Hardware-Software Co-Design of Deep Neural Networks: From Handcrafted to Automated Design and Deployment

Nael Y. A. Al-Fasfous

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender:
Prof. Dr.-Ing. Georg Sigl

Prüfende der Dissertation:
1. apl. Prof. Dr.-Ing. Walter Stechele
2. Prof. Dr.-Ing. Dr. h. c. Jürgen Becker,
   Karlsruher Institut für Technologie (KIT)

Die Dissertation wurde am 04.05.2022 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 18.08.2022 angenommen.

Two truths cannot contradict one another.

GALILEO GALILEI

Acknowledgement

My deepest gratitude goes to my supervisor, mentor, and role model, Prof. Dr.-Ing. Walter Stechele, for his continuous support throughout this work. Walter's kindness, guidance, and calm demeanor gave me the confidence and reassurance I needed during the highs and lows of this journey. I am eternally indebted to Walter for his impact on this chapter of my life. I also have to express how grateful I am to have undertaken this journey with Dr.-Ing. Alexander Frickenstein and Manoj-Rohit Vemparala. The three musketeers. A key motivator to keep going when times are hard is to see that others have not stopped. We were the perfect sparring partners for each other. We celebrated each other's successes and gave comfort to each other during the hard times. Even being away from friends, family, and most colleagues during the pandemic years, I never felt lonely in this endeavor, always having them by my side. To my mother, my father, my sister, and my late brother, thank you for your continuous support throughout my life, some things cannot be expressed in words on a paper. To my friends, Ramzi, Serina, Endri, Jose, Jorge, and Anne, thank you for taking care of me like my own family would. I would like to thank my colleagues at the Chair of Integrated Systems at the Technical University of Munich, BMW AG, Politecnico di Torino, Karlsruhe Institute of Technology, and the FINN team at AMD Xilinx for being great academic and industry partners and nurturing strong collaborations that will go on beyond the scope of this work.

Abstract

Over the past decade, deep neural network (DNN) algorithms have become increasingly popular in the field of machine learning (ML). Year-on-year improvements in off-the-shelf parallel computing hardware and the accessibility of big data have democratized the training, optimization, and development of DNNs. After rapidly surpassing classical algorithms in many domains, such as autonomous driving and robotics, DNNs solidified their state-of-the-art status for a wide range of classification and forecasting problems. Along with their popularity, new use-cases emerged to incorporate them in more deployment scenarios, ranging from constrained edge deployment to safety-critical settings. These presented several challenges in hardware and software design, where tight latency, energy, and resource budgets are typically set. This work reinterprets concepts from the mature discipline of hardware-software (HW-SW) co-design, which provides processes for finding synergies when deploying complex algorithms on hardware with precise execution targets and deployment costs. Handcrafted, semi-automated, and fully-automated methodologies are proposed to introduce co-design at different stages of development with varying design challenges. Hardware models in the form of analytical schedulers and mappers, look-up tables, hardware-in-the-loop setups, and differentiable regression models are developed to inject hardware-awareness into co-design problems for general-purpose or customized platforms, and spatial or dataflow architectures. Abstraction levels are exploited to enable divide-and-conquer approaches that tackle design challenges throughout the HW-SW development life cycle. The contributions shed light on the benefits of bringing together algorithm and hardware design to achieve the targets set in both worlds, while reducing the independent development effort on both sides and avoiding incoherent design compromises. Over the course of this work, hardware components were handcrafted to suit different types of neural network computations. Genetic and gradient descent algorithms, autoencoders, and reinforcement learning agents were used to compress neural networks. Fast analytical hardware models were developed for evaluation and automated hardware design. Neural networks were made safer by analyzing threats of adversarial attacks and hardware errors on their function, and training them for joint efficiency, robustness, and accuracy preservation. The resulting co-designed algorithms tackled autonomous driving problems with high efficiency, enabled power-forecasting on multiprocessor chips, provided mask detection and correction during the COVID-19 pandemic, and empowered semi-autonomous prostheses for amputees. The contributions of this work in HW-SW co-design of DNNs brought applications with societal impact to edge devices.

Zusammenfassung

Im Verlauf der letzten zehn Jahre sind tiefe künstliche neuronale Netze (engl. deep neural networks (DNNs)), eine Kategorie selbstlernender Algorithmen, im Bereich des maschinellen Lernens (ML) immer beliebter geworden. Jährliche Verbesserungen bei handelsüblicher paralleler Computerhardware und die Zugänglichkeit von “Big Data” haben das Training, die Optimierung und die Entwicklung von neuronalen Netzen demokratisiert. Nachdem sie die klassischen Algorithmen in vielen Bereichen wie dem autonomen Fahren und der Robotik schnell überholt hatten, festigten DNNs zunehmend ihren Status als Stand der Technik für ein breites Spektrum von Klassifizierungs- und Antizipationsaufgaben. Durch ihrer Popularität entstehen neuartige Anwendungsfälle und eine Vielzahl an Einsatzszenarien, welche von Anwendungen in stark eingeschränkten eingebetteten Systemen bis hin zu sicherheitskritischen Anwendungen reichen. Dies stellte eine Reihe von Herausforderungen für das Hardware- und Softwaredesign dar, bei denen in der Regel enge Latenz-, Energie- und Ressourcenbudgets vorgegeben sind. In dieser Arbeit werden Konzepte aus der ausgereiften Disziplin des Hardware-Software (HW-SW) Co-designs neu interpretiert, die Prozesse zur Erzielung von Synergien bei der Bereitstellung komplexer Algorithmen auf Hardware mit präzisen Ausführungszielen und Bereitstellungskosten bieten. Es werden im Rahmen dieser Arbeit handgefertigte, halbautomatische und vollautomatische Methoden vorgeschlagen, um das Co-design in verschiedenen Entwicklungsstadien mit unterschiedlichen Designherausforderungen einzuführen. Hardware-Modelle in Form von analytischen Schedulern, Umsetzungstabellen, “Hardware-in-the-Loop” Simulationen und differenzierbaren Regressionsmodellen werden entwickelt, um ein Hardwareverständnis in die Co-design Aufgabe für Allzweck- oder kundenspezifische Platformen sowie in räumliche oder Datenfluss-Architekturen einzubringen. Abstraktionsebenen wurden ausgenutzt, um Teile-und-herrsche (engl. divide-and-conquer) Verfahren zur Bewältigung von Designherausforderungen während des gesamten HW-SW-Entwicklungszyklus zu ermöglichen. Die Beiträge beleuchten die Vorteile der Zusammenführung des Algorithmen- und Hardwaredesigns, um die in beiden Welten gesetzten Ziele zu erreichen und gleichzeitig den unabhängigen Entwicklungsaufwand zu reduzieren und inkohärente Designkompromisse zu vermeiden. Im Rahmen dieser Arbeit werden Hardwarekomponenten zur Berechnung von verschiedenen Arten von neuronalen Netzen entwickelt. Genetische Algorithmen und Gradientenabstiegsalgorithmen, Autocodierer und Agenten für bestärkendes Lernen (engl. reinforcement learning) werden zur Komprimierung neuronaler Netze eingesetzt. Es werden schnelle analytische Hardwaremodelle für die Bewertung und den automatischen Entwurf von Hardware entwickelt. Zudem werden neuronale Netze sicherer gemacht, indem die Bedrohungen durch gegnerische Angriffe und Hardwarefehler auf ihre Funktion analysiert und sie für eine gemeinsame Effizienz, Robustheit und Erhaltung der Genauigkeit trainiert werden. Die daraus resultierenden, gemeinsam entwickelten Algorith-

