.4.2.1 Stimuli Generation for CNN Inference

The stimuli generation phase bridges the gap between high level CNN specifications and hardware-ready testbenches. The process starts with a CNN model specification in C++ format, where each layer is described through parameterized constructors as shown in Figure 4.15 (1). Weight and activation files are provided in text format, defining the complete computational graph to be executed on the NoC-based hardware.

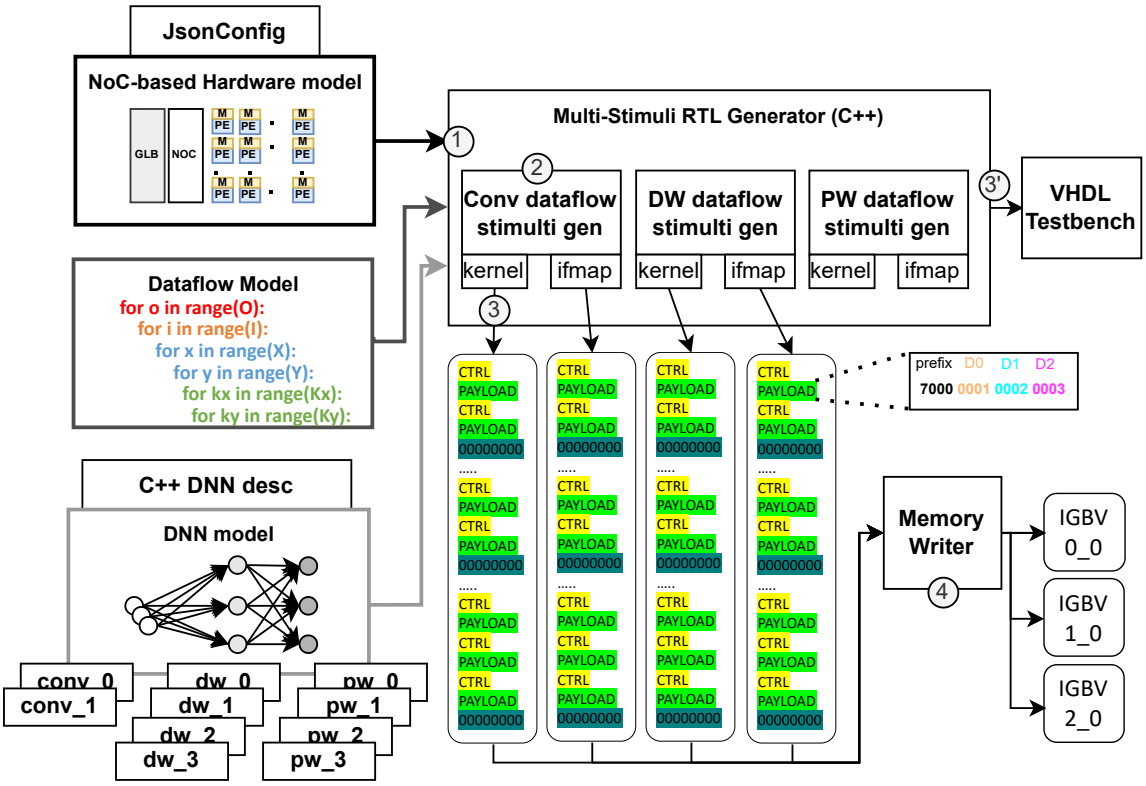


Figure 4.15: Multi Stimuli RTL Generator workflow: transforms CNN layer descriptions and dataflow models into VHDL testbenches and parameterized memory initialization files

Rather than generating traditional cycle by-cycle stimuli, the framework translates the dataflow orchestration directly into memory initialization patterns. For each layer type (convolution, depthwise, pointwise as shown in Figure 4.15 (2)), the C++ generator computes the corresponding access sequences and data dependencies according to the nested dataflow loop structure.

Each iteration generates a (control, payload) pair (as shown in Figure 4.15 (3)): the control word specifies routing and operation metadata, while the payload contains weights and activations. These

pairs are then encoded as hexadecimal blocks and dumped into global buffer initialization files (IGBV, IGBH, OGB, as shown in Figure 4.15 (4)). This memory-centric abstraction embeds computation directly within addressable memory regions, accessible to the hardware during initialization via standard VHDL readmemh directives.

Once memory contents are prepared, a minimal VHDL testbench (as shown in Figure 4.15 (3')) is automatically generated to instantiate the parameterized hardware architecture and assert the start signal. The testbench loads pre-computed memory files and initiates computation, delegating all timing and control flow to the RTL simulator. This decoupling of specification from timing enables flexible reuse across different simulation and synthesis configurations while maintaining reproducibility.

4.4.2.2 RTL Simulation Infrastructure

RTL-level simulation provides cycle-accurate hardware behavior during CNN layer execution using Mentor Graphics QuestaSim. The simulation infrastructure processes layers sequentially due to licensing constraints: only one simulation instance runs at a time (as shown in Algorithm 1). Simulations can execute a single layer or a batch of consecutive layers, with metrics collected per layer. Results from all simulation runs are aggregated into a unified dataset, indexed by layer type T (convolution, depthwise convolution, pointwise convolution) and execution state S from either Global Controller FSM or PE controller FSM.

The simulation infrastructure provides multiple operational modes. The primary mode, used for dataset generation, performs selective VCD tracing that captures only hardware performance counters from critical modules. These counters quantify key performance phenomena: cycle counts, memory contention, buffer utilization, PE occupancy. This targeted instrumentation produces compact traces optimized for rapid post processing and efficient storage.

For latency characterization, selective VCD instrumentation captures signals from two categories of hardware modules. The Global Controller provides state machine transitions and processing element activity flags that track the progression of computation through distinct phases. Individual Processing Element controllers instrument cycle counters and accumulation event signals that quantify local datapath activity. Memory access patterns are observed at the interface-level of global buffer modules (IGBV, IGBH, OGB), where read/write addresses and data values are logged. This instrumentation strategy captures hundreds of signals, sufficient to reconstruct both the sequence of execution states and their individual durations, without generating VCD files so large as to become impractical for post processing.

Analysis of the collected VCD traces yields two fundamental metrics per execution state S. The first metric, count, records the total number of times that state S is entered during the course of layer

execution. The second metric, mean_cycles, quantifies the average duration in clock cycles spent within state S each time it is visited. State transitions are detected by observing edges on the Global Controller state or PE controller state signals in the VCD trace. Within each state epoch, counter values are accumulated and then divided by the number of state visitations to derive mean_cycles. These per state measurements provide structured observability into execution behavior, decomposing layer execution into distinct computational phases (filter loading, feature map loading, computation, result writeback), each exhibiting distinct patterns of memory access, processing element utilization, and control overhead.

Algorithm 1 Multi stimuli RTL-level Simulation and Trace Collection

Phase 1: Framework Initialization
BUILDFRAMEWORK ▷ Compile application binary (optional)
Phase 2: Per Layer Processing for eachlayer $\in \mathcal{L}$ **do**
  Step 1: Stimuli and Testbench Generation
  GENERATESTIMULIANDMAPPING(layer) ▷ Multi Stimuli RTL Generator
  Step 2: Simulation configuration
  CONFIGURESIMULATION(layer) ▷ Parameterize simulation environment
  GENERATETRACECONFIGURATION ▷ Define VCD instrumentation counters
  EXECUTERTLSIMULATION ▷ Run HDL-level simulation with QuestaSim
  DUMPSIGNALTRACES(layer) ▷ Collect instrumentation counters activity data
    Traced signals include:
    Control signals (layer metadata, PE utilization)
    Datapath activity (buffer transfers, accumulations)
    Memory access patterns (read/write operations)
    Computation counters (arithmetic operations per PE)
  end for
Output:
Signal traces indexed by (T, S): layer type and execution state
Ground truth metrics: operation counts, cycle measurements per (T, S)
Dataset ready for supervised learning pipeline

Sequential execution ensures reproducibility but limits throughput during dataset generation. Automation scripts manage the simulation queue, coordinating layer scheduling, ensuring consistent parameter passing to the simulation environment, and organizing output trace files into a coherent directory structure. If simulation performance becomes a bottleneck, workloads can be further

parallelized using open-source simulators such as Verilator. The current selective VCD capabilities also establish a foundation for future power analysis studies, where complete signal dumps would be transformed into Switch Activity Interchange Format (SAIF) timing information to characterize dynamic power consumption across the design space.

4.4.2.3 Logic Synthesis Infrastructure

Logic synthesis characterizes FPGA resource utilization across the design space using Xilinx Vivado Design Suite version 2021.2. All synthesis experiments target the Xilinx Virtex UltraScale+ VU19P FPGA (part xcvu19p-fsvb3824-2 e), a high-capacity device suitable for prototyping large dataflow accelerators. This platform was selected to accommodate the full NoC-based dataflow architecture with 192 processing elements (3 rows of 64 PEs) and associated global buffer infrastructure without resource constraints. In contrast to the simulation process, the synthesis framework exploits task-level parallelism: multiple design configurations are synthesized concurrently through multi threaded orchestration (as shown in Algorithm 2). A thread pool manages independent synthesis jobs and distributes the computational load across available CPUs.

For each design configuration, RTL source files are instantiated with specific architectural parameters (PE dimensions, buffer sizes, dataflow type) via automated template substitution. These parameters are injected into the design hierarchy by regex-based modification of VHDL parameter declarations, ensuring exact parameterization without manual editing. The Vivado synthesis engine is then invoked with a fixed hierarchy flattening strategy (flatten_hierarchy=none), which preserves the modular structure of the original RTL design and prevents aggressive cross boundary optimizations that would obscure per module resource attribution.

Vivado generates detailed utilization reports at the module-level, capturing LUT usage (logic and distributed RAM), flip-flops (FFs), block RAM (BRAM), ultra-RAM (URAM), and DSP blocks. These per module counts are consolidated during post processing to produce a unified resource profile per design configuration. Module hierarchy aggregation reflects parent module counts as the sum of child module contributions, enabling accurate attribution of resource consumption to specific architectural components and supporting fine-grained area analysis.

Synthesis runtime exhibits substantial variability as a function of design complexity, ranging from approximately 1 hour for small configurations more than 3 hours for highly parameterized architectures. This variation arises from the increased complexity of logic optimization, placement, and routing, and the associated growth in resource demands. Moreover, the computational effort frequently scales in a non linear manner with architectural complexity and resource utilization.

Algorithm 2 Multi-Config Logic Synthesis and Resource Characterization

Phase 1: Multi-Config Logic Synthesis Initialization
INITIALIZETHREADPOOL(p workers) ▷ Create parallel synthesis environment
Phase 2: Per-Design Synthesisfor eachdesign $\in \mathcal{D}$ **do**
  Step 1: Parameter Instantiation
  LOADCONFIGURATIONFILE(CNN, architecture) ▷ Extract CNN dependent architecture dimensions
  GENERATEHDLPARAMETERS(design) ▷ Parameterize RTL constants via regex substitution
  Step 2: Hardware Synthesis
  INVOKEVIVADO(hierarchy flattening) ▷ Run Vivado synthesis engine
  GENERATEUTILIZATIONREPORTS ▷ Extract FPGA resource usage
    Captured metrics include:
    LUT utilization (logic, distributed RAM)
    Flip-flop and register counts
    Block RAM and URAM consumption
    DSP block allocation
  Step 3: Report Post-Processing
  PARSESYNTHESISREPORTS(design) ▷ Parse Vivado output files
  AGGREGATEBYMODULE ▷ Consolidate per module metrics
  EXPORTTOSPREADSHEET ▷ Generate CSV output
  end for
Output:
Resource metrics indexed by (T, S): design configuration and module type
Ground truth utilization: LUT, FF, BRAM, DSP per (T, S)
Dataset ready for supervised learning pipeline

To mitigate dataset collection time, the multi threaded orchestration distributes synthesis jobs across available processor cores through a work stealing task queue. Design checkpoints (DCP files) are persisted after synthesis completion, serving as starting points for Vivado Report Power analysis workflows and enabling traceability and incremental design refinement.

4.4.2.4 Area Estimation Implementation

FPGA resource utilization is determined statically at synthesis time by architectural parameters, independent of runtime execution patterns. The area estimation implementation extracts ground

truth resource counts directly from Vivado synthesis reports, then trains regression models to predict these counts from architectural and layer parameters.

Module decomposition follows the hardware hierarchy: processing elements (PE), global buffers (IGBV, IGBH, OGB), global controller, AI-NoC interconnect, bias adder, dimension converter, control word processor, and output buffer multiplexer. For each module, Vivado reports consumption of five resource types: lookup tables (LUTs), flip-flops (FFs), block RAMs (BRAMs), DSP blocks, and ultra-RAMs (URAMs). The framework therefore instantiates $M \times R$ independent regression models, where $M = 10$ architectural modules and $R = 5$ resource types, yielding 50 distinct submodels.

Module consolidation addresses the challenge of replicated hardware instances. The target architecture instantiates 192 processing elements, each reported individually in Vivado utilization reports. These instances are consolidated into a single PE resource profile through hierarchical summation, enabling the regression model to learn PE scaling behavior rather than memorizing individual instance counts. The same consolidation applies to replicated buffer banks and NoC router instances. Post-training inference reconstructs total resource utilization through hierarchical aggregation: each module-specific model predicts its resource consumption, and total utilization is computed as the sum of per-module predictions.

4.4.2.5 Latency Estimation Implementation

Latency estimation extracts ground truth timing measurements from RTL simulation traces, capturing cycle-accurate execution behavior across distinct dataflow phases. The latency model partitions data by layer type T (convolution, depthwise, pointwise, each with stride variants) and execution state S (filter loading, feature map loading, partial sum computation, result writeback).

The implementation learns two quantities per bucket (T, S): state visitation count c_s and mean cycles per visit μ_s. Total layer latency is reconstructed as:

$$\ell_{\text{layer}} = \sum_{s \in S} c_s \times \mu_s \tag{4.5}$$

Post-hoc validation compares this reconstructed latency against ground truth measurements obtained by summing cycle counts across the entire simulation. Floating-point regression outputs are discretized by rounding c_s and μ_s to the nearest integer and clipping to physical bounds.

4.4.2.6 Power Estimation Implementation

Power estimation requires switching activity information that captures dynamic behavior during workload execution. Unlike area, which is determined statically at synthesis time, and latency, which can be measured through cycle-accurate counters, power consumption depends on the toggle rates of individual nets throughout the design hierarchy.

The methodology follows a three-stage pipeline. First, design synthesis produces a gate-level netlist captured as a Vivado design checkpoint. Second, simulation generates switching activity traces converted to Switch Activity Interchange Format (SAIF). Third, Vivado report_power combines the netlist, activity traces, and FPGA device power model to compute power estimates.

Two simulation strategies are available. Pre-synthesis simulation operates at the RTL level, offering fast execution and seamless integration with the existing infrastructure. Post-synthesis simulation operates directly on the gate-level netlist with timing annotation, ensuring exact correspondence between simulated nets and the netlist under analysis, at the cost of substantially longer simulation times.

The current implementation provides the complete infrastructure for power estimation: synthesis checkpoint generation, VCD-to-SAIF conversion pipelines, and Vivado power analysis automation. Initial characterization has been performed on representative configurations from MobileNetV1, verifying end-to-end execution of the power-reporting pipeline.

Chapter 5

Experimental Results and Validation

5.1 Experimental Overview

This chapter validates the PANDA framework through a series of experiments designed to answer three fundamental questions:

- Can the automated RTL/synthesis pipeline generate ground truth measurements that are consistent and informative enough to train and validate learned PPA surrogates?
- Does the mapping model accurately capture the memory access patterns and computational behavior observed in cycle accurate hardware simulation, while enabling rapid evaluation for design space exploration?
- Can learned PPA surrogates trained on the datasets predict latency and area with sufficient accuracy to reduce reliance on expensive RTL evaluation during architectural exploration, and what are the current limitations for power?

To address these questions, we organize our experimental evaluation as follows. We begin by describing the target architecture instance and the workloads that serve as our validation benchmark (section 5.2). We then validate the RTL simulation infrastructure by examining detailed memory access patterns and processing element utilization across representative layer types, comparing observed behavior against manual calculations to establish ground truth correctness (section 5.3).

Building on this validated ground truth, we evaluate the mapping model's ability to reproduce these patterns through high-level loop abstractions, quantifying the correspondence between analytical predictions and RTL measurements (section 5.4).

Finally, we assess the PPA surrogate models: area prediction accuracy across architectural configurations through module decomposed ML-based surrogate model (subsection 5.5.1), latency estimation through state decomposed ML-based surrogate model (subsection 5.5.2), and the current limitations encountered in power characterization (subsection 5.5.3). We conclude with a synthesis of methodological insights and identified scope boundaries (section 5.6).

5.2 Experimental Setup

This section describes the experimental setup used throughout the validation: the target architecture instance, the MobileNet v1 benchmark, and the synthetic datasets used to train the surrogate models. We then detail the RTL simulation and FPGA synthesis toolchain, report the associated computational cost, and validate the multi stimuli RTL generator.

5.2.1 Compute environment

All RTL simulations and FPGA syntheses were run on a Linux server (Ubuntu kernel 4.15.0-213-generic) equipped with two Intel Xeon Gold 6154 CPUs (18 cores per socket, 2 threads per core; 36 physical cores / 72 hardware threads) and 187 GiB of RAM (8 GiB swap). Experiments were parallelized across independent jobs when possible; overall throughput was primarily limited by EDA (Electronic Design Automation) tool runtimes and licensing constraints.