men bewältigten Herausforderungen im Bereich des autonomen Fahrens mit hoher Effizienz, ermöglichten Leistungsvorhersagen auf Multiprozessor-Chips, bietet eine Maskenerkennung und -korrektur während der COVID-19-Pandemie und ermöglichten halbautonome Prothesen für Amputierte. Die Beiträge dieser Arbeit zum HW-SW-Co-design von DNNs bringen Anwendungen mit gesellschaftlicher Bedeutung für eingebettete Systeme und Edge-Devices.

Contents

Abstract vii

Zusammenfassung ix

List of Figures xv

List of Tables xxi

List of Abbreviations xxiii

List of Symbols xxvii

1 Introduction 1
1.1 Motivation . . . 2
1.2 Objectives . . . 3
1.3 Contributions . . . 3
1.4 Academic Work and Scope . . . 4
1.5 Copyright Notice . . . 6

2 Background 7
2.1 Fundamentals of Artificial Neural Networks . . . 7
2.1.1 Dense Layers . . . 9
2.1.2 Activation Functions . . . 9
2.1.3 Convolutional Neural Networks . . . 9
2.1.3.1 Convolutional Layers . . . 10
2.1.3.2 Pooling Layers . . . 11
2.1.3.3 Batch Normalization . . . 12
2.1.3.4 Dilated Convolution Layers . . . 12
2.1.4 Learning and Classifying . . . 13
2.2 Compression and Optimization of Deep Neural Networks . . . 13
2.2.1 Data Quantization . . . 14
2.2.2 Parameter Pruning . . . 16
2.2.3 Neural Architecture Search . . . 18
2.2.4 Adversarial Training . . . 19
2.3 Hardware Acceleration of Deep Neural Networks . . . 20
2.3.1 Deep Neural Networks on General-Purpose Hardware . . . 20

2.3.2 Deep Neural Network Accelerators 21
2.3.2.1 Spatial Accelerators 22
2.3.2.2 Schedules and Dataflow Mapping 24

3 HW-SW Co-Design of Deep Neural Networks 27
3.1 Contradicting Challenges of DNN and HW Design 27
3.2 Compromises and Extending the Hand of Truce 28
3.3 Problem Statement and Paths to Effective Co-Design 31
3.3.1 Co-Design Methodologies 31
3.3.2 Executable Models 33
3.3.3 Abstraction Levels 34

4 Handcrafted Co-Design 37
4.1 OrthrusPE: Runtime Reconfigurable Processing Elements for Binary Neural Networks 37
4.1.1 BNN Training Challenges and Motivation for Reconfigurable PEs . . . 38
4.1.2 Related Work 39
4.1.3 Accurate Binary Convolutional Neural Networks 39
4.1.4 OrthrusPE 41
4.1.4.1 SIMD Binary Hadamard Product in Binary Mode 42
4.1.4.2 Arithmetic Operations in Fixed-Precision Mode 43
4.1.4.3 Mode Switching and Partial Sum Accumulation 44
4.1.5 Evaluation 46
4.1.5.1 Experimental Setup 46
4.1.5.2 Resource Utilization Analysis 47
4.1.5.3 Dynamic Power Analysis 48
4.1.6 Discussion 49
4.2 Mind the Scaling Factors: Resilience Analysis of Quantized Adversarially Robust CNNs 50
4.2.1 Hardware Fault Resilience and Adversarial Robustness 50
4.2.2 Related Work 51
4.2.2.1 Hardware Fault Resilience Analysis 51
4.2.2.2 Fault Resilient Training and Adversarial Robustness 52
4.2.3 Methodology 52
4.2.3.1 Problem Formulation: Quantization and Bit-Flips 52
4.2.3.2 Error Model and Benchmark Phases 55
4.2.4 Evaluation 55
4.2.4.1 Large Scale Resilience Analysis 57
4.2.4.2 In-depth Analysis of Adversarially Trained CNNs 59
4.2.4.3 Results and Conclusions 59
4.2.5 Discussion 61

5 Semi-Automated Co-Design **63**
5.1 Binary-LoRAX: Low-power and Runtime Adaptable XNOR Classifier for Prosthetic Hands . . . 63
5.1.1 HW-DNN Co-design for Intelligent Prosthetics . . . 64
5.1.2 Related Work . . . 65
5.1.2.1 Efficient Intelligent Prosthetics . . . 65
5.1.2.2 Binary Neural Networks for Intelligent Prosthetics . . . 65
5.1.2.3 The XILINX FINN Framework . . . 66
5.1.3 Methodology . . . 66
5.1.3.1 Training and Inference of Simple BNNs . . . 66
5.1.3.2 Hardware Architecture . . . 68
5.1.3.3 Runtime Dynamic Frequency Scaling . . . 68
5.1.3.4 SIMD Binary Products on DSP Blocks . . . 69
5.1.4 Evaluation . . . 69
5.1.4.1 Experimental Setup . . . 69
5.1.4.2 Design Space Exploration . . . 70
5.1.4.3 Runtime Dynamic Frequency Scaling . . . 70
5.1.5 Discussion . . . 73
5.2 BinaryCoP: BNN COVID-19 Face-Mask Wear and Positioning Predictor . . . 74
5.2.1 Efficient Deployment of CNNs for Mask Detection . . . 74
5.2.2 COVID-19 Face-Mask Wear and Positioning . . . 75
5.2.3 BNN Interpretability with Grad-CAM . . . 75
5.2.4 Evaluation . . . 77
5.2.4.1 Experimental Setup . . . 77
5.2.4.2 Design Space Exploration . . . 78
5.2.4.3 Grad-CAM and Confusion Matrix Analysis . . . 79
5.2.4.4 Comparison with Other Works . . . 82
5.2.5 Discussion . . . 83

6 Fully-Automated Co-Design **85**
6.1 HW-FlowQ: A Multi-Abstraction Level HW-CNN Co-Design Quantization Methodology . . . 85
6.1.1 The Tripartite Search Space . . . 86
6.1.2 Related Work . . . 87
6.1.2.1 Quantization Methods . . . 87
6.1.2.2 Quantization & Search Schemes . . . 87
6.1.2.3 Hardware Modeling . . . 87
6.1.2.4 Hardware-Software Co-Design . . . 88
6.1.3 HW-FlowQ . . . 88
6.1.3.1 HW-Model Abstraction Levels . . . 89
6.1.3.2 Genetic Algorithm . . . 91
6.1.3.3 Fitness Evaluation . . . 93
6.1.3.4 Genetic Operators . . . 94
6.1.3.5 Modeling Mixed-Precision Inference . . . 94