5.2.2 Target Architecture

The experiments target the NoC-based dataflow architecture developed by CEA, instantiated on a Xilinx Virtex UltraScale+ VU19P FPGA [118]. This architecture serves as a research prototype for exploring CNN accelerator design points through parametric instantiation.

The compute fabric organizes processing elements into a two-dimensional grid. These PEs are interconnected through a mesh-based Network-on-Chip. Each PE couples a MAC-based compute unit with a local router, enabling distributed dataflow execution where computation and communication overlap at tile granularity. The grid dimensions are tied to layer parameters: the number of rows corresponds to the kernel height, while the number of columns sets the maximum output width processable in a single pass.

Memory is distributed across three buffer types positioned along the grid perimeter. Input Global Buffers Vertical (IGBV, one bank per row on the left edge) store filter weights and multicast them

horizontally across PE rows. Input Global Buffers Horizontal (IGBH, one bank per column along the bottom) store input feature maps for vertical or diagonal distribution toward the PE array. Output Global Buffers (OGB, along the top) collect computed results and serve as intermediate storage between consecutive layers, enabling on-chip data reuse without off-chip memory accesses.

The choice of the fixed 3×64 configuration (as shown in Figure 5.1) for our experiments represents an effective trade-off between parallelism, memory bandwidth, and interconnection complexity, ensuring stable and efficient utilization of hardware resources for the targeted input resolutions and target CNN MobileNet V1 backbone $(\alpha = 0.25,$ input $128 \times 128)$ generated from N2D2 [108]. We chose a 128×128 input image for our main evaluation to facilitate mapping onto our target architecture instead of using a 224×224 representation as in standard benchmarks.

The 3×64 configuration, comprising 192 PEs, serves as the reference instance for RTL validation, mapping experiments, and latency/power characterization throughout this work. The area study explores a broader range of grid dimensions: 3, 5, or 7 PE rows (matching kernel heights of 3×3, 5×5, and 7×7 convolutions), and 5 to 116 columns. We synthesize 300 configurations spanning this design space to characterize how area scales with architectural parallelism.

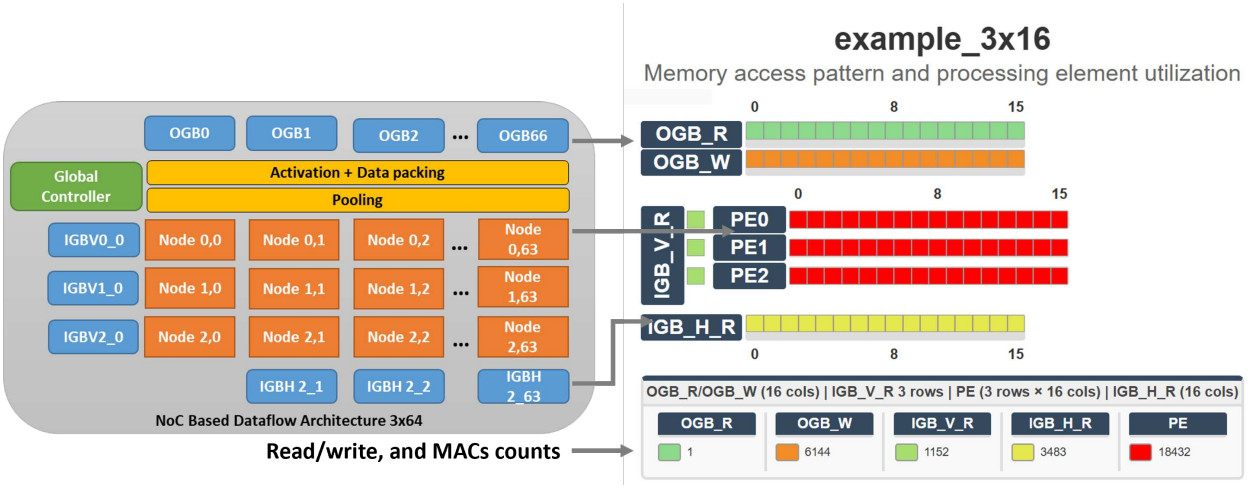


Figure 5.1: Target architecture instance: 3×64 PE grid with distributed memory organization (left) and execution visualization example (right). Color variation encodes either memory access count or MAC operation density.

Two architectural revisions are used across the experiments. The RTL validation, mapping model evaluation, and area study use an earlier revision where all memory banks share a uniform address width of 17-bit, yielding $2^{17} = 131,072$ entries per bank. The latency and power characterization use a later revision supporting per buffer type address widths (as described in Table 4.1), enabling heterogeneous memory allocation across IGBV, IGBH, and OGB banks.

5.2.3 CNN Workloads

With the architecture established, we now focus on the CNN workloads employed for evaluation. We rely on two datasets: (i) the MobileNet v1 backbone benchmark, used to validate the 3×64 reference architecture and the mapping framework; and (ii) a synthetic dataset of 300 layers, used to train and evaluate the latency ML-based models. For power characterization, we use an augmented convolution-only subset of 100 layers: the 50 convolution configurations already included in the 300-layer dataset, plus 50 additional convolutions generated to improve coverage while keeping full-VCD simulations tractable.

MobileNet v1 is used as the validation workload, selected for its widespread adoption in embedded AI vision systems and its extensive use of depthwise separable convolutions. It covers all supported operation types: standard convolution (Conv), depthwise convolution (DW), and pointwise convolution (PW). Table 5.1 lists the 25 layers executable on the uniform memory revision of the 3×64 reference architecture.

Layer	Type	Kernel	Input	Channels	Stride	Pad.	MACs
MN_Conv0	Conv	3×3	128×128	$3 \rightarrow 8$	2	1	885K
MN_DW0	DW	3×3	64×64	$8 \rightarrow 8$	1	1	295K
MN_PW0	PW	1×1	64×64	$8 \rightarrow 16$	1	0	524K
MN_DW1	DW	3×3	64×64	$16 \rightarrow 16$	2	0	147K
MN_PW1	PW	1×1	32×32	$16 \rightarrow 32$	1	0	524K
MN_DW2	DW	3×3	32×32	$32 \rightarrow 32$	1	1	295K
MN_PW2	PW	1×1	32×32	$32 \rightarrow 32$	1	0	1.05M
MN_DW3	DW	3×3	32×32	$32 \rightarrow 32$	2	0	74K
MN_PW3	PW	1×1	16×16	$32 \rightarrow 64$	1	0	524K
MN_DW4	DW	3×3	16×16	$64 \rightarrow 64$	1	1	147K
MN_PW4	PW	1×1	16×16	$64 \rightarrow 64$	1	0	1.05M
MN_DW5	DW	3×3	16×16	$64 \rightarrow 64$	2	0	37K
MN_PW5	PW	1×1	8×8	$64 \rightarrow 128$	1	0	524K
MN_DW6	DW	3×3	8×8	$128 \rightarrow 128$	1	1	74K
MN_PW6	PW	1×1	8×8	$128 \rightarrow 128$	1	0	1.05M
MN_DW7	DW	3×3	8×8	$128 \rightarrow 128$	1	1	74K
MN_PW7	PW	1×1	8×8	$128 \rightarrow 128$	1	0	1.05M
MN_DW8	DW	3×3	8×8	$128 \rightarrow 128$	1	1	74K
MN_PW8	PW	1×1	8×8	$128 \rightarrow 128$	1	0	1.05M
MN_DW9	DW	3×3	8×8	$128 \rightarrow 128$	1	1	74K
MN_PW9	PW	1×1	8×8	$128 \rightarrow 128$	1	0	1.05M
MN_DW10	DW	3×3	8×8	$128 \rightarrow 128$	1	1	74K
MN_PW10	PW	1×1	8×8	$128 \rightarrow 128$	1	0	1.05M
MN_DW11	DW	3×3	8×8	$128 \rightarrow 128$	2	0	18K
MN_DW12	DW	3×3	4×4	$256 \rightarrow 256$	1	1	36K
						Total (25 layers)	**11.74M**

Table 5.1: $\mathrm{MobileNetV1}$ $(\alpha = 0.25,$ input $128 \times 128)$ backbone layers executed in the experiments.

The network combines three operation types with distinct dataflow behaviors. Standard convolution (Conv0) processes the input image with high spatial dimensions (128×128) but few channels (3 input, 8 output), exercising the architecture under memory bound conditions. It employs the ADP

dataflow pattern.

Depthwise convolutions (DW) apply per channel 3×3 spatial filtering with diagonal or vertical data distribution patterns, testing the NoC multicast primitives. It employs the ADP dataflow pattern.

Pointwise convolutions (PW) perform channel mixing through 1×1 kernels with an output stationary dataflow, shifting the bottleneck toward compute throughput as channel counts increase. It employs an output-stationary dataflow pattern.

Across these 25 layers, spatial dimensions reduce progressively from 128×128 to 4×4 while channel depth expands from 3 to 256, covering a wide range of compute to memory ratios. The remaining layers are omitted for this revision due to buffer capacity limits : PW11, PW12.

Training the latency ML-based models and performing power characterization requires a dataset that spans the feasible configuration space of the 3×64 reference architecture. Table 5.2 summarizes this full space per layer type and reports the corresponding number of feasible configurations.

Type	Output size	Channels in	Channels out	Stride	Configurations
Conv	$[5 \times 5, 64 \times 64]$	3 (fixed)	$[3, 64]$	$1, 2$	7,688
DW	$[4 \times 4, 64 \times 64]$	-	$[3, 256]$	$1, 2$	22,860
PW	$[4 \times 4, 64 \times 64]$	$[3, 256]$	$[3, 256]$	1	3,865,440
Total feasible configurations					**3.89M**

Table 5.2: Design space summary per layer type for the 3×64 reference architecture.

For the synthetic training dataset, all layers respect the hardware constraints: kernels are 3×3 (Conv/DW) or 1×1 (PW), strides are restricted to $1, 2$, output spatial dimensions are bounded to 64×64, and channel counts are limited to powers of two (8 to 256). In contrast, Table 5.2 includes the full feasible ranges (e.g., channel values such as 3) even if some of these values are not sampled in the synthetic dataset.

For the MobileNet v1 benchmark, layer parameters are fixed by the network definition (e.g., 3 input channels in the first layer), and only the subset that fits the considered architecture revision is executed.

For power characterization, we further restrict the dataset to standard convolutions with stride 2 to keep full-VCD simulations tractable.

With nearly 4 million feasible configurations, exhaustive simulation is impractical. We therefore generate 300 configurations using stratified sampling with the C++ multi stimuli generator (subsubsection 4.4.2.1), representing approximately 0.008% of the design space. Because uniform sampling would be dominated by pointwise layers, we deliberately stratify by operation type: 50 standard convolutions (strides 1 and 2, input resolutions 8×8 to 128×128), 150 depthwise convolutions (75

per stride, channels 8 to 256), and 100 pointwise convolutions (even split between same channel and channel-expansion cases).

This stratification is necessary because pointwise layers dominate the design space (99.2% of configurations), yet each layer type operates under a distinct execution modes: memory bound for standard convolutions, communication-intensive for depthwise, and compute bound for pointwise. The ML-based models require sufficient examples from each execution modes to capture latency variations across the full behavioral spectrum. Power characterization is attempted separately on a restricted convolution subset due to the cost of full-VCD simulation. Duplicate configurations are eliminated during generation, and all retained layers are validated against buffer capacity constraints. This dataset also serves to validate the models beyond MobileNet v1.

5.2.4 Toolchain, Stimuli Generation, and Computational Cost

The experimental pipeline relies on two main tools. QuestaSim [119] performs cycle accurate RTL simulation with configurable instrumentation depth. For latency characterization, selective VCD tracing captures only the Global Controller FSM state, PE controller states, and memory interface counters, keeping trace files manageable at a few hundred megabytes per layer. For power characterization, complete signal dumps are required to generate SAIF activity files, increasing trace sizes to 1 to 30 GB per convolution layer and significantly extending simulation time. Vivado 2021.2 [99] handles logic synthesis targeting the VU19P device, with hierarchy flattening disabled to preserve module boundaries and enable per component resource attribution.

5.2.4.1 Stimuli generation and correctness checks

The C++ multi stimuli produces memory initialization files and VHDL testbenches for arbitrary layer configurations (as shown in subsubsection 4.4.2.1). Validation proceeds in two steps. First, generated memory files are checked for structural correctness: proper interleaving of control and data words, valid routing coordinates, and consistent packing factors. Second, functional RTL simulation verifies that each generated layer executes to completion without deadlock and produces the expected number of MAC operations. All 300 synthetic layers and 25 available MobileNet v1 layers pass both validation stages.

5.2.4.2 Computational cost

Dataset generation costs vary substantially across experiments. Stimulus generation (ifmaps and filters) through the C++ generator completes in approximately 4 hours for 300 layers.

RTL simulation for latency requires approximately 72 hours for the full dataset, constrained by sequential execution under single seat licensing of Questasim; individual layers range from 2 to 40 minutes depending on complexity. Power simulation with full VCD dumps is considerably slower; we therefore restricted the power dataset to 100 convolution layers to maintain tractable generation time.

The area characterization campaign, which requires an independent synthesis run for each architectural configuration, is the longest phase: at roughly 10 runs per day with 2 to 3 syntheses executed in parallel, generating the 300 configuration dataset required approximately 30 days of continuous execution.

Simulation cost varies substantially across layer types (as shown in Table 5.3) and follows the amount of work performed by the architecture rather than the spatial size alone.

Layer	Type	Dimensions	Cycles	Sim. time
MN_Conv0	Conv	$128 \times 128 \rightarrow 64 \times 64$, $3 \rightarrow 8$ ch.	402K	2m 23s
MN_DW9	DW	$8 \times 8 \rightarrow 8 \times 8$, $128ch.$	48K	41s
MN_PW9	PW	$8 \times 8 \rightarrow 8 \times 8$, $128 \rightarrow 128$ ch.	1.2M	21m

Table 5.3: Representative simulation costs of MobileNet v1 on the CEA architecture using QuestaSim.

Pointwise convolutions require the longest runs because their execution scales with channel mixing ($C_{in} \times C_{out}$) even when operating on small feature maps (MN_PW9: 1.2M cycles for 8×8 with $128 \rightarrow 128$ channels). In contrast, MN_Conv0 processes a much larger spatial input but remains comparatively cheaper due to its limited channel counts ($3 \rightarrow 8$), resulting in fewer accumulated operations and a shorter cycle trace. Depthwise layers are consistently the least expensive since they filter channels independently, reducing both the number of MAC operations and the associated control/communication overhead.

5.3 Ground Truth Data Generation

Having established the simulation infrastructure, we now validate its correctness using the MobileNet v1 benchmark. This validation confirms that the automated stimuli generator produces executable configurations without deadlock across all 25 layers. We examine memory transaction patterns, processing element utilization, and computational behavior across representative layer types, and compare simulation traces with manual calculations derived from layer parameters and architectural constraints.

5.3.1 Conv0: Standard Convolution Analysis

Conv0 executes using ADP dataflow (kernel row by output row parallelism, as shown in subsection 4.2.6), with the 3 PE rows processing kernel rows in parallel and the 64 PE columns computing output pixels across the spatial width.

The simulation trace Figure 5.2 captures raw transaction counts that include control signals, data payloads, and packing overhead from three memory hierarchies: IGB_H banks unicast input features vertically, IGB_V banks multicast filter weights horizontally, and OGB banks accumulate partial results through column wise transmission. To verify correctness, we decode these observations using the architectural protocol and compare against expected values computed from layer geometry.

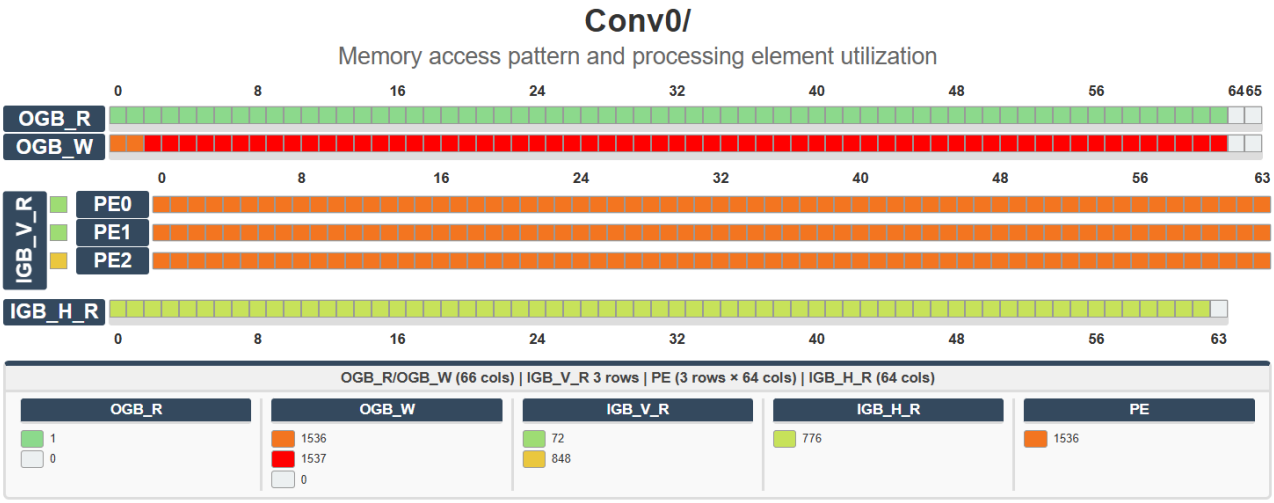


Figure 5.2: Conv0 RTL simulation trace: memory transactions per bank (OGB_R/OGB_W top, IGB_V_R left, IGB_H_R bottom) and MAC operations per processing element (center matrix). Color variation indicates activity-level.

5.3.1.1 Input Feature Map Distribution

IGB_H banks show 776 raw transactions per bank. The architecture uses an interleaved control/data protocol where each logical transfer generates two transactions (one control word, one data word), and packs 3 data elements per payload. Additionally, separator values are inserted between the $I = 3$ input channels. The actual number of data elements transferred is:

$$N_{\text{IGB_H}} = \frac{T_{\text{raw}}}{2} \times P - I = \frac{776}{2} \times 3 - 3 = 1161 \text{ elements per bank} \tag{5.1}$$

where $T_{\text{raw}} = 776$ is the raw transaction count, division by 2 accounts for control/data interleaving, $P = 3$ is the packing factor (elements per payload), and $I = 3$ is the number of input channels (used

here as separator count). With 64 IGB_H banks serving a stride-2 convolution over a $W \times H = 128 \times 128$ input and padding 1, this value matches the expected unicast volume per PE in the column.

5.3.1.2 Filter Distribution

IGB_V shows 72 transactions total across the 3 banks. Filters use a simpler encoding where each transaction carries one weight coefficient:

$$N_{\text{IGB_V}} = \frac{O \times Kx \times Ky}{Ky} = \frac{8 \times 3 \times 3}{3} = 24 \text{ elements per bank} \tag{5.2}$$

where $O = 8$ is the number of output channels, $Kx \times Ky = 3 \times 3$ is the kernel size, and division by $Ky = 3$ distributes weights across the 3 PE rows (one per kernel row). Each IGB_V bank stores 24 weight elements and multicasts them horizontally to all 64 PEs in its row. The 72 transactions observed (24×3 banks) confirm the expected distribution pattern.

5.3.1.3 Output Accumulation and MAC Validation

OGB banks show 1,536 write transactions per column. Each transaction writes one partial sum to be accumulated across input channels:

$$N_{\text{OGB}} = Y \times O \times I = 64 \times 8 \times 3 = 1{,}536 \text{ writes per column} \tag{5.3}$$

where $Y = 64$ is the output height, $O = 8$ is the number of output channels, and $I = 3$ is the number of input channels. This confirms that each of the 64 PE columns produces one partial sum per output row, per output channel, per input channel.

The total MAC count validates computational correctness. Each partial sum requires $Kx \times Ky = 3 \times 3 = 9$ MAC operations, yielding per column:

$$\text{MAC}_{\text{column}} = N_{\text{OGB}} \times Kx \times Ky = 1{,}536 \times 9 = 13{,}824 \text{ operations} \tag{5.4}$$

Distributed across 3 PEs per column, this gives $13{,}824/3 = 4{,}608$ MACs per PE. The simulation trace confirms 1,536 MAC events per PE, each triggering 3 parallel operations within the PE's compute unit (specific to standard convolutions), yielding $1{,}536 \times 3 = 4{,}608$ MACs per PE. Across all 64 columns:

$$\text{MAC}_{\text{total}} = \text{MAC}_{\text{column}} \times X = 13{,}824 \times 64 = 884{,}736 \text{ operations} \tag{5.5}$$

This validates both the computational workload and the parallel execution pattern across the 192-PE array.

5.3.2 Validation Summary Across Layer Types

Table 5.4 summarizes the key metrics across the three representative layers analyzed in detail above.

Metric	Conv0	DW0	PW1
Input source	IGB_H (1,161 elem.)	OGB (1,025 trans.)	OGB (1,024/bank)
Filter distribution	IGB_V (72 elem.)	IGB_V (72 elem.)	IGB_V (13,312 trans.)
Output writes	OGB_W (1,536/col.)	OGB_W (1,536)	OGB_W (3,072)
Total MACs	884,736	98,304	524,288
Dataflow pattern	Unicast + multicast (ADP)	Diagonal + multicast (ADP)	Unicast + multicast (OS)

Table 5.4: RTL validation summary across representative layer types. All observed values match manual calculations within measurement precision ($< 0.1\%$ error from timing artifacts).

DW0 (depthwise convolution, 64×64 input, 8 channels, stride 1) exhibits a diagonal multicast ifmap and multicast filter memory access pattern. Input feature maps are read from OGB rather than IGB_H, with 1,025 OGB_R transactions observed. After accounting for control/data interleaving ($1{,}025/2 \approx 512$ reads per bank), scaling across 64 banks yields $512 \times 64 = 32{,}768$ pixels, exactly matching the $64 \times 64 \times 8$ input volume. The diagonal multicast pattern efficiently distributes these inputs across the PE array. Filter traffic remains minimal (72 IGB_V transactions for 8 filters of size 3×3), and the total MAC count of 98,304 operations confirms correct depthwise execution.

PW1 (pointwise convolution, 32×32 spatial, $16 \rightarrow 32$ channels) uses an output stationary dataflow with unicast input distribution and multicast filter. OGB_R shows 1,024 transactions per bank across 32 active banks, totaling 32,768 reads. After control/data separation, this yields $32{,}768/2 = 16{,}384$ elements, matching the $32 \times 32 \times 16$ input volume. Filter distribution via IGB_V multicast shows 13,312 transactions, which after normalization accounts for the 16×32 channel mixing plus alignment padding (16 input channels distributed across 3 PE rows requires padding to 18 effective channels). The non-uniform PE utilization reflects this channel alignment: the first row processes 6 input channels (6,144 MACs per row), while the second and third rows each process 5 channels (5,120 MACs per row). The total of 524,288 MACs validates the $32 \times 32 \times 16 \times 32$ pointwise operation.

All 25 MobileNet v1 layers pass validation with observed values matching calculations within 0.1% (discrepancies arise from asynchronous timing artifacts during snapshot capture). The 300 synthetic layers similarly execute without deadlock and produce expected MAC counts, confirming the correctness of the automated stimuli generation pipeline.

5.3.3 Global Execution Characterization

Figure 5.3 demonstrates distinct behavioral patterns across layer types. Standard convolution (Conv0) and most pointwise layers exhibit compute-dominant profiles where COMPUTE time significantly exceeds data loading overhead (e.g., PW9: 1.15M cycles computation vs 50K cycles loading, 96% compute utilization). This indicates effective PE array utilization for channel mixing operations where high arithmetic intensity amortizes memory access costs.

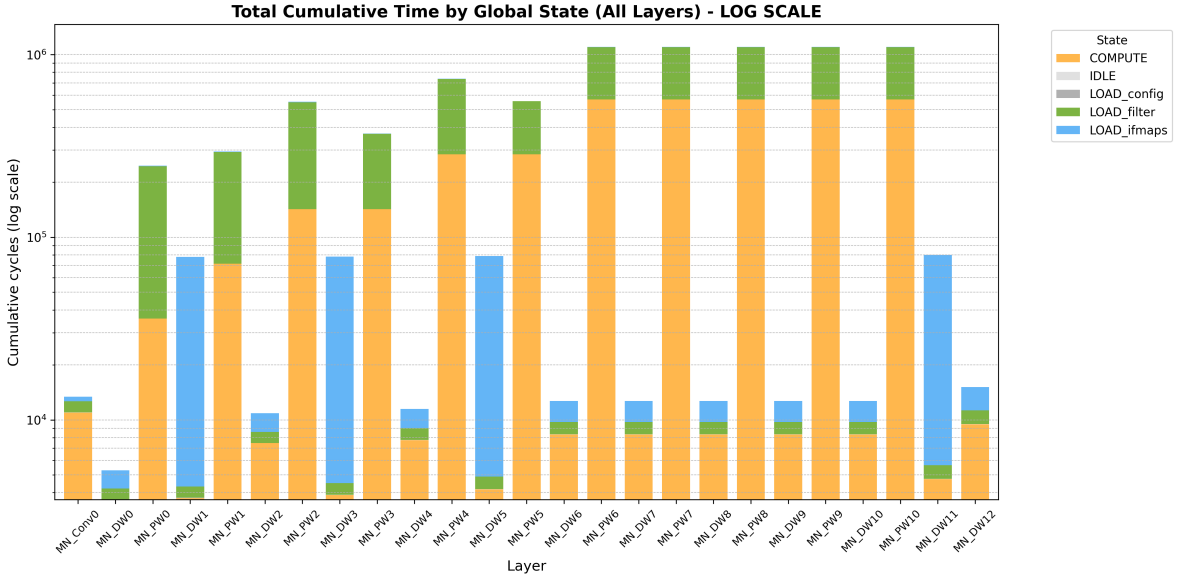


Figure 5.3: Execution time breakdown by state for all MobileNet v1 layers (logarithmic scale).

In contrast, several depthwise convolutions (DW1, DW3, DW5) show substantial LOAD_ifmaps contributions, with some layers spending more time loading inputs than computing. DW1 allocates 120K cycles to input loading versus only 80K cycles to computation (40% compute utilization).

This overhead stems from a suboptimal dataflow implementation in the evaluated architecture revision; the newer revision addresses this bottleneck through improved input staging. Filter loading (LOAD_filter) remains modest across all layers (<10% of execution time), validating the horizontal multicast primitive's efficiency. IDLE and LOAD_config overhead stays negligible (<1% even for small layers), confirming tight coupling between execution phases.

The nearly two-order-of magnitude range in execution time (18K cycles for DW11 to 1.2M cycles for PW9) correlates with total MAC count rather than spatial dimensions. PW9 (8×8 spatial, $128 \rightarrow 128$ channels, 1.05M MACs) requires 21 minutes of simulation despite processing 256× fewer pixels than Conv0 (128×128 spatial, $3 \rightarrow 8$ channels, 0.88M MACs, 2 minutes). This confirms that the architecture's bottleneck shifts from memory bandwidth (Conv layers) to compute throughput (PW

layers) as channel counts increase, a characteristic that the subsequent ML-based models must capture.

5.4 Mapping Model Evaluation

The mapping model's analytical formulation (as shown in section 4.3) derives memory transactions and MAC operations from layer parameters through hierarchical loop traversal, without cycle accurate simulation. Here we evaluate whether these derived values correspond to RTL measurements, establishing whether the model can replace expensive simulation during design space exploration. The critical question is not whether the model matches RTL protocol details (control/data interleaving, packing, bank assignments), but whether it correctly computes the computational workload and logical data movement volumes that determine performance.

5.4.1 Model Abstraction and Output Semantics

The mapping model operates on parameterized loop structures that can represent arbitrary array configurations (Figure 5.4), in contrast to RTL simulation which is tied to fixed array dimensions.

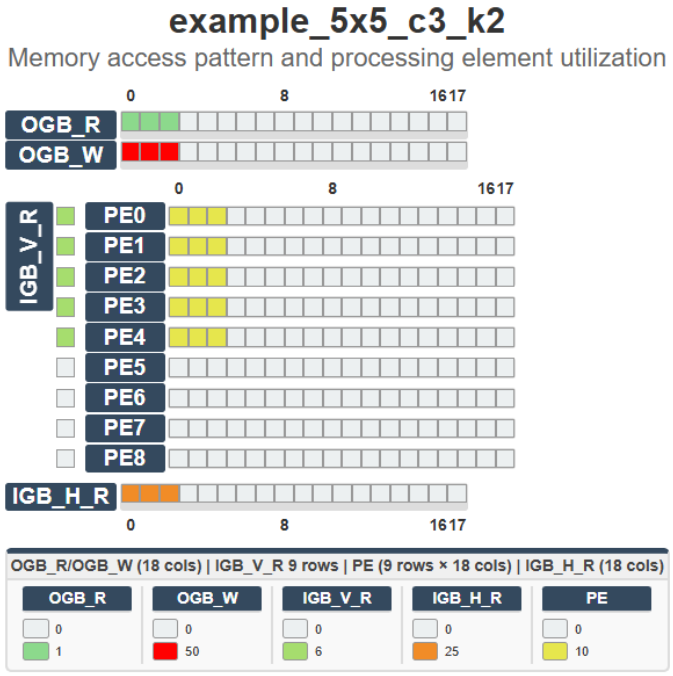


Figure 5.4: Parameterized loop structure output for a 5×32 configuration: the mapping model supports flexible hardware configurations beyond the fixed 3×64 reference array

Beyond dimensional flexibility, the model decouples layer type from dataflow strategy: while the RTL implementation binds each layer type (standard, depthwise, pointwise) to a specific dataflow, the mapping model can evaluate any layer under any dataflow variant.

This decoupling, combined with the absence of buffer capacity constraints, has three practical implications: (1) layers exceeding physical memory bounds can be evaluated to determine minimum buffer requirements before synthesis; (2) different PE grid dimensions can be compared without regenerating RTL; and (3) alternative dataflow mappings can be explored for a given layer to identify optimal execution strategies.

Memory access indicators differ semantically from RTL transaction counts. OGB_R is a boolean flag indicating whether input feature maps are sourced from OGB (1) or from IGB_H (0). The remaining indicators are element counters: IGB_H_R reports elements per bank, IGB_V_R reports the total multicast volume (elements × recipient columns), OGB_W reports output writes per column, and PE reports MAC operations per processing element. To enable visual comparison with RTL traces, the mapping model aggregates these counters into the same visualization format (Figure 5.5).

5.4.2 Correspondence Validation

Figure 5.5 presents the mapping model output for Conv0. The model derives the following values from layer parameters ($W \times H = 128 \times 128$, $I = 3$, $O = 8$, $Kx \times Ky = 3 \times 3$, $stride = 2$).

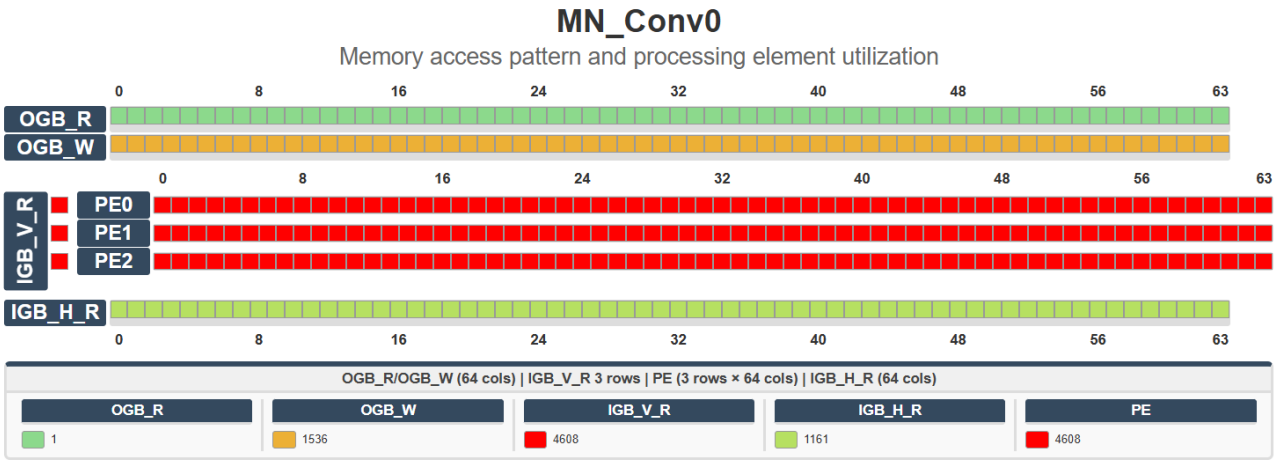


Figure 5.5: Conv0 mapping model output: memory access pattern and PE utilization. Legend shows per element values: OGB_R (boolean), OGB_W (writes/column), IGB_V_R (multicast volume), IGB_H_R (elements/bank), PE (MACs/PE).

5.4.2.1 Input Source Flag

The mapping model sets OGB_R = 0, indicating that input feature maps are routed through IGB_H rather than OGB. This boolean flag correctly distinguishes standard convolutions (IGB_H sourcing) from depthwise and pointwise layers (OGB sourcing).

5.4.2.2 Input Feature Map Distribution

The mapping model derives the input element count per IGB_H bank:

$$N_{\text{IGB_H}} = \frac{(H - Ky) \times Ky}{\text{stride}} + Ky \times I = \frac{(128 - 3) \times 3}{2} + 3 \times 3 = 387 \times 3 = 1{,}161 \text{ elements/bank} \quad (5.6)$$

This value matches the RTL measurement after protocol normalization (776 raw transactions for 1,161 elements via the decoding formula established in section 5.3).

5.4.2.3 Filter Distribution

The mapping model computes the total multicast volume for filter weights:

$$N_{\text{IGB_V}} = O \times Kx \times Ky \times X = 8 \times 3 \times 3 \times 64 = 4{,}608 \text{ logical accesses} \quad (5.7)$$

where $X = 64$ is the number of PE columns receiving the multicast. The 72 filter elements ($O \times Kx \times Ky = 72$) are unicast to all 64 columns, yielding $72 \times 64 = 4{,}608$ total accesses. This representation captures the multicast amplification that determines NoC traffic.