6.1.4 Mixed-Precision Model Validation 96
6.1.5 Cross-Abstraction Level Interactions 96
6.1.6 Evaluation 99
6.1.6.1 A HW-CNN Co-Design Example 101
6.1.6.2 Multi-Objective Genetic Optimization using NSGA-II 102
6.1.6.3 HW Modeling and Exploration 104
6.1.6.4 Mixed-Precision Quantization for Semantic Segmentation . . 107
6.1.6.5 Comparison with State-of-the-Art 108
6.1.7 Discussion 110
6.2 AnaCoNGA: Analytical HW-CNN Co-Design using Nested Genetic Algorithms 112
6.2.1 Introduction 112
6.2.2 Related Work 113
6.2.2.1 CNN-Aware Hardware Design 113
6.2.2.2 Joint HW-CNN Co-Design 113
6.2.3 Methodology 113
6.2.3.1 Bit-Serial Accelerator Modeling for BISMO 114
6.2.3.2 Genetic Quantization Strategy Search (QSS) 116
6.2.3.3 Genetic Hardware Architecture Search (HAS) 117
6.2.3.4 Model Validation and Real Hardware Measurements 117
6.2.3.5 AnaCoNGA: Nested HW-CNN Co-Design 117
6.2.4 Evaluation 120
6.2.4.1 Experimental Setup 120
6.2.4.2 Quantization Strategy Search (QSS) Loop Evaluation 121
6.2.4.3 Hardware Architecture Search (HAS) Loop Evaluation . . . 121
6.2.4.4 Analysis of AnaCoNGA Co-Designed Solutions 122
6.2.5 Discussion 126

7 Conclusion & Outlook **127**

Bibliography **131**

A Appendix **147**

List of Figures

2.1 Visualization of a single neuron in the context of a DNN. 8
2.2 Activation functions used to introduce non-linearities in DNNs. The simple rectified linear unit (ReLU) function is the most commonly used in modern DNNs due to its simplicity in terms of computation and effective training results. 10
2.3 Visualization of the convolution operation in convolutional neural networks (CNNs). The computation of a single output pixel is highlighted. 11
2.4 Reduction in error rate during the ImageNet large scale visual recognition challenge (ILSVRC). Models needed more layers and parameters to push the boundaries each year. In 2010 and 2011, classical computer-vision algorithms were used. 14
2.5 Linear quantization represents the numerical distribution with uniformly spaced quantization levels. Log-based quantization represents the more frequently occurring values more finely, while less frequent values are more sparsely represented, with higher rounding error. 16
2.6 Different example pruning regularities showing structured to unstructured parameter removal. Structured pruning can bring benefits to hardware acceleration without any specialized zero-detectors or complex memory management. . . . 17
2.7 Example handcrafted DNN architectural blocks developed to improve training and reduce total computations and parameters. 19
2.8 Abstract visualization of spatial and dataflow architectures. Spatial accelerators use an array of processing elements (PEs) to perform the parallel operations of a DNN. Dataflow architectures reflect the neural network architecture in hardware and process the layers as a classical dataflow graph. 21
2.9 Tiling and unrolling example on a spatial accelerator. 24

3.1 Inter-dependencies of hardware and software DNN optimization techniques. . . 29
3.2 Works published under the scope of this thesis categorized with respect to concepts used from the very-large-scale integration (VLSI) design domain. 32
3.3 The Gajski-Kuhn diagram (left) and the possible transitions to traverse the views and abstraction levels (right). 35

4.1 Binary bases can differentiate the values of the full-precision kernel more accurately by preserving more information through linear transformations. 40
4.2 Preconditioning signals A, B and C to compute five 3×3 Hadamard products. Pixels represented with an `X` are not relevant for this cycle of operation. 42

4.3 The DSP48E1 Slice [1]. Appended bold paths illustrate the relevant signals for our operating modes. . . . 43
4.4 SIMD register utilization of the DSP48 in OrthrusPE, with and without partial operations. . . . 44
4.5 Block diagram showing the main components of the OrthrusPE . . . 45
4.6 Switch count and partial result memory analysis for a single input channel from different convolutional layers of binary ResNet18, with $M = 3$, $N = 3$. Each point represents a different configuration of P. . . . 46
4.7 Synthesis results for look-up table (LUT) utilization across different design target frequencies. Each plot point represents a different synthesis run. . . . 47
4.8 Dynamic power estimation at different design target frequencies. Each plot point represents a different synthesis run. . . . 48
4.9 Batch-norm limits activation range at *training time*, effectively lowering v and c of the subsequent convolutional layer at *deployment time* (on hardware). Errors in the convolutional layer can at most grow in magnitude to the defined clip c of the next layer. . . . 53
4.10 Layer-wise scaling factors v of ResNet20 CNNs trained on CIFAR-10, with and without batch-norm. Works investigating bit-flips on aged CNNs (without batch-norm after every layer) cannot be extended to modern CNNs. . . . 54
4.11 Adversarial attacks apply input perturbations to cause incorrect classifications. Training for such attacks implies training for pixel value distributions outside of the natural dataset. Differently, hardware faults can occur at any point within the CNN, and are not limited to the input of the network. . . . 54
4.12 Parameters to determine bit-flip characteristics of the benchmark. . . . 56
4.13 Bit-flip experiments following algorithm 4.1 on vanilla, pruned and adversarially trained ResNet20 and ResNet56. Each bar represents the failure rate of a particular bit-flip setting $\{f, t, b, m\}$ tested over 10K test images. Each sub-figure comprises 900K bit-flip experiments. . . . 58
4.14 Convolutional layer scaling factors for vanilla trained and adversarially robust variants of ResNet20 and ResNet56. High weight decay (α_d = 0.05) brings the high scaling factors v of FastAT back to vanilla levels. . . . 60

5.1 KIT Prosthetic Hand (50th percentile female) with Zynq Z7010-based processing system . . . 65
5.2 Overview of Binary-LoRAX: binary neural network (BNN) tensor slices are fed into digital signal processing (DSP) blocks which perform high-throughput XNOR operations. DSP results are forwarded to the PEs of a matrix-vector-threshold unit (MVTU). A single MVTU of the pipeline is shown for compactness. Runtime frequency scaling allows high-performance functions, or power-saving mode. . . . 67
5.3 The large input image is sliced into smaller images and reclassified. High confidence classifications are bounded. . . . 72
5.4 Runtime frequency scaling ranging from 2MHz to 111MHz for the v-CNV prototype. . . . 72