5.4.2.4 Output Accumulation

The mapping model derives output writes per column:

$$N_{\text{OGB_W}} = Y \times O \times I = 64 \times 8 \times 3 = 1{,}536 \text{ writes/column} \quad (5.8)$$

where $Y = 64$ is the output height, $O = 8$ the number of output channels, and $I = 3$ the number of input channels contributing partial sums. This matches the RTL observation exactly.

5.4.2.5 Computational Workload

The mapping model derives MAC operations per processing element:

$$N_{\text{PE}} = \frac{\text{MAC}_{\text{total}}}{R \times X} = \frac{X \times Y \times O \times Kx \times Ky \times I}{R \times X} = \frac{884{,}736}{3 \times 64} = 4{,}608 \text{ MACs/PE} \tag{5.9}$$

where $R = 3$ is the number of PE rows and $X = 64$ the number of columns. This value matches the RTL trace exactly, validating the computational workload derivation.

5.4.3 Validation Summary

Table 5.5 summarizes correspondence and speed-up across representative layers. For DW0 and PW1, the mapping model shows identical functional behavior despite differences in dataflow strategy (ADP for DW0 input distribution, output stationary for PW1). The OGB_R flag correctly distinguishes layers sourcing inputs from OGB (DW0, PW1) versus IGB_H (Conv0). Minor discrepancies (<0.1%) arise from asynchronous timing artifacts during RTL snapshot capture, not from model inaccuracies. The mapping model achieves speedups of 35× to 250× over RTL simulation while maintaining exact MAC correspondence and near-exact memory correspondence.

Layer	RTL sim. time	Mapping time	MAC error	Memory error
Conv0	2m 23s	<4s	0.00%	0.00%
DW0	41s	<3s	0.00%	<0.1%
PW1	21m	<5s	0.00%	<0.1%

Table 5.5: Mapping model validation: exact MAC correspondence and near-exact memory correspondence across representative layers

The correspondence between analytical derivation and RTL measurement confirms that the hierarchical loop structure correctly captures computation and memory flow patterns. This equivalence establishes the mapping model as a reliable abstraction for rapid design space exploration without full RTL simulation overhead.

5.5 PPA Model Evaluation

This section evaluates the predictive models developed for performance, power, and area estimation. The objective is twofold: first, to assess the accuracy of each model with respect to reference measurements obtained from detailed simulation and synthesis flows. Second, to analyze the limitations and strengths of the models in the context of architectural exploration, where fast estimation and correct ranking of design alternatives are critical.

For each PPA dimension, we follow a consistent structure that separates dataset characterization, behavioral or resource analysis, predictive signal assessment, and interpretation of results. The evaluation metrics (MAE, MAPE, NMAE, R^2) follow the definitions introduced in Table 4.2. For latency we additionally employ Spearman's rank correlation coefficient ρ to quantify the preservation of relative ordering between predictions and ground truth. It is particularly relevant for design space exploration where ranking candidates correctly may matter more than achieving exact absolute values. Spearman's ρ ranges from -1 to $+1$, with $\rho = 1$ indicating perfect ranking concordance.

5.5.1 Area Model Evaluation

This section evaluates the area prediction models by examining the synthesis dataset, analyzing resource distribution across the RTL hierarchy, assessing feature target correlations, and reporting prediction accuracy on held-out configurations. The goal is to determine whether the models can reliably rank architectural candidates by FPGA resource utilization.

5.5.1.1 Dataset Characterization and Coverage

We first characterize the synthesis dataset used to train and evaluate the area prediction models. The dataset consists of 300 architectural configurations generated by varying spatial and structural parameters of the PE array. The effective array size is controlled by the derived scaling variable $GRID = filter_width \times output_width$, where filter_width corresponds to the number of PE rows (matching kernel height) and output_width to the number of PE columns. Figure 5.6 reports the distribution of GRID values across all generated configurations using a logarithmic x axis.

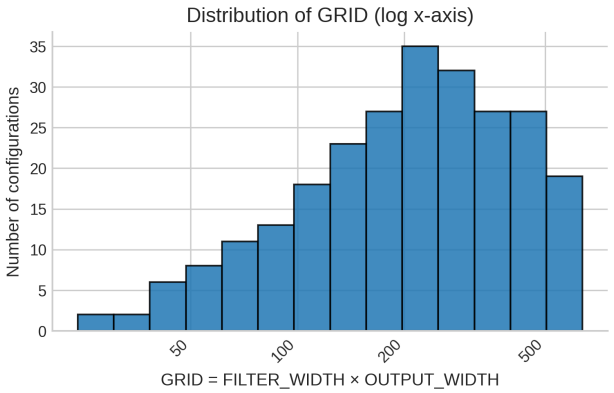


Figure 5.6: Distribution of $GRID = filter_width \times output_width$ (log scaled x axis). The dataset spans more than one order of magnitude in PE array size, ensuring coverage of diverse architectural configurations.

The log scale representation highlights that the dataset spans more than one order of magnitude in PE array size, rather than being concentrated around a narrow set of typical configurations. This broad coverage is essential for generalization, as both logic utilization and interconnect cost scale primarily with the number of instantiated PEs.

5.5.1.2 Resource Distribution Analysis

Before building predictive models, we analyze how FPGA resources are distributed across the RTL hierarchy. This analysis identifies which modules dominate the area footprint and therefore require accurate prediction.

Figure 5.7 reports the median resource utilization per module and per resource type across all synthesis runs. To accommodate large dynamic ranges and zero-valued entries, values are displayed as $log10(\text{median} + 1)$: where 0 indicates unused resources and higher values reflect increasing utilization on a logarithmic scale. For instance, the pe module reaches 5.9 for Total LUTs, corresponding to approximately 800,000 LUT instances.

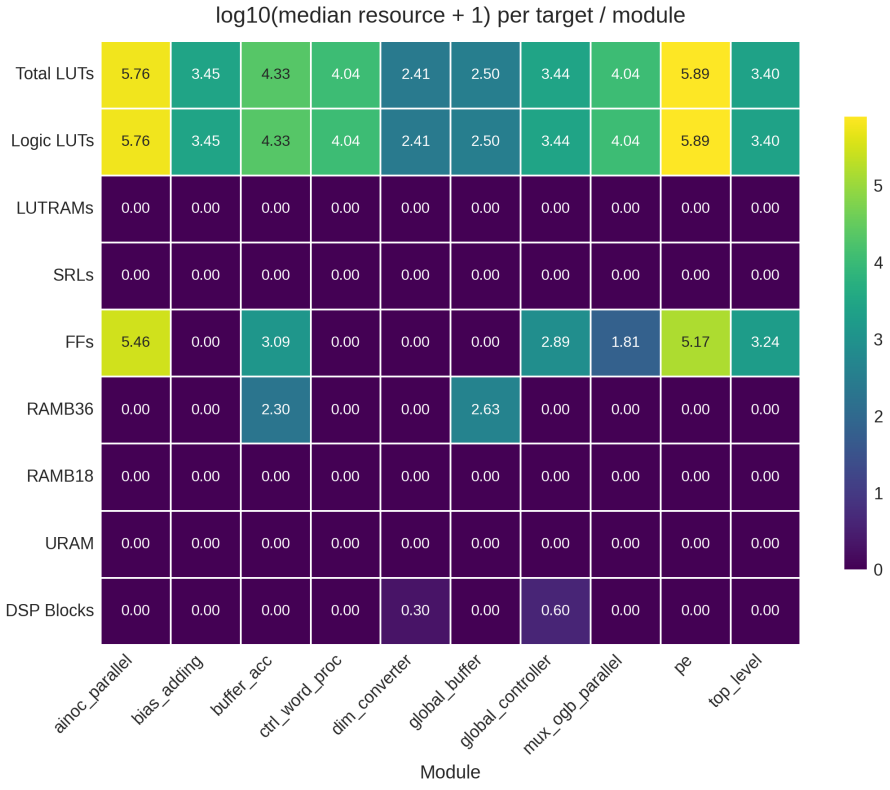


Figure 5.7: Median resource utilization per module and resource type ($\log_{10}(\text{median} + 1)$ scale). Logic resources concentrate in pe and ainoc_parallel, while memory primitives localize in buffer modules.

Two dominant trends emerge from this analysis. First, logic resources are concentrated in the compute fabric and NoC datapaths: pe and ainoc_parallel reach median log scaled values approaching 5.9 and 5.8 respectively, corresponding to several hundred thousand LUTs and FFs per module. This confirms that the majority of the area footprint scales with spatial replication governed by GRID, and that accurate prediction of these modules is critical for reliable area estimation. Second, memory and specialized resources exhibit strong localization. BRAM36 usage is almost exclusively confined to buffer_acc and global_buffer modules, while BRAM18 and URAM remain unused in all configurations.

DSP blocks are sparse, with only two modules showing non-zero median usage, suggesting opportunistic synthesis tool mapping rather than systematic DSP-based datapath construction. This distribution establishes a structural baseline: prediction errors on pe and ainoc_parallel will dominate total area error, while errors on smaller control modules have limited impact.

5.5.1.3 Feature-Target Correlation Analysis

Having identified which features are relevant, we now evaluate the intrinsic predictive information contained within the feature set. This analysis determines theoretical upper limits on attainable linear prediction performance prior to model training by employing Pearson correlation analysis.

Figure 5.8 reports Pearson correlation coefficients between architectural parameters and resource targets. The main observation is that linear effects are present but limited: the largest positive correlation coefficients remain below $r \approx 0.27$, indicating that while linear relationships exist, they explain only a fraction of the variance.

These observations motivate the adoption of non-linear regression techniques. Among the considered features, GRID exhibits the strongest correlation with logic-related targets (Total LUTs, Logic LUTs, FFs), which is expected given that it directly encodes the PE array size. OUTPUT_WIDTH and IFMAPS_WIDTH display similar but slightly lower correlation coefficients ($r \approx [0.22, 0.23]$), indicating a degree of redundancy with GRID.

In contrast, STRIDE and PADDING maintain correlations near zero for all logic-related targets ($|r| \leq 0.05$), corroborating that these parameters predominantly influence control logic and addressing, rather than the structural resource footprint. The memory-related targets exhibit distinct behavior: BRAM36 demonstrates only weak correlation with the scaling variables ($r \approx [0.16, 0.19]$), which aligns with the observation that BRAM instantiation is largely governed by the underlying buffer template. DSP utilization is effectively uncorrelated with all examined features, consistent with its sparse and nearly constant usage across configurations.

Collectively, these results suggest that area prediction is primarily governed by a small subset of

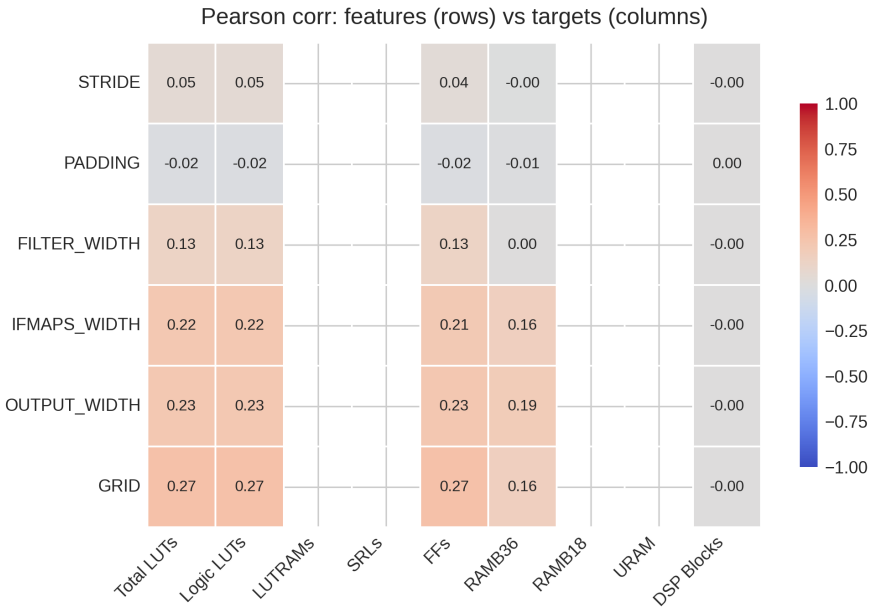


Figure 5.8: Pearson correlation between input features (rows) and resource targets (columns). GRID shows the strongest correlation with logic targets ($r \approx 0.27$), while STRIDE and PADDING have negligible impact.

structural variables, whereas secondary parameters provide only a limited linear contribution.

5.5.1.4 Model Performance and Interpretation

We now evaluate prediction accuracy per module and per resource target. For each combination, we train multiple regression algorithms (Linear, RandomForest, ExtraTrees, GradientBoosting) with cross-validation and select the best performer based on held-out error.

Figure 5.9 and Figure 5.10 report test R^2 and NMAE for each module target pair. We interpret these metrics jointly: R^2 measures whether the model captures configuration dependent trends, while NMAE quantifies absolute prediction error.

For modules that dominate the overall FPGA resource footprint, prediction accuracy is consistently high. The pe module attains an NMAE of approximately 3.7% and $R^2 \approx 1.00$ for total LUT utilization, and an NMAE of approximately 1.2% with $R^2 \approx 1.00$ for FF utilization. The ainoc_parallel module achieves an NMAE of approximately 0.1% and $R^2 = 1.00$ for both LUTs and FFs. These results indicate that the proposed models accurately capture the dominant scaling behavior induced by the GRID parameter.

For memory and DSP resources, several module–target combinations are structurally constant or nearly constant. For instance, BRAM36 is instantiated exclusively within buffer modules and is predicted with perfect accuracy ($R^2 = 1.00$, NMAE $\approx 0.0\%$). DSP blocks are correspondingly

identified as constant predictors where structurally appropriate. This behavior is desirable, as it prevents overfitting to noise in sparsely varying targets and is consistent with the regularity imposed by the architectural templates.

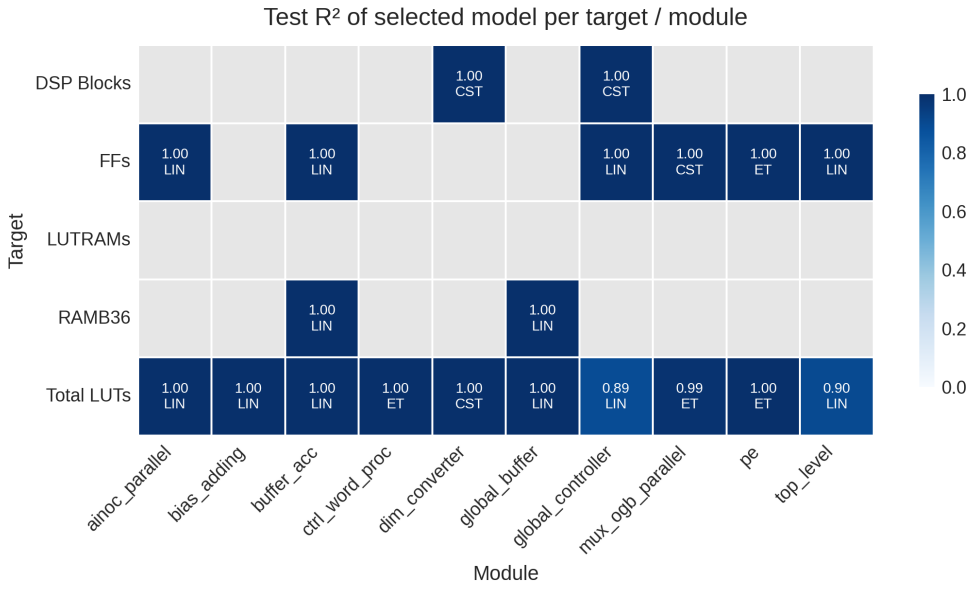


Figure 5.9: Test R^2 for each module target pair. Text shows R^2 value and selected algorithm (ET: ExtraTrees, RF: RandomForest, LIN: Linear, CST: Constant).

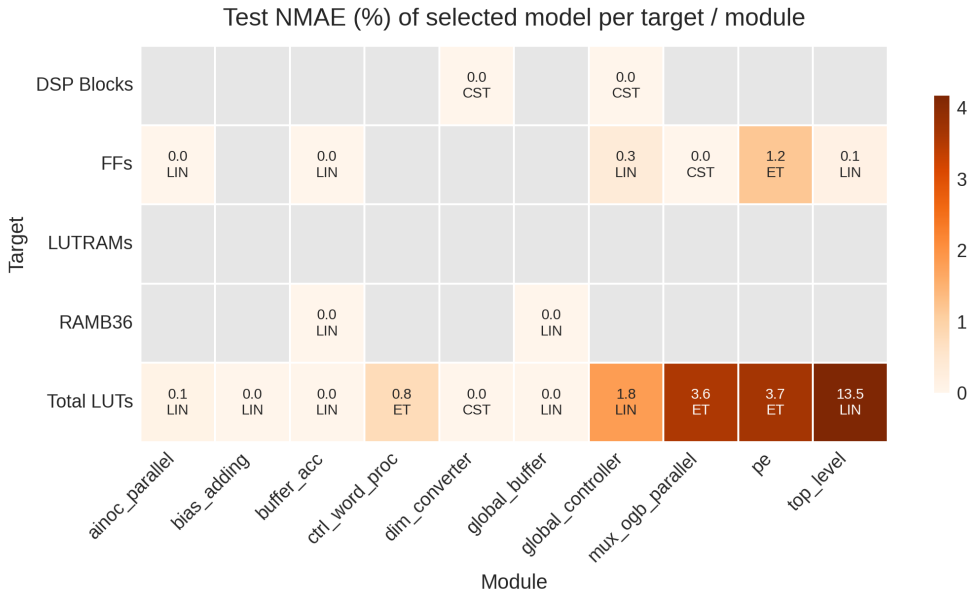


Figure 5.10: Test NMAE (%) for each module target pair and selected algorithm (ET: ExtraTrees, RF: RandomForest, LIN: Linear, CST: Constant).