5.5 Runtime change in operation mode based on application scenario, e.g. motion, delicate object or low battery. . . . 73
5.6 Main components of BinaryCoP. The BNN requires low memory and provides good generalization. The FINN-based accelerator allows for privacy-preserving edge deployment of the algorithms without sacrificing performance. The synthetic data helps in maintaining a diverse set of subjects and gradient-weighted class activation mapping (Grad-CAM) can be used to assert the features being learned. 76
5.7 The Grad-CAM approach used to assert that correct and reasonable features are being learned from the synthetic data. . . . 77
5.8 Binary operations and layer-wise latency estimates based on PE/single instruction multiple data (SIMD) choices for BinaryCoP-n-CNV. . . . 79
5.9 Confusion matrix of BinaryCoP-CNV on the test set. . . . 80
5.10 Grad-CAM output of two BinaryCoP variants and a single-precision floating-point (FP32) CNN. Results are collected for all four wearing positions on a diverse set of individuals. Binarized models show distinct regions of interest which are focused on the exposed part of the face rather than the mask. The FP32 model is difficult to interpret in some cases. **It is recommended to view this figure in color.** . . . 81
5.11 Grad-CAM results for age generalization. **It is recommended to view this figure in color.** . . . 82
5.12 Grad-CAM results for hair/headgear generalization. **It is recommended to view this figure in color.** . . . 82
5.13 Grad-CAM results for face manipulation with double-masks, face paint and sunglasses. **It is recommended to view this figure in color.** . . . 83

6.1 Overview of the HW-FlowQ methodology. Population $\mathcal{P}$ is evaluated on task-related accuracy ψ and hardware metrics φ. The three proposed hardware (HW)-modeling abstraction levels: *Coarse*, *Mid* and *Fine*, enable the genetic algorithm $\mathcal{G}$ to consider the hardware metrics of the CNN relative to the current design phase. 89
6.2 Iterative refinement increases the likelihood of finding the global optimum. Flow inspired by [2]. . . . 90
6.3 Input, output and optimization details of the HW-model μ abstraction levels used at each phase. After refinement, the inputs of the preceding phase are inherited to the next. . . . 91
6.4 Layer-wise genome encoding allows for intuitive use of genetic operators (crossover, mutation) to capture and maintain good localities of bitwidth-to-layer encodings from two fit parents into their offspring. . . . 92
6.5 16-bit AlexNet validation of HW-FlowQ with Eyeriss [3] and Timeloop [4] (top-left), as well as 8, 4 and 2-bit vectorized execution. Validation with Timeloop on DeepBench workloads [5] (top-right). Bit-serial execution of AlexNet (bottom-left) and DeepBench (bottom-right). . . . 97
6.6 Analysis of DRAM access and throughput on on-chip buffer size at different levels of hardware abstraction and quantization. . . . 98

6.7 2-D projections of three 3-D Pareto-fronts for ResNet56 quantization: left to right ($|\mathcal{P}|$, generations) = (25, 25), (25, 50), (50, 50). Grey to black shades represent Pareto-fronts of older to newer generations, red points belong to the final Pareto-front. **It is recommended to view this figure in color.** 103

6.8 2-D projections of 3-D Pareto-fronts of 3 exploration experiments on ResNet20 for CIFAR-10 for hardware dimensioning, bit-serial processing and dataflow variants. **It is recommended to view this figure in color.** 105

6.9 Layer-wise bitwidth strategy for BS-256 hardware. Batch size 1 (left) and 4 (right). non-dominated sorting genetic algorithm (NSGA-II) compensates for larger activations (batch=4) by lowering b_A and maintains accuracy by increasing b_W, when compared to batch=1 inference. 107

6.10 Layer-wise bitwidths (b_W=b_A) of a DeepLabv3 Pareto-choice strategy with 67.3% mean intersection over union (mIoU) on Cityscapes. Short and parallel layers have b_A equal to their respective bottom layer. 108

6.11 Qualitative results of DeepLabv3 quantization on Cityscapes scenarios. Black regions have no ground-truth labels. Pareto-choice has 21.6% fractional operations (Frac. OPs) compression compared to uniform 8-bit PACT. **It is recommended to view this figure in color.** 109

6.12 High-level abstraction of a bit-serial accelerator [6]: The dimensions $\mathrm{D}_m, \mathrm{D}_n, \mathrm{D}_k$ determine the tiling degree of matrices RHS and LHS. 116

6.13 Validation of the HW-model vs. real HW measurements for compute cycles and DRAM accesses on three BISMO configurations (HW1-3). Small and large workloads are verified from ResNet20-CIFAR-10 (left) and ResNet18-ImageNet (right). 118

6.14 AnaCoNGA: Each individual from quantization strategy search (QSS) executes its *own* hardware architecture search (HAS) multi-objective genetic algorithm (MOGA). Any QSS individual can prove itself efficient on its *own* hardware design to get a chance for its accuracy to be evaluated. QSS is relieved from optimizing hardware and is transformed to a single objective genetic algorithm (SOGA) (i.e. accuracy focused). 119

6.15 HAS: 2-D projections of a 4-D Pareto-front in a multi-objective search space. The genetic algorithm (GA) optimizes for hardware resources (LUTs, block random-access memory (BRAM)) and performance metrics (dynamic random-access memory (DRAM) accesses, execution cycles) for ResNet20 (top) and ResNet18 (bottom). The proposed analytical model allows for fast exploration and evaluation of solutions. 123

6.16 Breakdown of execution on synthesized hardware. Higher DRAM accesses are correlated with lower compute efficiency and stalls. AnaCoNGA reduces latency and DRAM accesses while maintaining high accuracy. 124

A.1 QSS: 2-D projections of a 3-D Pareto-front for optimal quantization with respect to accuracy, compute cycles, and DRAM accesses on HW3. Compute cycles and DRAM accesses are normalized to an 8-bit execution on HW3. "Reward Accuracy" is with minimal fine-tuning (not fully trained). **It is recommended to view this figure in color.** 148

A.2 Comparison of a HAS solution (D_m, D_n, D_k = 8, 14, 96) found for ResNet18-ImageNet 4-bit against the larger standard symmetric hardware configuration HW3. The CONV1 layer follows the same trend but is not shown to maintain plot scale. 148

List of Tables

4.1 Requirements of most common binary neural networks and the respective hardware operations for execution. 41
4.2 Resource Utilization results of the tested implementations. 47
4.3 Summary of results on shallow (ResNet20) and deep (ResNet56) CNNs as vanilla, pruned, and adversarially trained variants. Percentage improvement shown for FastAT α_d = 0.05 over regular FastAT. 61

5.1 Hardware results of design space exploration. Power is averaged over a period of 100 seconds of operation. 71
5.2 Hardware results of design space exploration. Power is averaged over 100s of operation. 78

6.1 Hardware configurations and normalized access energy costs used for experiments and validation. 100
6.2 ResNet20 for CIFAR-10 quantized at different abstraction levels of the Spatial-256 hardware with SOGA and NSGA-II. 101
6.3 Quantization of ResNet20 for CIFAR-10 on different hardware dimensions. . . 106
6.4 Quantization of ResNet20 for CIFAR-10 on bit-serial accelerators. 106
6.5 Quantization of ResNet20 for CIFAR-10 on different dataflows. 107
6.6 Comparison of HW-FlowQ with state-of-the-art quantization methods on Eyeriss-256 Vectorized. 110
6.7 Classification of HW-CNN optimization methods. 114
6.8 Hardware configurations used for model validation. 117
6.9 Hardware and quantization search space. 121
6.10 Quantization and hardware design experiments. Uniform and standalone QSS are executed on a standard edge variant (HW3) used in [7]. Latency and DRAM are measured on hardware. 125

A.1 Network architectures and hardware dimensioning used in Binary-LoRAX and BinaryCoP. Pool, ReLU and batch normalization layers not shown. FC_3 | [25/4] indicates YCB (25 classes) or MaskedFace-Net (4 classes). 147

List of Abbreviations

AI artificial intelligence.

ALU arithmetic logic unit.

ASIC application specific integrated circuit.

AVX advanced vector extensions.

BFA bit-flip attack.

BNN binary neural network.

BRAM block random-access memory.

CAM class activation mapping.

CMFD correctly masked face dataset.

CMOS complementary metal-oxide-semiconductor.

CNN convolutional neural network.

CPU central processing unit.

CTC computation-to-communication.

DFG data flow graph.

DNN deep neural network.

DRAM dynamic random-access memory.

DSP digital signal processing.

DVFS dynamic voltage and frequency scaling.

EMG electromyographic.

FastAT fast adversarial training.

FC fully-connected.

FFT fast Fourier transform.

FGSM fast gradient sign method.

FIFO first in, first out.

FP32 single-precision floating-point.

FPGA field programmable gate array.

FPS frames per second.

Frac. OP fractional operation.

GA genetic algorithm.

GEMM general matrix multiplication.

GPU graphics processing unit.

Grad-CAM gradient-weighted class activation mapping.

HAS hardware architecture search.

HDL hardware description language.

HIL hardware-in-the-loop.

HLS high-level synthesis.

HV hypervolume.

HW hardware.

HW-SW hardware-software.

ILSVRC ImageNet large scale visual recognition challenge.

IMFD incorrectly masked face dataset.

INT8 8-bit signed integer.

IRO input reuse oriented.

IS input-stationary.

LUT look-up table.

MAC multiply-accumulate.

MFR mean failure rate.

mIoU mean intersection over union.

ML machine learning.

MLP multilayer perceptron.

MoC model of computation.

MOGA multi-objective genetic algorithm.

MVTU matrix-vector-threshold unit.

NAS neural architecture search.

NoC network on chip.

NSGA-II non-dominated sorting genetic algorithm.

OP operation.

ORO output reuse oriented.

OS output-stationary.

PCB printed circuit board.

PE processing element.

PGD projected gradient descent.

PL programmable logic.

PLL phase-locked loop.

PMBus power measurement bus.

PPE personal protective equipment.

PS processing system.

PSUM partial sum.

QSS quantization strategy search.

ReLU rectified linear unit.

RF register file.

RL reinforcement learning.

RoI region of interest.

RS row-stationary.

RTL register-transfer level.

SDF synchronous data flow.

SGD stochastic gradient descent.

SIMD single instruction multiple data.

SNN spiking neural network.

SoC system-on-chip.

SOGA single objective genetic algorithm.

SRAM static random-access memory.

STE straight-through estimator.

SWU sliding-window unit.

TOPS trillion operations per second.

VLIW very long instruction word.

VLSI very-large-scale integration.

WRO weight reuse oriented.

WS weight-stationary.

List of Symbols

$BS_{speedup}$ Speed-up of bit-serial quantization.

C Effective capacitance.

C_i Input channel dimension.

C_o Output channel dimension.

D_k PE SIMD lanes.

D_m PE array height.

D_n PE array width.

F Fitness.

I Input image.

K_x Weight kernel width.

K_y Weight kernel height.

M Number of binary weight bases.

Mem_{psum} Memory required for partial sums.

N Number of binary activation bases.

P Predictions.

P_{dyn} Dynamic Power.

Q Set of possible quantization levels.

TC_o Output channel tile size.

V_{dd} Supply voltage.

$V_{speedup}$ Speed-up of vectorized quantization.

X_i Input tensor width.

X_o Output tensor width.

Y Ground truth, label.

Y_i Input tensor height.

Y_o Output tensor height.

Γ Learning rate.

α Non-linear activation function.

α_m Scaling factor for binary activation base $m \in M$.

α_d Weight decay.

α_{sw} Switching activity.

β_n Scaling factor for binary weight base $n \in N$.

β_{bn} Batch normalization shift.

δ Adversarial perturbation.

ϵ_{bug} Adversarial perturbation budget.

ϵ_{stab} Numerical stability constant.

η_{OPs} Compute efficiency.

γ_{bn} Batch normalization scale.

$\hat{P}$ Predictions with bit-flips.

$\mathbb{B}$ Set of binary numbers.

$\mathbb{E}$ Expected loss.

$\mathbb{R}$ Set of real numbers.

$\mathbf{A}$ Activation tensor/matrix.

$\mathbf{B}$ Multi-base binarized weight tensor.

$\mathbf{H}$ Multi-base binarized activation tensor.

$\mathbf{W}$ Synaptic weight tensor/matrix.

$\mathbf{b}$ Neuron bias vector.

$\mathcal{B}$ Set of bit positions.

$\mathcal{D}$ Dataset.

$\mathcal{F}$ Set of bit-flip injection frequencies.

$\mathcal{G}$ Genetic algorithm.

$\mathcal{L}$ Loss.

$\mathcal{L}_{CE}$ Cross-entropy loss.

$\mathcal{L}_{Reg}$ Regularization loss.

$\mathcal{M}$ Set of faulty MAC unit percentages.

$\mathcal{N}$ Neural network.

$\mathcal{P}$ Population.

$\mathcal{T}$ Set of datatypes.

$\mathtt{x_b}$ Binary variable.

$\mathtt{x_q}$ Quantized variable.

μ Hardware model.

μ_{bn} Mini-batch mean.

ψ Task-related accuracy.

ρ Individual/Genome.

σ_{bn} Mini-batch variance.

τ_{thold} Threshold for BNN batch normalization.

φ Hardware metrics (latency or power).

a^{in} Input activation pixel.

a^{out} Output activation pixel.

b Neuron bias value.

b^l Binary weight kernel.

b_A Bitwidth for activations.

b_W Bitwidth for weights.

b_{max} Maximum bitwidth supported.

c Quantization clipping threshold.

f Clock frequency.

h^{l-1} Binary input activation tensor 2-D slice.

m Matrix rows.

n Matrix columns.

p Partial result or sum.

p_y Unrolling (parallel) degree.

p_{mut} Mutation probability.

s Convolution stride.

t Threshold for accuracy degradation.

v Quantization scaling factor.

w Synaptic weight value.

x_f Floating point variable.