Larger relative errors are concentrated in small, control-oriented modules. The logic glue in the top_level module exhibits an NMAE of approximately 13.5% with $R^2 \approx 0.90$, indicating that the model captures the underlying trend but fails to fully account for absolute offsets. A similar, though less pronounced, degradation is observed for the global_controller module (NMAE $\approx 1.8\%$, $R^2 \approx 0.89$). In these modules, resource utilization is driven primarily by corner-case control logic rather than by PE-grid replication, and their absolute contribution to the total area is comparatively minor.

In summary, the area model provides high fidelity for the resource-dominant modules, while residual inaccuracies are restricted to secondary control blocks with negligible impact on the global area estimate. Consequently, the model is well suited for use as a fast surrogate for area during architectural design space exploration and candidate ranking.

5.5.1.5 Prediction Speed and Practical Utility

To quantify the practical benefit of ML-based area estimation, we compare prediction time against full FPGA synthesis. Table 5.6 reports representative timings for each evaluation method.

Method	**Time**	**Speedup**
Vivado synthesis (1 config)	$\sim$4h	1$\times$
Model training (50 tasks, one time)	2m 30s	n/a
Model inference (1 config, all modules)	<100ms	$\sim 10^5\times$

Table 5.6: Time comparison between FPGA synthesis and ML-based area prediction.

The synthesis dataset required approximately 30 days to generate 300 configurations, running two parallel Vivado jobs with an average synthesis time of 4 hours per configuration. Once trained, the ML-based models predict resource utilization for all 10 modules and 5 resource types in under 100 milliseconds per candidate architecture, representing a speedup of approximately five orders of magnitude. The training phase completes in 2 minutes 30 seconds for all 50 module target combinations, with 34 tasks corresponding to structurally constant targets that are automatically identified.

This acceleration expands the scope of feasible design exploration: where synthesis-based evaluation limits exploration to tens of candidates, the ML-based surrogate enables sweeps over larger configuration spaces in minutes. Combined with the prediction accuracy reported in Figure 5.9 and Figure 5.10, the area model can serve as a surrogate for early stage architectural exploration.

As with any ML-based approach, prediction reliability depends on the training distribution. The models are trained on GRID values ranging from approximately 10 to 200, and extrapolation beyond

this range may yield less accurate estimates. For configurations far outside the training envelope, selective synthesis runs can be used to validate predictions or extend the model coverage.

5.5.2 Latency Model Evaluation

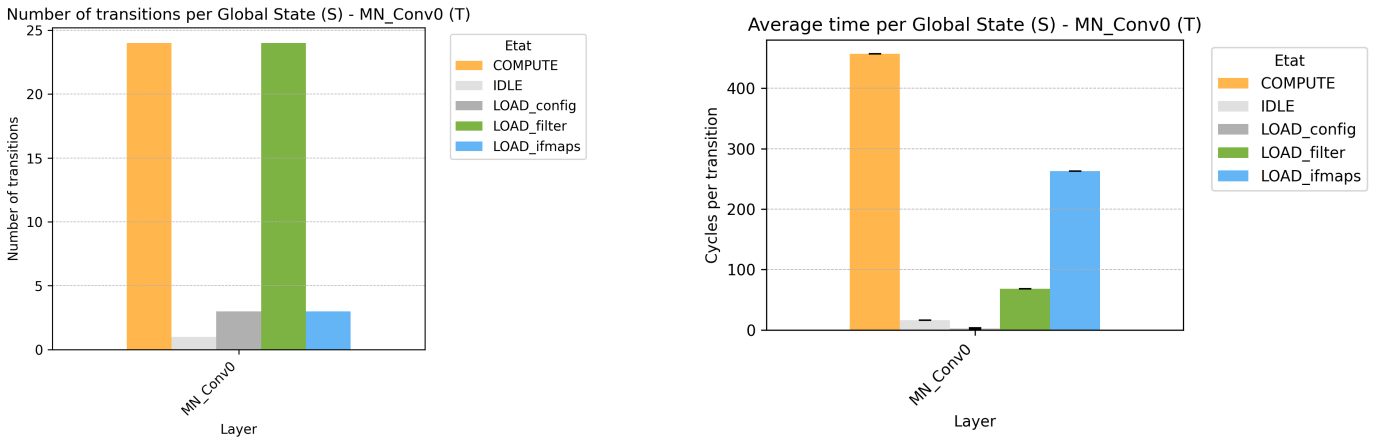
In contrast to area, which is determined by the static structural characteristics of the design, latency is governed by dynamic execution behavior. This dependence on run time activity introduces additional modeling complexities, as execution time is a function of the interaction among layer parameters, dataflow patterns, and controller state machines.

We assess the latency models through a structured analysis. First, we profile execution states on MN_Conv0 to illustrate how latency decomposes across FSM states and to motivate our dual-target prediction approach. Then, we focus on the PW_COMPUTE bucket (pointwise convolution, compute state) as an illustrative example to examine target distributions and feature correlations in detail. We validate the resulting predictions against ground truth measurements obtained from MobileNet v1, and conclude by quantifying the speedup relative to RTL simulation.

5.5.2.1 Convolution Execution State Profiling

To characterize the sources of execution latency, we profile representative layers by analyzing both the frequency and the duration of FSM states. This profiling identifies which states dominate overall latency and thereby guides the selection of prediction targets. We illustrate this analysis on the MN_Conv0 layer; the same methodology applies to all layer types, with results varying in magnitude but following similar structural patterns.

Figure 5.11 shows the execution profile of MN_Conv0, reporting both the number of transitions per state and their corresponding average duration. Two principal sources of latency can be identified. The COMPUTE and LOAD_filter states exhibit the highest transition counts, each with 24 transitions. Among these, the COMPUTE state dominates overall latency, as each transition incurs approximately 450 cycles, making it the primary performance bottleneck. The LOAD_filter state, with an average of about 80 cycles per transition but also occurring 24 times, emerges as the second-largest contributor to total latency. By contrast, the LOAD_ifmaps state requires a higher number of cycles per transition (260) but is activated only 3 times, resulting in a comparatively smaller impact on the total execution time.



(a) Number of transitions per state

(b) Average latency per transition (cycles)

Figure 5.11: Execution profile for MN_Conv0: transition counts (left) and average latency (right). S denotes the controller state, T the layer type.

This analysis reveals that total layer latency arises from the joint effect of transition frequency and per transition cost. The relative contribution of each state varies significantly across layer types. For MN_Conv0, the COMPUTE state dominates total latency. In contrast, pointwise layers such as PW1 are dominated by LOAD_filter, which accounts for over 70% of total cycles due to the large number of filter weights that must be loaded. This variability motivates training separate models per layer state bucket rather than a single global model.

5.5.2.2 Pointwise Target Distribution and Implications for Learning

Prior to constructing predictive models, we analyze the statistical characteristics of the two latency-related target variables. We illustrate the key phenomena on the PW_COMPUTE bucket (pointwise convolution, compute state) as a concrete example; similar patterns are observed across other layer state combinations, though with varying magnitudes.

Figure 5.12 shows the empirical distributions of Count and Mean Cycles across the 300 synthetic layer configurations for PW_COMPUTE. The Count variable exhibits pronounced right skew: approximately 130 observations fall within the interval 0 to 650,000, whereas a sparse tail extends up to 350,000. This heavy tailed behavior, characterized by mean $\gg$ median, introduces a regression trade-off: models that fit the dense region accurately will tend to incur large errors on the extreme values, whereas models adjusted to reduce error on these outliers will overestimate the more frequent instances.

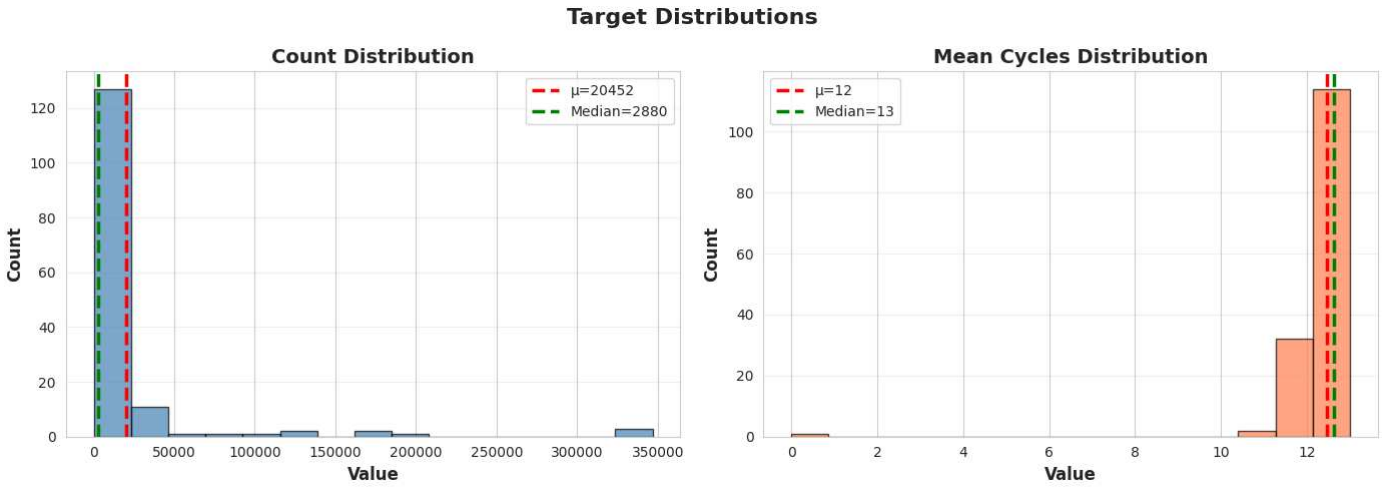


Figure 5.12: Empirical target distributions for PW_COMPUTE: (a) Count is strongly right skewed (mean$= 20{,}452$, median$= 2{,}880$), with outliers spanning roughly three orders of magnitude; (b) Mean Cycles is tightly concentrated around 12-13 cycles, with notably limited dispersion.

In contrast, Mean Cycles displays a highly concentrated distribution, with the vast majority of observations lying between 12 and 13 cycles and exhibiting extremely low variance. This near degenerate distribution substantially simplifies prediction but simultaneously reduces the information content available for discriminating among configurations: a trivial baseline that always predicts 12 cycles already attains a mean absolute error (MAE) of approximately 1 cycle.

These contrasting distributional properties suggest that Count is intrinsically challenging to predict from static features due to its skewness and heavy tail, whereas Mean Cycles behaves almost deterministically within this configuration space.

5.5.2.3 Pointwise Feature–Target Relationship Analysis

We assess the predictive information contained in the input features by means of correlation analysis, continuing with the PW_COMPUTE bucket. Figure 5.13 presents the Pearson correlation matrix between the features and the latency targets, while Figure 5.14 provides complementary visualizations via feature space projections colored by target values.

For Mean Cycles, the feature Total MACs attains a correlation of $r = 0.42$, indicating that execution latency increases with computational workload. This moderate correlation, in conjunction with the relatively concentrated distribution of the target, yields a sufficiently strong signal to enable accurate prediction.

In contrast, for Count, the maximum correlation is only $r = 0.39$ for Output Height (Y), corresponding to at most $r^2 = 0.15$, i.e., 15% of the variance explained. The remaining 85% of the variance is

attributable to factors that are not represented in the static layer specification, such as data access patterns, tiling strategies, and memory system behavior.

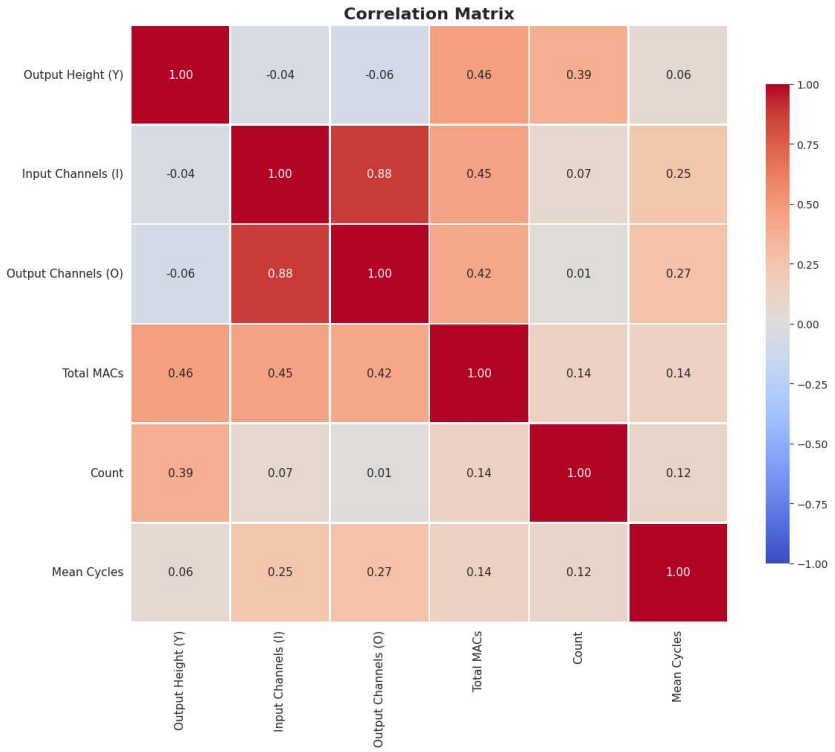
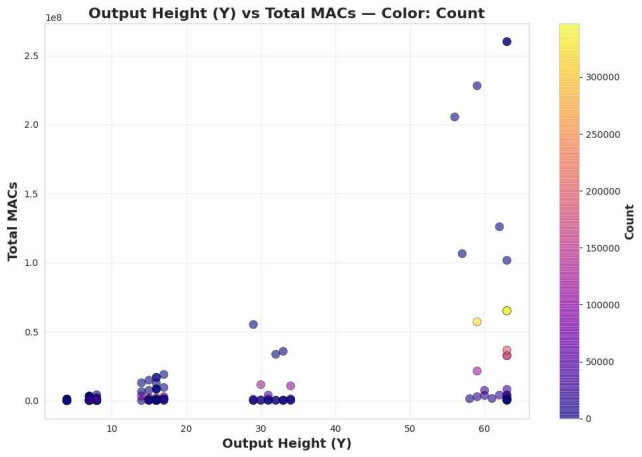
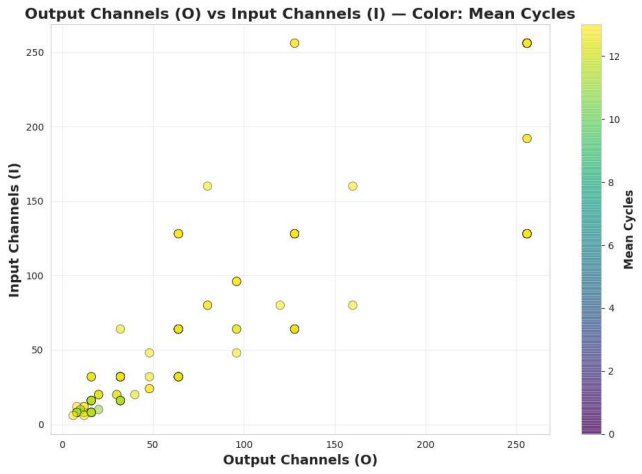


Figure 5.13: Pearson correlation matrix for PW_COMPUTE. Notable values: Total Macs shows moderate correlation with Mean Cycles ($r = 0.42$), while Count correlations remain weak ($r \leq 0.39$).



(a) Count: scattered color distribution reveals weak predictive signal



(b) Mean Cycles: consistent gradient indicates strong feature target coupling

Figure 5.14: Feature space projections colored by target values for PW_COMPUTE.

Moreover, a strong correlation is observed between Input Channel (I) and Output Channel (O) ($r = 0.88$), indicating substantial feature redundancy. Consequently, feature selection should prioritize Total MACs, which exhibits the strongest association with both latency targets.

The scatter plots in Figure 5.14 visually confirm these observations. For count (Figure 5.14a), the color distribution appears scattered across the feature space: points with similar coordinates exhibit widely varying count values, reflecting the weak predictive signal identified above. In contrast, the Mean Cycles plot (Figure 5.14b) displays a consistent color gradient along the feature axes, indicating that this target varies smoothly with input features and is therefore more amenable to regression.

Overall, these findings suggest that prediction of count is fundamentally limited by the present feature set, whereas prediction of Mean Cycles is comparatively well supported.

5.5.2.4 Overall Model Performance and Generalization

Building upon the preceding analysis, we trained separate ML-based models for each combination of layer type (Conv, DW, PW) and FSM state (COMPUTE, LOAD_filter, LOAD_ifmaps, LOAD_config, IDLE). Each model predicts two targets: (i) the number of entries into that state during layer execution, and (ii) the average number of cycles spent per entry. The training procedure utilizes cross-validation with hyperparameter optimization across multiple algorithms (Linear, RandomForest, ExtraTrees, GradientBoosting), with MAE as the primary evaluation metric.

To evaluate generalization, we apply these models to MobileNet v1 layers that were held out from training. For each layer, predictions are generated for all FSM states, and errors are computed by comparing against RTL simulation ground truth. Table 5.7 summarizes these results for a representative subset of layers; each row corresponds to one layer, and the reported MAE and MAPE values are averaged across all FSM states for that layer.

For cycles per entry, prediction accuracy is consistently high across all layer types: MAPE remains below 6% and MAE under 1 cycle for most layers, confirming that this target is inherently predictable due to its narrow distribution.

Prediction of state entries varies strongly by layer type. Depthwise convolutions achieve exact predictions (MAE = 0), indicating fully deterministic control flow where the number of state transitions is entirely determined by layer parameters. Pointwise convolutions exhibit larger errors (average MAE = 1256, MAPE = 10.34% across all states). This elevated average suggests that the limited predictive signal identified on PW_COMPUTE extends to other FSM states as well.

Layer	state entries		cycles per entry	
	MAE	MAPE (%)	MAE	MAPE (%)
MN_Conv0	0	1.67	2.87	3.91
MN_DW0	0	0.00	0.07	2.86
MN_DW1	0	0.00	30.47	4.65
MN_DW2	0	0.00	0.07	2.86
MN_PW0	459	17.37	0.13	3.43
MN_PW1	316	7.47	0.13	3.43
MN_PW2	490	7.60	0.19	3.94
MN_PW5	2262	14.78	0.19	3.94
MN_PW6	2917	9.51	0.10	3.13
DW avg.	0	0.00	8.64	5.46
PW avg.	1256	10.34	0.15	3.60

Table 5.7: Prediction errors on held-out MobileNet v1 layers (representative subset). For each layer, MAE and MAPE are averaged over all FSM states (COMPUTE, LOAD_filter, LOAD_ifmaps, LOAD_config, IDLE).

Despite these differences, the combined latency estimate (state entries $\times$ cycles per entry) achieves strong overall performance. Figure 5.15 shows predictions tracking ground truth across all layers on a logarithmic scale, with Spearman rank correlation $\rho = 0.986$ ($p < 0.01$).

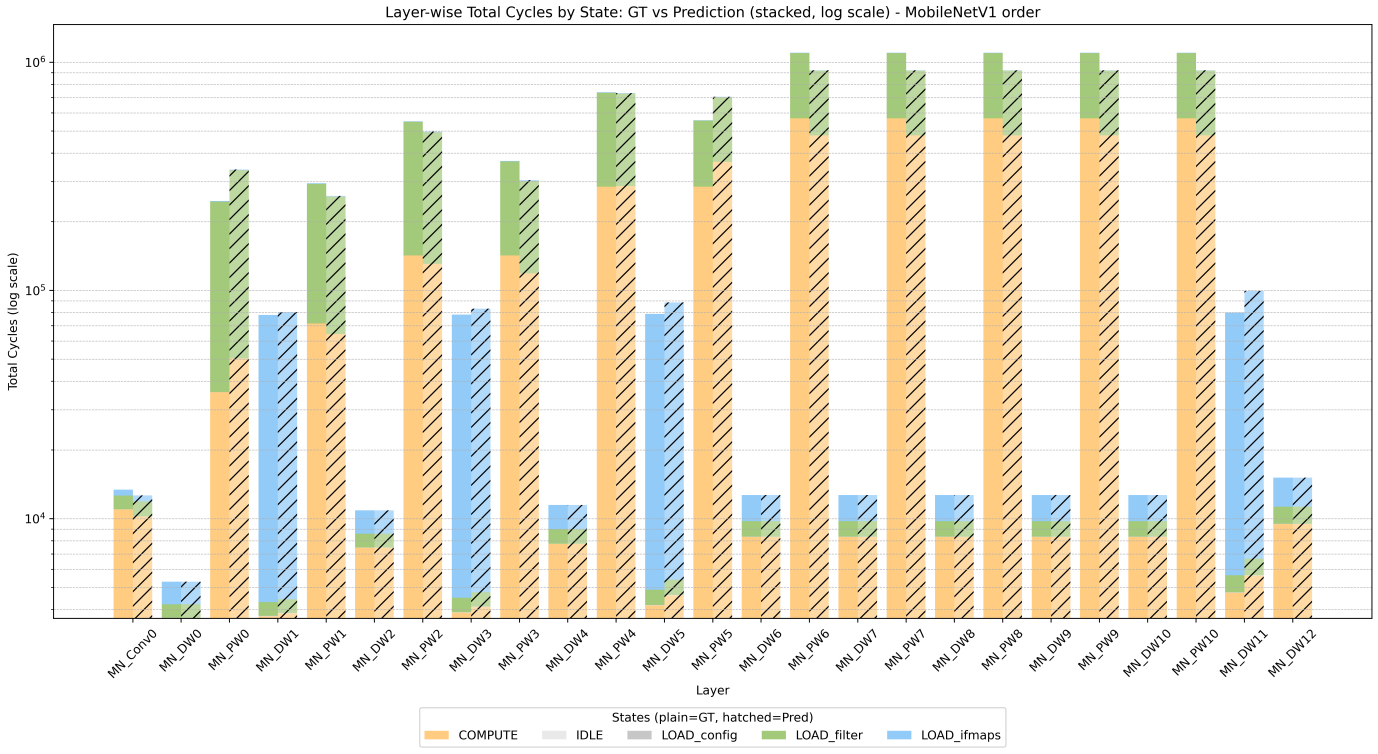


Figure 5.15: Layer wise total cycles comparison: ground truth (solid) vs. predictions (hatched). Log scale reveals strong agreement across five orders of magnitude. Spearman rank correlation: $\rho = 0.986$ ($p < 0.01$).

This near perfect rank preservation means the model correctly identifies which layers and states are most expensive, even when individual state entry predictions contain errors.

In summary the model's strength lies in predicting relative costs rather than absolute values: it correctly identifies that later PW layers consume more cycles than earlier ones, that COMPUTE states dominate over LOAD operations, and that specific architectural parameters drive systematic latency differences. For early stage exploration where ranking candidates is more critical than precise cycle counts, this rank preserving property makes the model practically useful despite individual prediction errors.

5.5.2.5 Prediction Speed and Accuracy Trade-off

To quantify the practical benefit of the ML-based latency model, we compare prediction time against both RTL simulation and analytical mapping across representative layers. Table 5.8 reports wall-clock execution times alongside prediction errors measured as MAPE on total cycles.

The speedup factors are substantial: prediction is approximately 120$\times$ faster than RTL simulation for Conv0, 760$\times$ for DW0, and 1400$\times$ for PW1. These gains arise because the ML-based model bypasses cycle accurate simulation entirely, requiring only feature extraction and model inference. The cold start overhead observed on the first prediction (due to model loading) is amortized when evaluating multiple layer configurations in sequence.

Layer	RTL sim.	Prediction	MAPE (%)
Conv0	2m 23s	1.2s	5.0
DW0	41s	54ms	2.9
PW1	21m	0.9s	9.8

Table 5.8: Execution time comparison across evaluation methods. Prediction achieves two to three orders of magnitude speedup over RTL simulation while maintaining single-digit MAPE on total cycle estimates.

Prediction accuracy varies by layer type, consistent with the distributional analysis presented earlier. Depthwise layers achieve the lowest error (2.9% MAPE), reflecting their deterministic control flow where state entry counts are exactly predictable. Pointwise layers exhibit higher error (9.8% MAPE), attributable to the weak feature–target correlation identified in the PW_COMPUTE analysis. The standard convolution (Conv0) falls between these extremes at 5.0% MAPE.

These results demonstrate a favorable accuracy speed trade-off for early stage design exploration. While RTL simulation remains necessary for final validation, the ML-based model enables rapid screening of candidate configurations with errors bounded below 10%. Combined with the strong rank correlation ($\rho = 0.986$) reported in Figure 5.15, This level of performance enables a rigorous

and computationally efficient exploration of the design space prior to conducting computationally expensive cycle-accurate evaluations.

5.5.3 Power Characterization and Baseline Model

Fast and accurate power estimation is essential for evaluating accelerator efficiency. In the context of learning-based PPA modeling, power data must exhibit sufficient variance to establish meaningful correlations with workload characteristics. This section describes the experimental dataset, reports the results obtained through RTL-based power estimation, analyzes the module-level power distribution, and discusses the pipeline limitations that currently prevent predictive modeling.

5.5.3.1 Dataset and Methodology

We orchestrated an experimental dataset of 100 convolutional layer configurations spanning representative parameter ranges: input feature map sizes from 8×8 to 126×126, input channels from 3 to 32, output channels from 6 to 64, with kernel size fixed at 3×3 and stride at 2. The dataset was restricted to standard convolutions due to the computational and storage costs of accurate power simulation.

In particular generating complete Value Change Dump (VCD) files through QuestaSim requires between 1 and 10 GB of disk storage per convolution layer, and up to 20 GB for pointwise operations. Given these constraints, we prioritized validating the simulation to estimation pipeline on convolutions before extending to other layer types.

The power estimation pipeline proceeds as follows: cycle accurate RTL simulation with full VCD dump, conversion to Switching Activity Interchange Format (SAIF), and post synthesis power analysis using Vivado's report_power command (as shown in Figure 4.12).

5.5.3.2 Results and Observed Limitations

We evaluated the pipeline on configurations with substantially different computational requirements, ranging from small layers (32×32 inputs, 6 channels) to large ones (100×100 inputs, 32 channels). Despite this variation, the dynamic power reported by Vivado remains nearly constant across all configurations, with less than 1% difference between extremes (as shown in Table 5.9).

Layer	Input Size	Channels (in→out)	Dynamic Power (W)
conv2	32×32	$6 \rightarrow 16$	3.146
conv3	56×56	$9 \rightarrow 12$	3.152
conv5	100×100	$32 \rightarrow 10$	3.166

Table 5.9: Power estimation results across layers of varying complexity. Dynamic power shows less than 1% variation despite order-of magnitude differences in computational workload.

The cause of this behavior lies in the net matching statistics between pre synthesis RTL-level SAIF and post synthesis netlist. Only 1,616 out of 1,365,749 nets (0.12%) were successfully reconciled. When matching fails, Vivado falls back to default toggle rate assumptions (typically 12.5%), applied uniformly regardless of actual switching activity [99]. The absence of variance in power estimates renders the data unsuitable for supervised learning: the target variable carries no information about input features, making it impossible to build a meaningful ML-based model with the current data.

5.5.3.3 Module-Level Power Distribution

Although workload dependent variation is absent in the current setup, the relative distribution of the reported dynamic power across architectural components remains informative for identifying where optimization efforts should focus. On-chip memory accounts for approximately 40% of the dynamic power (1.22 W), while the remainder is attributed to CLB logic (0.64 W, 20%), clock distribution (0.62 W, 20%), and signal routing (0.55 W, 20%). This breakdown suggests that, meaningful energy efficiency gains require system-level optimization beyond memory access reduction alone.

Despite the limitations for predictive modeling, these measurements provide a coarse baseline for energy efficiency assessment of the current FPGA prototype. With the architecture operating at 100 MHz, consuming 2.9 W and achieving 25.9 GOPS during MobileNet v1 inference, the resulting average efficiency is approximately 9 GOP/J (equivalently 0.11 nJ/MAC). This estimate should be interpreted cautiously, as it relies on tool-reported power under the current activity trace reconciliation limitations; nonetheless, it provides a useful reference point for the present implementation at 100 MHz.

5.5.3.4 Pipeline Limitations and Future Directions

At this stage, the power pipeline cannot yet produce workload sensitive labels suitable for predictive modeling. The main issue is switching activity reconciliation: Vivado cannot reliably map activity extracted from RTL simulation to the post synthesis netlist, whose signals are renamed and structurally

transformed by synthesis. When reconciliation fails, Vivado applies default toggle-rate assumptions, which largely removes workload dependent variation from the reported dynamic power.

We attempted to mitigate this mismatch by constraining synthesis transformations with `dont_touch` style directives, but this had only a limited effect on the reconciliation rate and did not restore meaningful workload dependent variance. In addition, the tool does not provide sufficient visibility to diagnose which portions of the activity successfully match and which do not, which makes iterative debugging of the reconciliation process difficult.

The most direct next step is therefore to obtain switching activity in a form that matches the analyzed netlist, by relying on simulation flows aligned with the synthesized design. Once activity to netlist alignment is reliable, the same supervised learning methodology used for latency and area can be revisited for power.

5.6 Discussion and Scope Boundaries

This section synthesizes the experimental results and methodological contributions of the PANDA framework, articulating how its design addresses fundamental challenges in DNN accelerator design space exploration for NoC-based dataflow architectures. We examine the framework's positioning within the accuracy speed trade-off that constrains exploration at scale, analyze decomposition as an organizing principle across area, latency, and power modeling, discuss the abstraction gap between high-level mapping specifications and implementation-level behavior, and conclude with a critical assessment of validated contributions and scope boundaries.

5.6.1 Addressing the Accuracy-Speed Gap

The core contribution of PANDA lies in its hybrid approach to the evaluator throughput-fidelity trade-off that constrains design space exploration. Analytical models enable rapid candidate evaluation but can underestimate communication and implementation effects in distributed architectures. This limitation becomes particularly acute for NoC-based designs, where arbitration, congestion, routing delays, and protocol overhead depend jointly on mapping decisions and network parameters in ways that are difficult to capture with compact closed-form models. In contrast, RTL simulation and synthesis capture these phenomena with high fidelity but impose evaluation costs that prohibit exploration at scale. Our measurements indicate synthesis times ranging from minutes to multiple hours per configuration depending on architectural complexity, while cycle accurate simulation exhibits similar variability driven by workload size and instrumentation depth.

PANDA addresses this gap by decoupling upfront data generation cost from runtime evaluation speed. Rather than invoking simulation or synthesis during exploration, the framework amortizes these costs by generating training datasets offline, then deploying learned ML-based models as fast surrogates during iterative search. This strategy achieves evaluation speeds approaching analytical methods, typically completing inference in under a second per configuration, while preserving sensitivity to NoC contention, routing delays, and FPGA specific resource constraints that analytical abstractions typically omit.

The investment in automated dataset generation is critical to enabling this approach. For the MobileNet v1 architectural family, dataset generation required on the order of hundreds of simulation and synthesis runs, spanning several days of elapsed time on a multi core workstation. Once trained, however, the resulting models support hundreds to thousands of evaluations per minute, enabling exploration configurations that would be entirely impractical with simulation-based backends. This amortization strategy becomes increasingly favorable as exploration scope expands: the more candidates that must be evaluated, the greater the relative advantage of paying high upfront costs to enable low marginal costs during search.

Analytical DSE toolflows such as Timeloop+Accelergy, MAESTRO, and ZigZag provide strong abstractions for loop nest mapping and memory-hierarchy traffic estimation. PANDA is best viewed as complementary: it targets the configuration in which implementation-level communication effects and toolflow dependent phenomena materially influence outcomes, particularly in NoC-based dataflow architectures. Instead of replacing analytical reasoning, PANDA leverages high-fidelity simulation and synthesis offline to learn surrogates that empirically capture the residual effects that are difficult to model analytically (e.g., contention and arbitration patterns). A direct quantitative benchmark against these frameworks on aligned architecture representations is not performed in this thesis and remains an important direction for future work; nevertheless, the results here demonstrate that implementation-calibrated surrogates can preserve candidate ranking for latency while operating at exploration compatible speed.

This approach differs from end-to-end predictors such as AIRCHITECT [103], which directly predict optimal configurations from workload characteristics. PANDA instead serves as an evaluation backend within an iterative DSE loop, providing per candidate PPA estimates that guide search algorithms toward promising regions without prescribing a specific exploration strategy. This design preserves flexibility in how search is conducted, whether through evolutionary algorithms [104, 105], Bayesian optimization, or reinforcement learning [106], while ensuring that evaluations remain sensitive to implementation-level phenomena that can dominate NoC-based designs.

5.6.2 Decomposition as Organizing Principle

A defining characteristic of PANDA is its systematic use of decomposition across multiple dimensions: by layer type, by architectural module, and by execution state. This organizational principle distinguishes the approach from monolithic end-to-end predictors and yields consistent benefits across PPA estimation workflows.

Layer type stratification avoids forcing ML-based models to learn discontinuous conditional behaviors across standard, depthwise, and pointwise convolutions. Module-level decomposition for area enables component-level bottleneck identification, supports generalization through scaling (training on one PE count and predicting for others), and facilitates validation through hierarchical aggregation. State-level decomposition for latency decouples tiling dependent visitation counts from microarchitecture dependent timing, enabling models that are more interpretable and easier to validate while remaining sensitive to contention-induced delays.

Rather than modeling accelerator evaluation as a black box ML-based problem, PANDA constructs surrogate models that mirror the physical structure of hardware and the temporal structure of execution. This alignment facilitates debugging and validation, while enabling generalization strategies such as module scaling that would be opaque in monolithic predictors. The trade-off is coordination complexity: per module and per state predictions must aggregate consistently, and discretization artifacts can introduce small inconsistencies requiring validation.