1 Introduction

ALGORITHMS are compositions of functional and conditional operations carried out in a defined sequence. Fundamentally, algorithms are omnipresent, both in natural and artificial forms, simple and complex, explainable and emergent. The seemingly miraculous existence of animate, biological organisms is, at its core, a composition of looped algorithms in the form of chemical reactions involving inanimate material. The elegance of algorithms can be compelling enough to convince an observer of being greater than the sum of its parts. Consciousness and intelligence are examples of such phenomena.

Algorithms can be thought of as blueprints, existing as abstract concepts. Without an implementation in the real world, they cannot interact with or affect real systems. In biological systems, algorithms are implemented through matter and chemical processes. The existence of matter in particular amounts under specific conditions results in sequences of chemical processes and reactions, based on the fundamental physical properties of the matter. The algorithm is, therefore, more of a description of what happens in such settings than a *planned* sequence of operations to be executed in a particular manner. Scaling up in abstraction from fundamental chemical processes, we can consider biological algorithms at the neuronal level in the nervous systems of complex organisms. The placement, positioning, firing rate, and other properties of the neurons in the context of an organism's brain produce its reasoning and interaction with the physical world. The algorithm resulting in how an individual behaves is not *designed* beforehand, but emerges from the neural networks and the biological processes of the individual. The algorithm and the biological matter are one and the same.

For artificial algorithms, a human-designed set of operations is planned. This abstract artificial algorithm must then be executed in some form to be tested in the real world. Here, a second stage of design takes place, where the execution medium must be decided. Hence, artificial algorithms require two phases of development, one for the algorithm itself, and one for its execution medium.

The difference between algorithms emerging in nature and algorithms developed by humans is that the former fundamentally leads to one holistic manifestation of algorithm and medium, whereas the latter is a two-step process, which decouples the planned artificial algorithm and the design of the medium, through which it interacts with the real world. The argument for co-designing artificial algorithms and the execution medium is not only logical, but also the most natural approach to achieve efficient, performant, and seemingly miraculous algorithms, as those we observe in nature.

The work presented in this dissertation focuses on co-designing hardware and artificial deep neural network algorithms. Hardware components were handcrafted to suit different types of neural network computations [8]. Genetic and gradient descent algorithms, autoencoders, and reinforcement learning agents were used to compress neural networks, while fast analytical

hardware models are developed for evaluation and automated hardware design [9, 10, 11, 12, 13, 14, 15]. Neural networks were made safer by analyzing threats of adversarial attacks and hardware errors on their function, and training them for joint efficiency, robustness, and accuracy preservation [16, 17, 18]. The resulting co-designed algorithms tackled autonomous driving problems with high efficiency [19, 20], enabled power-forecasting on multiprocessor chips [21, 22, 23], provided mask detection and correction during the COVID-19 pandemic [24], and empowered semi-autonomous prosthetics to help amputees [25].

In the following sections 1.1-1.4, the motivation of the work is elaborated further, the objectives are listed out, and the scope of the dissertation is defined. In chapter 2, a coarse background and literature review of relevant related topics is presented. Chapter 3 covers the pitfalls of sub-optimal deployments, incoherent co-design, and other challenges faced by machine learning (ML) and hardware (HW) engineers in this field. This chapter also introduces the paths proposed to achieve hardware-software (HW-SW) co-design for deep neural network (DNN) deployments. Chapters 4, 5, and 6 elaborate the proposed paths towards HW-DNN co-design by presenting six design challenges tackled by handcrafted, semi-automated, and fully-automated co-design techniques. Finally, chapter 7 concludes the dissertation and presents the outlook and future work in this field.

1.1 Motivation

Over the past decade, DNN algorithms have become increasingly popular in the field of ML. Year-on-year improvements in off-the-shelf parallel computing hardware and the accessibility of big data have democratized the training, optimization, and development of DNNs [26, 27]. After rapidly surpassing classical algorithms in many domains, such as autonomous driving and robotics, DNNs solidified their state-of-the-art status for a wide range of classification and forecasting problems [28, 29, 30].

Along with the popularity of DNNs, new use-cases emerged to incorporate them in more deployment scenarios, ranging from constrained edge deployment to safety-critical settings. These present several challenges in hardware and software design, where tight latency, energy, and resource budgets are typically set. In essence, the challenge of developing efficient DNN deployments necessitates searching multiple design spaces, from hardware designs to neural network architectures and the compression space. Several high-impact works in this field have investigated one design space at a time, with the assumption that solutions from other design spaces are static and/or already provided [7, 31, 32, 33, 34, 35, 36, 37]. A large co-design opportunity is often missed, where multiple search spaces are open for co-exploration.

HW-SW co-design is a matured discipline which provides processes for finding synergies when deploying complex algorithms on hardware with precise execution targets and deployment costs [38, 2]. These processes heavily rely on divide-and-conquer approaches to achieve near-optimal solutions in prohibitively large design spaces. Techniques from this field can produce solutions to new problems emerging in the field of edge DNN deployment. In this work, HW-SW co-design methods are reinterpreted and applied to DNN deployment challenges.

1.2 Objectives

This work sets out to identify the challenges of HW-DNN co-design at different stages of development, for different use-cases, and different deployment goals. The objectives can be summarized in the following:

- Identifying key characteristics to classify DNN design and optimization problems, which help in planning and choosing the correct methods for search and metaheuristics, design automation, and handcrafted design.
- Investigating the implications of incoherent HW-DNN co-design and identifying methods to introduce awareness towards hardware and deployment goals, such as latency, energy, and resource budgets, into DNN training and optimization methods.
- Analyzing the design challenges that occur throughout the development life cycle of DNN deployments and breaking down the complex co-design paradigm into stages that can be addressed independently with less effort.

1.3 Contributions

Inspired by the discipline of HW-SW co-design, this work studies the holistic formulation of hardware design, DNN design and training, and optimization techniques, through a combination of models, metaheuristics, and expert knowledge. Different paths towards co-design are proposed based on the properties of the design problem at hand. Enabled by these paths, multiple design challenges which necessitate handcrafted to fully-automated solutions are presented.

The contributions of this work can be summarized in the following:

- Proposing handcrafted, semi-automated, and fully-automated co-design methodologies to apply HW-SW co-design for DNN deployment at different stages of development and different classes of design challenges.
- Developing and comparing different hardware-awareness injection methods, such as high-level analytical models, look-up tables, hardware-in-the-loop setups, and differentiable models, each well-suited for different design and optimization problems.
- Employing very-large-scale integration (VLSI)-inspired abstraction levels to enable divide-and-conquer approaches which tackle challenges throughout the HW-DNN development life cycle. This lowers the design effort by reducing the complexity of problems at each stage of development.
- Achieving tightly-coupled HW-DNN deployments for semi-autonomous prosthetics, face detection, and autonomous driving, bringing DNN applications with high societal impact to edge devices.

1.4 Academic Work and Scope

The following papers were published in peer-reviewed conferences and journals during the course of this work and fall under the scope of this thesis:

- **N. Fasfous, L. Frickenstein, M. Neumeier, M.R. Vemparala, A. Frickenstein, E. Valpreda, M. Martina, W. Stechele**, "Mind the Scaling Factors: Resilience Analysis of Quantized Adversarially Robust CNNs". *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2022
- **N. Fasfous, M.R. Vemparala, A. Frickenstein, E. Valpreda, D. Salihu, J. Höfer, A. Singh, N.S. Nagaraja, H.J. Vögel, N.A.V. Doan, M. Martina, J. Becker, W. Stechele**, "AnaCoNGA: Analytical HW-CNN Co-design using Nested Genetic Algorithms". *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2022 [1]
- **N. Fasfous, M.R. Vemparala, A. Frickenstein, E. Valpreda, D. Salihu, N.A.V. Doan, C. Unger, N.S. Nagaraja, M. Martina, W. Stechele**, "HW-FlowQ: A Multi-Abstraction Level HW-CNN Co-design Quantization Methodology". *ACM Transactions on Embedded Computing Systems (TECS)*, 2021
- **N. Fasfous, M.R. Vemparala, A. Frickenstein, M. Badawy, F. Hundhausen, J. Höfer, N.S. Nagaraja, C. Unger, H.J. Vögel, J. Becker, T. Asfour, W. Stechele**, "Binary-LoRAX: Low-power and Runtime Adaptable XNOR Classifier for Semi-Autonomous Grasping with Prosthetic Hands". *International Conference on Robotics and Automation (ICRA)*, 2021
- **N. Fasfous, M.R. Vemparala, A. Frickenstein, L. Frickenstein, M. Badawy, W. Stechele**, "BinaryCoP: Binary Neural Network-based COVID-19 Face-Mask Wear and Positioning Predictor on Edge Devices". *IEEE International Parallel & Distributed Processing Symposium, Reconfigurable Architectures Workshop (IPDPS-RAW)*, 2021 [2]
- **N. Fasfous, M.R. Vemparala, A. Frickenstein, W. Stechele**, "OrthrusPE: Runtime Reconfigurable Processing Elements for Binary Neural Networks". *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2020
- **M.R. Vemparala, N. Fasfous[3], A. Frickenstein, E. Valpreda, M. Camalleri, Q. Zhao, C. Unger, N.S. Nagaraja, M. Martina, W. Stechele**, "HW-Flow: A Multi-Abstraction Level HW-CNN Codesign Pruning Methodology". *Leibniz Transactions on Embedded Systems (LITES)*, 2022
- **P. Mori, M.R. Vemparala, N. Fasfous, S. Mitra, S. Sarkar, A. Frickenstein, L. Frickenstein, D. Helms, N.S. Nagaraja, W. Stechele, C. Passerone**, "Accelerating and Pruning CNNs for Semantic Segmentation on FPGA". *Design Automation Conference (DAC)*, 2022

[1] Best paper award nominee.
[2] XILINX Open Hardware Design Competition winner.
[3] Equal contribution to first author.

- **M.R. Vemparala, N. Fasfous[4], L. Frickenstein, A. Frickenstein, A. Singh, D. Salihu, C. Unger, N.S. Nagaraja, W. Stechele**, “Hardware-Aware Mixed-Precision Neural Networks using In-Train Quantization”. *British Machine Vision Conference (BMVC)*, 2021
- **M.R. Vemparala, N. Fasfous, A. Frickenstein, S. Sarkar, Q. Zhao, S. Kuhn, L. Frickenstein, A. Singh, C. Unger, N.S. Nagaraja, C. Wressnegger, W. Stechele**, “Adversarial Robust Model Compression using In-Train Pruning”. *IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2021 [5]
- **E.H. Chen, M.R. Vemparala, N. Fasfous, A. Frickenstein, A. Mzid, N.S. Nagaraja, J. Zeisler, W. Stechele, D. Burschka**, “Investigating Binary Neural Networks for Traffic Sign Detection and Recognition”. *IEEE Intelligent Vehicles Symposium (IV)*, 2021
- **M. Sagi, N.A.V. Doan, N. Fasfous, T. Wild, A. Herkersdorf**, “Fine-Grained Power Modeling of Multicore Processors using FFNNs”. *International Journal of Parallel Programming (IJPP)*, 2022
- **M. Sagi, M. Rapp, H. Khdr, Y. Zhang, N. Fasfous, N.A.V. Doan, T. Wild, J. Henkel, A. Herkersdorf**, “Long Short-Term Memory Neural Network-based Power Forecasting of Multi-Core Processors”. *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021
- **M.R. Vemparala, A. Frickenstein, N. Fasfous[6], L. Frickenstein, Q. Zhao, S. Kuhn, D. Ehrhardt, Y. Wu, C. Unger, N.S. Nagaraja, W. Stechele**, “BreakingBED - Breaking Binary and Efficient Deep Neural Networks by Adversarial Attacks”. *Intelligent Systems Conference (IntelliSys)*, 2021
- **A. Frickenstein, M.R. Vemparala, N. Fasfous[7], L. Hauenschild, N.S. Nagaraja, C. Unger, W. Stechele**, “ALF: Autoencoder-based Low-rank Filter-sharing for Efficient Convolutional Neural Networks”. *The Design Automation Conference (DAC)*, 2020
- **M. Sagi, N.A.V. Doan, N. Fasfous, T. Wild, A. Herkersdorf**, “Fine-Grained Power Modeling of Multicore Processors using FFNNs”. *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS XX)*, 2020
- **M.R. Vemparala, N. Fasfous[8], A. Frickenstein, M.A. Moraly, A. Jamal, L. Frickenstein, C. Unger, N.S. Nagaraja, W. Stechele**, “L2PF - Learning to Prune Faster”. *International Conference on Computer Vision & Image Processing (CVIP)*, 2020
- **M.R. Vemparala, A. Singh, A. Mzid, N. Fasfous, A. Frickenstein, F. Mirus, H.J. Voegel, N.S. Nagaraja, W. Stechele**, “Pruning CNNs for LiDAR-based Perception in Resource Constrained Environments”. *IEEE Intelligent Vehicles Symposium Workshops (IV Workshops)*, 2021

[4]Equal contribution to first author.
[5]Best paper award nominee.
[6]Equal contribution to first author.
[7]Equal contribution to first author.
[8]Equal contribution to first author.