5.6.3 The Mapping-PPA Abstraction Gap

A central tension in PANDA's design lies in the abstraction gap between two distinct representations of workload execution. The mapping model produces action counts: operational metrics quantifying MAC operations, memory accesses, and data transfers. The PPA ML-based models operate on controller-level observables: state visitation counts and mean cycles per visit as defined by the FSMs. While a semantic correspondence exists between these representations, the current framework does not explicitly reconcile them.

A conceptual mapping exists between action counts and FSM states. Memory operations correspond to loading states, MAC counts relate to computation states, and writeback transfers align with output storing states. However, establishing this correspondence precisely remains unimplemented. Action counts are aggregate quantities, while FSM execution is sequential and iterative: a mapping specifying a given number of loads translates to repeated state visits whose counts and durations depend on tiling, bank conflicts, and NoC congestion. The FSM also includes states with no direct action count equivalent (initialization, synchronization barriers, and idle cycles).

Bridging this gap would strengthen interpretability and may improve generalization. Two directions can be envisioned. First, data collection could operate at a finer granularity than FSM states, capturing sub state transitions that map more directly to individual operations, at increased instrumentation cost. Second, analytical equations could be derived to express state visitation patterns as functions of action counts and validated against RTL measurements.

5.6.4 Summary

PANDA demonstrates that hybrid evaluation strategies combining offline implementation-level data generation with runtime learned surrogates can narrow the accuracy speed gap for NoC-based DNN accelerator exploration. The framework's modular architecture, automated infrastructure, and systematic decomposition principles establish a foundation for scalable and interpretable latency/area modeling that preserves sensitivity to distributed communication effects while sustaining throughput comparable to analytical approaches. The scope boundaries identified throughout this work (workload sensitive power modeling and end-to-end DSE integration) define clear directions for future development and clarify the current validated contributions within their intended application domain.

Chapter 6

Conclusion and Perspectives

6.1 Conclusion

This thesis addressed the challenge of efficient performance, power, and area estimation for NoC-based CNN accelerators, a key bottleneck for design space exploration in embedded AI vision systems spanning sub watt to tens of watts operating ranges.

The analysis of existing estimation frameworks highlighted a methodological gap for distributed dataflow architectures. While analytical tools provide strong support for compute and memory hierarchy reasoning, they often reduce communication to bandwidth constraints, leaving contention, arbitration, protocol overhead, and implementation driven timing effects insufficiently captured. This gap motivated a hybrid approach that combines implementation-level data generation with machine learning surrogates.

The core contribution of this thesis is the PANDA framework, which provides scalable, implementation grounded estimation through three integrated components. First, the mapping model specifies CNN to hardware mappings through compact dimension vectors without requiring register transfer-level descriptions, and generates action count reports quantifying MAC operations, accumulations, and memory accesses. This abstraction remains consistent with cycle accurate RTL behavior on representative workloads. Second, an automated data generation pipeline addresses the limited data availability inherent to supervised learning for hardware estimation: a multi stimulus generator produces testbenches and memory initialization files for arbitrary CNN layer configurations, while orchestration scripts automate RTL simulation and FPGA synthesis, including instrumentation, trace handling, and resource extraction. Third, the learning methodology structures estimation models through layer type stratification, module-level decomposition for area, and state-level partitioning for latency. In particular, the dual head ML-based formulation for latency predicting state visitation

counts and mean cycles per visit separates control flow effects from per state timing effects and improves interpretability relative to monolithic predictors.

Experimental validation on MobileNet v1 deployed on a processing element grid targeting the Xilinx Virtex UltraScale+ FPGA demonstrated the framework's effectiveness. Latency prediction achieved strong rank correlation (Spearman $\rho = 0.986$), indicating that the surrogates preserve the relative ordering of candidates that is essential for iterative exploration. Depthwise convolutions yielded near perfect predictions, while pointwise convolutions exhibited greater variability, consistent with their stronger sensitivity to communication and controller effects. Area estimation achieved robust accuracy across the resource dominant modules, enabling reliable early stage feasibility screening.

Beyond accuracy, PANDA enables exploration scale throughput by reducing evaluation cost by orders of magnitude. The mapping model produces layer-level access and compute indicators in seconds, whereas equivalent validation at RTL granularity requires tens of seconds to tens of minutes depending on layer type and instrumentation. Once trained, the learned surrogates replace expensive backends with millisecond second inference per candidate: for example, area inference is below 100 ms versus $\sim$4 h for FPGA synthesis in our flow, and latency inference is in the $\sim$50 ms-1 s range versus $\sim$40 s-21 min for RTL simulation on representative MobileNet v1 layers. These speedups (two to five orders of magnitude depending on the metric) shift evaluation from a limiting factor to an enabler for broad architectural and mapping sweeps.

Overall, the results validate that hybrid evaluation strategies combining offline implementation-level data generation with runtime learned surrogates can narrow the accuracy speed gap for NoC-based accelerator exploration, achieving near analytical evaluation speed while retaining sensitivity to implementation dependent effects such as contention, routing overhead, and FPGA toolflow constraints.

6.2 Perspectives

The scope boundaries identified during validation define natural directions for future work.

6.2.1 Power Estimation Refinement

Power estimation is a priority for future development. In the current setup, workload sensitive dynamic power could not be validated reliably due to limited net matching between pre synthesis activity traces and post synthesis power analysis. A practical next step is to obtain switching activity with

a flow that preserves net correspondence, e.g., post synthesis simulation (or post implementation simulation when feasible) combined with consistent stimulus loading. Once activity data reflects workload dependent variation, the same decomposition and ML-based methodology used for latency and area can be extended to power prediction.

6.2.2 Design Space Exploration Integration

PANDA provides evaluation backends that make automated exploration practical. A first integration protocol is to couple PANDA with large-scale sampling or guided search and validate only a small top-k subset using RTL simulation or synthesis. This directly measures the surrogate's utility as a ranking function under realistic budgets. More sophisticated strategies (Bayesian optimization, evolutionary search, reinforcement learning) can then leverage the low marginal evaluation cost to explore larger joint hardware-mapping spaces. We target three exploration axes: Hardware variations across architectural configurations, including topology changes, memory sizing, and operator choices. Dataflow variations adapted to each DNN and, when needed, to individual layers within a given DNN. Multiple DNNs evaluated on the same hardware architecture.

6.2.3 Generalization and Transferability

Extending the study beyond MobileNet v1 requires expanding the PPA dataset to cover a broader range of workloads, from CNNs to Transformer-based models up to LLMs. This expansion is particularly critical because execution behavior can vary both across layers and across different types of DNNs, as a consequence of differing operating modes, heterogeneous dataflows, and newly emerging dataflow variants.

The PANDA methodology is not tied to a single accelerator architecture. The same pipeline can be applied across diverse hardware families, ranging from hardwired systolic architectures to dataflow-centric designs and fully programmable accelerators. Supporting different target technologies, such FPGA prototyping and ASIC implementation, primarily requires adapting the data-generation infrastructure (measurement/simulation/synthesis) and retraining the surrogate models.

Finally, systematic benchmarking against analytical frameworks using aligned inputs and metrics would identify the conditions in which implementation-calibrated surrogates provide the strongest predictive advantage.

Bibliography

[1] P. Beckman, R. Sankaran, C. Catlett, N. Ferrier, R. Jacob, and M Papka. Waggle: An open sensor platform for edge computing. In *2016 IEEE Sensors*, pages 1–3, 2016. doi: 10.1109/ICSENS.2016.7808975.

[2] A Theuwissen. There's More to the Picture Than Meets the Eye and in the Future It Will Only Become More so. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 64, pages 30–35, 2021. doi: 10.1109/ISSCC42613.2021.9366058.

[3] W. Fabre, K. Haroun, V. Lorrain, M. Lepecq, and G Sicard. From Near-Sensor to In-Sensor: A State-of-the-Art Review of Embedded AI Vision Systems. *Sensors*, 24, 2024. doi: 10.3390/s24165446.

[4] D.G Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *Int. J. Comput. Vis.*, 60, 2004. doi: 10.1023/B:VISI.0000029664.99615.94.

[5] H. Bay, A. Ess, T. Tuytelaars, and L.V Gool. Speeded-Up Robust Features (SURF). *Comput. Vis. Image Underst.*, 110, 2008. doi: 10.1016/j.cviu.2007.09.014.

[6] N. Dalal and B Triggs. Histograms of oriented gradients for human detection. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 1, pages 886–893, 2005. doi: 10.1109/CVPR.2005.177.

[7] B.L. Deng, G. Li, S. Han, L. Shi, and Y Xie. Model Compression and Hardware Acceleration for Neural Networks: A Comprehensive Survey. *Proc. IEEE*, 108, 2020. doi: 10.1109/JPROC.2020.2976475.

[8] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S Emer. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. In *IEEE*, volume 105, pages 2295–2329, 2017. doi: 10.1109/JPROC.2017.2761740.

[9] M. Capra, B. Bussolino, A. Marchisio, M. Shafique, G. Masera, and M Martina. An Updated Survey of Efficient Hardware Architectures for Accelerating Deep Convolutional Neural Networks. *Future Internet*, 12, 2020. doi: 10.3390/fi12070113.

[10] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and F.-F Li. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009. doi: 10.1109/CVPR.2009.5206848.

[11] T.Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C.L. Zitnick, and P Dollár. Microsoft COCO: Common Objects in Context. In *European Conference on Computer Vision (ECCV)*, pages 740–755, 2015. doi: 10.1007/978-3-319-10602-1_48.

[12] A. Krizhevsky, I. Sutskever, and G.E Hinton. ImageNet Classification with Deep Convolutional Neural Networks, 2012.

[13] S. Minaee, Y. Boykov, F. Porikli, A. Plaza, N. Kehtarnavaz, and D Terzopoulos. Image Segmentation Using Deep Learning: A Survey. *IEEE Trans. Pattern Anal. Mach. Intell.*, 2021. doi: 10.1109/TPAMI.2021.3059968.

[14] S.S.A. Zaidi, M.S. Ansari, A. Aslam, N. Kanwal, M. Asghar, and B Lee. A Survey of Modern Deep Learning Based Object Detection Models. *Digital Signal Processing*, 126, 2021. doi: 10.1016/j.dsp.2022.103514.

[15] K. He, X. Zhang, S. Ren, and J Sun. Deep Residual Learning for Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. doi: 10.1109/CVPR.2016.90.

[16] I.O. Tolstikhin, N. Houlsby, A. Kolesnikov, L. Beyer, X. Zhai, T. Unterthiner, J. Yung, A. Steiner, D. Keysers, J. Uszkoreit, and et al. MLP-Mixer: An all-MLP Architecture for Vision. In *Advances in Neural Information Processing Systems*, pages 24261–24272, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/cba0a4ee5ccd02fda0fe3f9a3e7b89fe-Abstract.html.

[17] H. Touvron, P. Bojanowski, M. Caron, M. Cord, A. El-Nouby, E. Grave, G. Izacard, A. Joulin, G. Synnaeve, J. Verbeek, and et al. ResMLP: Feedforward networks for image classification with data-efficient training. *IEEE Trans. Pattern Anal. Mach. Intell.*, pages 5314–5321, 2022. doi: 10.1109/TPAMI.2022.3206148.

[18] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, and et al. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *International Conference on Learning Representations (ICLR)*, 2021. doi: https://doi.org/10.48550/arXiv.2010.11929.

[19] H. Touvron, M. Cord, M. Douze, F. Massa, A. Sablayrolles, and H Jégou. Training data-efficient image transformers & distillation through attention. In *38th International Conference on Machine Learning*, 2021.

[20] M Horowitz. 1.1 Computing's Energy Problem (and What We Can Do about It). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, 2014. doi: 10.1109/ISSCC.2014.6757323.

[21] M. Lepecq, T. Dalgaty, W. Fabre, and S Chevobbe. End-to-End Implementation of a Convolutional Neural Network on a 3D-Integrated Image Sensor with Macropixel Array. *Sensors*, 23, 2023. doi: 10.3390/s23041909.

[22] P. Liu, Z. Yang, L. Kang, and J Wang. A Heterogeneous Architecture for the Vision Processing Unit with a Hybrid Deep Neural Network Accelerator. *Micromachines*, 13, 2022. doi: 10.3390/mi13020268.

[23] X. Zheng, L. Cheng, M. Zhao, Q. Luo, H. Li, R. Dou, S. Yu, and N Wu. ViP: A Hierarchical Parallel Vision Processor for Hybrid Vision Chip. *IEEE Trans. Circuits Syst. II Express Briefs*, 69:2957–2961, 2022. doi: 10.1109/TCSII.2022.3156945.

[24] ADLINK Technology. NEON-1000-MDX Starter Kit Series Datasheet. 2020, 2020. URL https://www.adlinktech.com/Products/Machine_Vision/SmartCamera/NEON-1000-MDX_Series. Accessed on 3 April 2023.

[25] JeVois. Jevois pro: Smart embedded vision toolkit, 2021. URL http://jevois.org/doc/ProUserConnect.html. Accessed on 3 April 2023.

[26] ADLINK Technology. NEON-2000-JNX Series Datasheet. 2021, 2021. URL https://www.adlinktech.com/Products/Machine_Vision/SmartCamera/NEON-2000-JNX_Series. Accessed on 18 August 2024.

[27] Teledyne FLIR. Fireflydl datasheet, 2021. URL https://www.flir.com/support-center/iis/machine-vision/knowledge-base/technical-documentation-firefly-s/. Accessed on 3 April 2023.

[28] Luxonis. Oak-1 depthai hardware product page, 2021. URL https://shop.luxonis.com/products/oak-1?variant=42664380334303. Accessed on 3 April 2023.

[29] DFRobot. Huskylens AI Machine Vision Sensor., 2024. URL https://store-usa.arduino.cc/products/gravity-huskylens-ai-machine-vision-sensor. Accessed on 3 August 2024.

[30] G AIY. AIY Vision Kit v1.1. 2024, 2024. URL https://pi3g.com/products/machine-learning/google-coral/aiy-vision-kit/. Accessed on 3 August 2024.

[31] ImagoTechnologies. VisionAI v1.2 Datasheet. 2021, 2021. URL https://imago-technologies.com/wp-content/uploads/2021/01/Specification-VisionAI-V1.2.pdf. Accessed on 3 August 2023.

[32] Seeed Studio. SenseCAP A1101 LoRaWAN Vision AI Sensor User Guide. 2024, 2024. URL https://files.seeedstudio.com/wiki/SenseCAP-A1101/SenseCAP_A1101_LoRaWAN_Vision_AI_Sensor_User_Guide_V1.0.2.pdf. Accessed on 18 August 2024.

[33] JeVois. JeVois Smart Machine, 2021. URL https://www.jevoisinc.com/pages/hardware.

[34] Amazon Web Services. AWS DeepLens AI Vision. 2024, 2024. URL https://bloggeek.me/aws-deeplens-ai-vision/. Accessed on 4 August 2024.

[35] K. Bong, S. Choi, C. Kim, S. Kang, Y. Kim, and H.J Yoo. 14.6 a 0.62mW Ultra-Low-Power Convolutional-Neural-Network facial recognition Processor and a CIS Integrated with Always-on Haar-like Face Detector. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 248–249, 2017. doi: 10.1109/ISSCC.2017.7870354.

[36] D. Palossi, A. Loquercio, F. Conti, E. Flamand, D. Scaramuzza, and L Benini. A 64-mW DNN-based Visual Navigation Engine for Autonomous Nano-Drones. *IEEE Internet Things J.*, 6:8357–8371, 2019. doi: 10.1109/JIOT.2019.2917066.

[37] H. An, S. Schiferl, S. Venkatesan, T. Wesley, Q. Zhang, J. Wang, K.D. Choo, S. Liu, Z. Li, and L Gong. An Ultra-Low-Power Image Signal Processor for Hierarchical Image Recognition with Deep Neural Networks. *IEEE J. Solid-State Circuits*, pages 1071–1081, 2021. doi: 10.1109/JSSC.2020.3041858.

[38] K. Bong, S. Choi, C. Kim, D. Han, and H.-J Yoo. A Low-Power Convolutional Neural Network Face Recognition Processor and a CIS Integrated with Always-on Face Detector, 2018.

[39] A. Verdant, W. Guicquero, N. Royer, G. Moritz, T. Seifert, C.-H. Hsu, and Z He. A 3.0μW5fps QQVGA Self-Controlled Wake-up Imager with on-Chip Motion Detection, Auto-Exposure and Object Recognition. In *2020 IEEE Symposium on VLSI Circuits*, pages 1–2, 2020. doi: 10.1109/VLSICircuits18222.2020.9162854.

[40] R. Eki, S. Yamada, H. Ozawa, H. Kai, K. Okuike, H. Gowtham, H. Nakanishi, E. Almog, Y. Livne, G. Yuval, and et al. 9.6 a 1/2.3inch 12.3Mpixel with on-Chip 4.97TOPS/W CNN Processor Back-Illuminated Stacked CMOS Image Sensor. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 64, pages 154–156, 2021. doi: 10.1109/ISSCC42613.2021.9365965.

[41] M. Lefebvre, L. Moreau, R. Dekimpe, and D Bol. A 0.2-to-3.6TOPS/W Programmable Convolutional Imager SoC with in-Sensor Current-Domain Ternary-Weighted MAC Operations for Feature Extraction and Region-of-Interest Detection. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 64, pages 118–120, 2021. doi: 10.1109/ISSCC42613.2021.9365839.

[42] R. Singh, S. Bailey, P. Chang, A. Olyaei, M. Hekmat, and R Winoto. A 21pj/frame/pixel imager and 34pj/frame/pixel image processor for a low-vision augmented-reality smart contact lens. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 64, pages 482–484, 2021. doi: 10.1109/ISSCC42613.2021.9365771.

[43] L.C. Gontard, R. Carmona-Galan, and A Rodriguez-Vazquez. Cellular-Neural-Network Focal-Plane Processor as Pre-Processor for ConvNet Inference. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2020. doi: 10.1109/ISCAS45731.2020.9181102.

[44] L. Bose, P. Dudek, J. Chen, S. Carey, P. Everitt, H. Wang, P. Davies, J. Liu, Y. Wang, and A Uusitalo. A Camera That Cnns: Towards Embedded Neural Networks on Pixel Processor Arrays. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 1335–1344, 2019. doi: 10.1109/ICCV.2019.00142.

[45] M.Z. Wong, B. Guillard, R. Murai, S. Saeedi, and P.H.J Kelly. AnalogNet: Convolutional Neural Network Inference on Analog Focal Plane Sensor Processors, 2020. URL http://arxiv.org/abs/2006.01765. Accessed on 18 August 2024.

[46] I. Miro-Panades, V. Lorrain, L. Billod, I. Kucher, V. Templier, S. Choisnet, N. Ali, B. Rossigneux, O. Bichler, and A. Valentian. A 772µj/frame ImageNet feature extractor accelerator on HD

images at 30FPS. In *Asia Pacific Conference on Circuits and Systems (APCCAS)*, Taipei, Taiwan, 2024.

[47] Y. Lecun, L. Bottou, Y. Bengio, and P Haffner. Gradient-Based Learning Applied to Document Recognition. *Proc. IEEE*, 86, 1998. doi: 10.1109/5.726791.

[48] S. Bianco, R. Cadene, L. Celona, and P Napoletano. Benchmark Analysis of Representative Deep Neural Network Architectures. *IEEE Access*, 6:64270–64277, 2018. doi: 10.1109/ACCESS.2018.2877890.

[49] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A Rabinovich. Going Deeper with Convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015. doi: 10.1109/CVPR.2015.7298594.

[50] M. Tan and Q.V Le. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *36th International Conference on Machine Learning*, 2019.

[51] 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 <0.5 MB Model Size, 2017. URL https://openreview.net/forum?id=S1xh5sYgx.

[52] J. Hu, L. Shen, and G Sun. Squeeze-and-Excitation Networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7132–7141, 2018. doi: 10.1109/CVPR.2018.00745.

[53] A. Howard, M. Sandler, B. Chen, W. Wang, L.C. Chen, M. Tan, G. Chu, V. Vasudevan, Y. Zhu, and R Pang. Searching for MobileNetV3. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 1314–1324, 2019. doi: 10.1109/ICCV.2019.00140.

[54] A. Gholami, K. Kwon, B. Wu, Z. Tai, X. Yue, P. Jin, S. Zhao, and K Keutzer. SqueezeNext: Hardware-Aware Neural Network Design. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 1719–171909, 2018. doi: 10.1109/CVPRW.2018.00215.

[55] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q.V Le. MnasNet: Platform-Aware Neural Architecture Search for Mobile. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2815–2823, 2019. doi: 10.1109/CVPR.2019.00293.

[56] Google AI Blog. Introducing the Next Generation of On-Device Vision Models: MobileNetV3 and MobileNetEdgeTPU, 2019. URL https://ai.googleblog.com/2019/11/introducing-next-generation-on-device.html. Accessed on 3 April 2023.

[57] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.C Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4510–4520, 2019. doi: 10.1109/CVPR.2018.00474.

[58] 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, 2017. URL http://arxiv.org/abs/1704.04861. Accessed on 18 August 2024.

[59] P. Ramachandran, B. Zoph, and Q.V Le. Searching for Activation Functions. In *International Conference on Learning Representations (ICLR)*, 2017.

[60] F Chollet. Xception: Deep Learning with Depthwise Separable Convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1251–1258, 2017. doi: 10.1109/CVPR.2017.195.

[61] K. Simonyan and A Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition, 2015. URL http://arxiv.org/abs/1409.1556. Accessed on 18 August 2024.

[62] J.C Mitchell. MobileNetV2 Keras Implementation. 2018, 2018. URL https://github.com/JonathanCMitchell/mobilenet_v2_keras/blob/master/mobilenetv2.py. Accessed on 27 July 2024.

[63] C. White, M. Safari, R. Sukthanker, B. Ru, M. Lindauer, and F Hutter. Neural Architecture Search: Insights from 1000 Papers, 2023. URL https://arxiv.org/abs/2301.08727. Accessed on 18 August 2024.

[64] Keras Team. Keras Applications-MobileNet. 2023, 2023. URL https://keras.io/api/applications/mobilenet/. Accessed on 27 July 2024.

[65] A. Amid, K. Kwon, A. Gholami, B. Wu, K. Keutzer, and M.W Mahoney. Co-design of deep neural nets and neural net accelerators for embedded vision applications, 2019.

[66] TensorFlow Blog. Higher Accuracy on Vision Models with EfficientNet-Lite. 2020, 2020. URL https://blog.tensorflow.org/2020/03/higher-accuracy-on-vision-models-with-efficientnet-lite.html. Accessed on 3 April 2023.

[67] K Fukushima. Cognitron: A self-organizing multilayered neural network, 1975.

[68] P.K.A. Vasu, J. Gabriel, J. Zhu, O. Tuzel, and A Ranjan. MobileOne: An Improved One millisecond Mobile Backbone. In *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7907–7917, 2023. doi: 10.1109/CVPR52729.2023.00764.

[69] J. Cheng, J. Wu, C. Leng, Y. Wang, Q. Hu, and D Tao. Quantized CNN: A Unified Approach to Accelerate and Compress Convolutional Networks, 2018.

[70] R Krishnamoorthi. Quantizing Deep Convolutional Networks for Efficient Inference: A Whitepaper, 2018. URL http://arxiv.org/abs/1806.08342. Accessed on 18 August 2024.

[71] M. Nagel, M. Fournarakis, R.A. Amjad, Y. Bondarenko, and T Blankevoort. A White Paper on Neural Network Quantization, 2021. URL https://arxiv.org/abs/2106.08295.

[72] T. Liang, J. Glossner, L. Wang, S. Shi, and X Zhang. Pruning and Quantization for Deep Neural Network Acceleration: A Survey. *Neurocomputing*, 461:370–403, 2021. doi: 10.1016/j.neucom.2021.07.045.

[73] S. Han, J. Pool, J. Tran, and W.J Dally. Learning Both Weights and Connections for Efficient Neural Networks. In *28th International Conference on Neural Information Processing Systems (NIPS'15)*, pages 1135–1143, 2015. URL https://proceedings.neurips.cc/paper_files/paper/2015/file/ae0eb3eed39d2bcef4622b2499a05fe6-Paper.pdf.

[74] M. Courbariaux, Y. Bengio, and J.P David. BinaryConnect: Training Deep Neural Networks with Binary Weights during Propagations. In *28th International Conference on Neural Information Processing Systems (NIPS'15)*, pages 3123–3131, 2015. URL https://dl.acm.org/doi/10.5555/2969442.2969588.

[75] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision – ECCV 2016*, pages 525–542, Cham, 2016. Springer International Publishing. ISBN 978-3-319-46493-0.

[76] M Grootendorst. A Visual Guide to Quantization. 2023, 2023. URL https://newsletter.maartengrootendorst.com/p/a-visual-guide-to-quantization. Accessed on 4 August 2024.

[77] NVIDIA. Jetson AGX Xavier Developer Kit User Guide. NVIDIA Corporation, Online Resource, 201. URL https://developer.nvidia.com/embedded/downloads. Accessed on 20 August 2024.

[78] Intel. Intel Movidius Myriad X Vision Processing Unit (VPU). 2021, 2021. URL https://www.intel.com/content/www/us/en/products/details/processors/movidius-vpu/movidius-myriad-x.html. Accessed on 3 April 2023.

[79] A. Garofalo, M. Rusci, F. Conti, D. Rossi, and L Benini. PULP-NN: Accelerating Quantized Neural Networks on Parallel Ultra-Low-Power RISC-V Processors. In *26th IEEE International Conference on Electronics*, pages 33–36, 2019. doi: 10.1109/ICECS46596.2019.8965067.

[80] Y. Kagawa and H Iwamoto. 3D Integration Technologies for the Stacked CMOS Image Sensors. In *2019 International 3D Systems Integration Conference (3DIC)*, pages 1–4, 2019. doi: 10.1109/3DIC48104.2019.9058895.

[81] Á. Rodríguez-Vázquez, J. Fernández-Berni, J.A. Leñero-Bardallo, I. Vornicu, and R Carmona-Galán. CMOS Vision Sensors: Embedding Computer Vision at Imaging Front-Ends. *IEEE Circuits Syst. Mag.*, 18, 2018. doi: 10.1109/MCAS.2018.2821772.

[82] S. Chevobbe, M. Lepecq, K. Benchehida, M. Darouich, T. Dombek, F. Guellec, and L Millet. A Versatile 3D Stacked Vision Chip with Massively Parallel Processing Enabling Low Latency Image Analysis. In *2019 International Image Sensor Workshop*, 2019. URL https://api.semanticscholar.org/CorpusID:210114478.

[83] A. Parashar and et al. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 304–315, 2019. doi: 10.1109/ISPASS.2019.00042.

[84] L. Mei, P. Houshmand, V. Jain, S. Giraldo, and M Verhelst. ZigZag: Enlarging Joint Architecture-Mapping Design Space Exploration for DNN Accelerators. *IEEE Trans. Comput.*, 70:1160–1174, 2021. doi: 10.1109/TC.2021.3059962.

[85] C. Hong, Q. Huang, G. Dinh, M. Subedar, and Y. S Shao. DOSA: Differentiable Model-Based One-Loop Search for DNN Accelerators. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 209–224, 2023. doi: 10.1145/3613424.3623797.

[86] H. Kwon, P. Chatarasi, V. Sarkar, T. Krishna, M. Pellauer, and A Parashar. MAESTRO: A Data-Centric Approach to Understand Reuse, Performance, and Hardware Cost of DNN Mappings. *IEEE Micro*, 40:20–29, 2020. doi: 10.1109/MM.2020.2985963.

[87] S. Xi, Y. Yao, K. Bhardwaj, P. Whatmough, G.-Y. Wei, and D Brooks. SMAUG: End-to-End Full-Stack Simulation Infrastructure for Deep Learning Workloads, 2020.

[88] Y. N. Wu, J. S. Emer, and V Sze. Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2019. doi: 10.1109/ICCAD45719.2019.8942149.

[89] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P Jouppi. CACTI 5.1. Technical report, Technical Report HPL-2008-20, 2008. URL https://www.hpl.hp.com/techreports/2008/HPL-2008-20.html.

[90] R. Venkatesan, Y.S. Shao, M. Wang, J. Clemons, S. Dai, M. Fojtik, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, and et al. MAGNet: A Modular Accelerator Generator for Neural Networks. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2019. doi: 10.1145/3316781.3317770.

[91] Y.H. Chen, T. Krishna, J.S. Emer, and V Sze. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE J. Solid-State Circuits*, 52:127–138, 2017. doi: 10.1109/JSSC.2016.2616357.

[92] Y. Chen, T. Yang, J. Emer, and V Sze. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *IEEE J. Emerg. Sel. Top. Circuits Syst.*, 9:292–308, 2019. doi: 10.1109/JETCAS.2019.2910232.

[93] H. Krichene and R Prasad. A Dataflow Architecture with Distributed Control for DNN Acceleration. In *2024 13th Mediterranean Conference on Embedded Computing (MECO)*, pages 1–4, 2024. doi: 10.1109/MECO62516.2024.10577802.

[94] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, and et al. The gem5 Simulator, 2011.

[95] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 519–530, 2013. doi: 10.1145/2491956.2462176.

[96] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. Raina, C. Kozyrakis, and M Horowitz. Interstellar: Using Halide's Scheduling Language to Analyze DNN Accelerators. In *Twenty-Fifth International Conference on Architectural Support*

for Programming Languages and Operating Systems (ASPLOS), pages 369–383, 2020. doi: 10.1145/3373376.3378514.

[97] D. Kim, S. Jeong, and J.-Y Kim. Agamotto: A Performance Optimization Framework for CNN Accelerator With Row Stationary Dataflow. *IEEE Trans. Circuits Syst. I Regul. Pap.*, 70: 2487–2496, 2023. doi: 10.1109/TCSI.2023.3258411.

[98] Y. S. Shao, B. Reagen, G.-Y. Wei, and D Brooks. Aladdin: A Pre-RTL, Power-Performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures. In *41st Annual IEEE/ACM International Symposium on Computer Architecture (ISCA)*, pages 97–108, 2014. doi: 10.1109/ISCA.2014.6853196.

[99] AMD (formerly Xilinx). Vivado Design Suite User Guide: Synthesis (UG901), 2024. URL https://docs.amd.com/r/en-US/ug901-vivado-synthesis. Accessed on 15 October 2025.

[100] Inc Synopsys. Design Compiler: RTL Synthesis Solution, 2024. URL https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/design-compiler-graphical.html. Accessed on 15 October 2025.

[101] Inc Synopsys. PrimeTime: Gold Standard for Full-Chip Static Timing and Power Analysis, 2024. URL https://www.synopsys.com/implementation-and-signoff/signoff/primetime.html. Accessed on 15 October 2025.

[102] Y. N. Wu, P.-A. Tsai, A. Parashar, V. Sze, and J. S Emer. Sparseloop: An Analytical Approach to Sparse Tensor Accelerator Modeling. In *55th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1377–1395, 2022. doi: 10.1109/MICRO56248.2022.00098.

[103] A. Samajdar, J. M. Joseph, and T Krishna. AIrchitect: Automating Hardware Architecture and Mapping Optimization. In *2023 Design*, pages 1–6, 2023. doi: 10.23919/DATE56975.2023.10137333.

[104] S.-C. Kao and T Krishna. GAMMA: Automating the HW Mapping of DNN Models on Accelerators via Genetic Algorithm. In *39th International Conference on Computer-Aided Design*, pages 1–9, 2020. doi: 10.1145/3400302.3415639.

[105] S.-C. Kao, M. Pellauer, A. Parashar, and T Krishna. DiGamma: Domain-Aware Genetic Algorithm for HW-Mapping Co-Optimization for DNN Accelerators. In *2022 Design*, pages 232–237, 2022. doi: 10.23919/DATE54114.2022.9774498.

[106] S.-C. Kao, G. Jeong, and T Krishna. ConfuciuX: Autonomous Hardware Resource Assignment for DNN Accelerators using Reinforcement Learning. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 622–636, 2020. doi: 10.1109/MICRO50266.2020.00058.

[107] Chirag Sakhuja, Charles Hong, and Calvin Lin. Polaris: Multi-fidelity design space exploration of deep learning accelerators, 2024. URL https://arxiv.org/abs/2412.15548.

[108] CEA-LIST. N2D2, 20. URL https://github.com/CEA-LIST/N2D2. Accessed on 21 October 2025.

[109] Rajendra Bishnoi, Mohammad Amin Yaldagard, Said Hamdioui, Kanishkan Vadivel, Manolis Sifalakis, Nicolas Daniel Rodriguez, Pedro Julian, Lothar Ratschbacher, Maen Mallah, Yogesh Ramesh Patil, Rashid Ali, and Fabian Chersi. Multi-partner project: A deep learning platform targeting embedded hardware for edge-ai applications (neurokit2e). In *2025 Design, Automation & Test in Europe Conference (DATE)*, pages 1–7, 2025. doi: 10.23919/DATE64628.2025.10993206.

[110] Hana Krichene and Rohit Prasad. A dataflow architecture with distributed control for dnn acceleration. In *2024 13th Mediterranean Conference on Embedded Computing (MECO)*, pages 1–4, 2024. doi: 10.1109/MECO62516.2024.10577802.

[111] H. Krichene, R. Prasad, and A Mouhagir. AINoC: New Interconnect for Future Deep Neural Network Accelerators. In *Design and Architecture for Signal and Image Processing, DASIP 2023*, Lecture Notes in Computer Science. Springer, Cham, 2023. doi: 10.1007/978-3-031-29970-4_5.

[112] The pandas development team. pandas-dev/pandas: Pandas, February 2020. URL https://doi.org/10.5281/zenodo.3509134.

[113] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. *International Joint Conference on Artificial Intelligence*, 14(2):1137–1145, 1995.

[114] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 2nd edition, 2019.

[115] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001. doi: 10.1023/A:1010933404324.

[116] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29(5):1189–1232, 2001. doi: 10.1214/aos/1013203451.

[117] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794, 2016. doi: 10.1145/2939672.2939785.

[118] AMD Xilinx. *Virtex UltraScale+ FPGA Data Sheet: DC and AC Switching Characteristics*, 2023. URL https://docs.amd.com/v/u/en-US/ds923-virtex-ultrascale-plus. DS923 (v1.18).

[119] Siemens EDA. Questa Advanced Simulator: Industry Leading Functional and Fault Simulation, 2024. URL https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/. Accessed on 15 October 2025.