The following papers were published in peer-reviewed conferences and journals during the course of this work, but are out of the scope of this thesis:

- **A. Srivatsa, N. Fasfous, N.A.V. Doan, S. Nagel, T. Wild, A. Herkersdorf**, "Exploring a Hybrid Voting-based Eviction Policy for Caches and Sparse Directories on Manycore Architectures". *Microprocessors and Microsystems*, 2021
- **A. Srivatsa, S. Nagel, N. Fasfous, N.A.V. Doan, T. Wild, A. Herkersdorf**, "HyVE: A Hybrid Voting-based Eviction Policy for Caches". *IEEE Nordic Circuits and Systems Conference (NorCAS)*, 2020 [9]
- **N.A.V. Doan, A. Srivatsa, N. Fasfous, S. Nagel, T. Wild, A. Herkersdorf**, "On-Chip Democracy: A Study on the Use of Voting Systems for Computer Cache Memory Management". *International Conference on Industrial Engineering and Engineering Management (IEEM)*, 2020 [10]

1.5 Copyright Notice

Parts of this work have been published in [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25] by the Institute of Electrical and Electronics Engineers (IEEE), the Association for Computing Machinery (ACM), and Springer-Verlag. Personal use of this material is permitted. Permission from IEEE, ACM and/or Springer must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

[9] Best paper award winner.
[10] Best paper award nominee.

2 Background

ARTIFICIAL INTELLIGENCE is a broad term referring to human-engineered algorithms which implement functions that exhibit a general perception of intelligence. Although defining intelligence itself is complex, artificial intelligence (AI) typically refers to algorithms which perform classification, forecasting, decision-making, and generative tasks. Machine learning (ML) algorithms are an approach to realize artificial intelligence. Particularly, ML algorithms are able to learn the task at hand without being explicitly programmed for it. Exposure to data allows such algorithms to improve their performance in executing the desired task. Deep neural networks (DNNs) are one such class of algorithms, loosely inspired by biological neural networks, and are the focus of this dissertation. In the following sections, the fundamental components of DNNs are presented. Following that, a specialized form of DNNs suited for computer vision tasks is presented, namely convolutional neural networks (CNNs). The procedure through which such algorithms learn and perform their tasks is elaborated. The challenges of deploying these algorithms in resource constrained settings are discussed, followed by an overview of optimization and hardware acceleration techniques applied to DNNs.

2.1 Fundamentals of Artificial Neural Networks

In the pursuit of developing intelligent machines, researchers intuitively attempted to study and take inspiration from the structure of biological brains [39]. A fundamental building block of a biological brain's structure is the neuron, which takes input stimuli through its *dendrites* and produces activation signals through its *axons*. The relationship between the input stimuli and the output activation defines the function of a single neuron. Dendrites and axons of neurons are connected over synapses, which transfer the signals from one neuron to another, creating a neural network. It is important to note, that the function of a biological brain is substantially more complex than the description provided here, with many details yet to be discovered. Nonetheless, based on this high-level specification, a loose representation of biological neural networks can be captured in computation graphs, where interconnected nodes represent neurons, each executing a simple function, but collectively achieving a more complex task.

Figure 2.1 shows the abstraction of a biological neuron to an artificial one, forming the basic building block of an artificial neural networks. Correspondingly, equation 2.1 describes the arithmetic operations necessary to compute the output of a neuron, a^{out}. An artificial neuron incorporates the output of the preceding neurons' axons into its function by performing a weighted sum on the incoming activations a^{in}. The weights w mimic the synaptic connections which amplify or attenuate the inputs to the neuron. The weights indicate the relevance of the

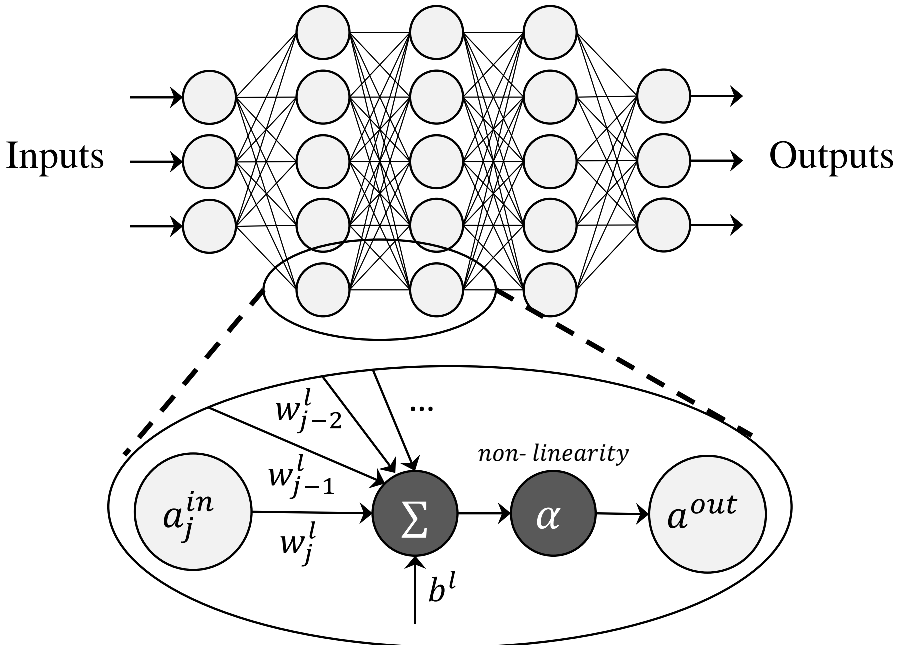


Figure 2.1: Visualization of a single neuron in the context of a DNN.

information from each preceding neuron with respect to the current neuron's sub-function. This creates paths through the network which get activated when certain data patterns are observed. The phenomenon is similar to strong and weak neural firing paths in biological brains. Following the weighted sum of inputs, a bias term b is added to the resulting value. The composition of linear operations till this point cannot represent a non-linear function, which would limit the neural network's representation capability. For this reason, the neuron's output is finally *activated* with a non-linear function α, typically referred to as the activation function.

$$a^{out} = \alpha(\sum_j w_j a_j^{in} + b) \tag{2.1}$$

Generally, artificial neural networks have their neurons organized in *layers*, as shown in figure 2.1. In the simplest form, a feed-forward network has layers in a sequential order, one feeding its output to the next. Other networks exist where recurrent reuse of activations among the layers takes place, or bypass and parallel paths are incorporated in the network [28]. When referring to DNNs, the networks are composed of more than three layers. The depth of a neural network is simply one parameter in deciding its architecture, among other important parameters such as the number of neurons, their organization, the reuse and bypass of activations, etc. Deeper neural networks tend to exhibit better performance on complex tasks, as they have more layers to aggregate simple features into complex ones [28, 40]. In the following subsections, common neural network layers and operations relevant to this dissertation are introduced.

2.1.1 Dense Layers

Dense layers, often referred to as fully-connected (FC) layers, appeared in the earliest artificial neural networks, the multilayer perceptrons (MLPs). The dense layer is a composition of neurons, each performing the function described in equation 2.1 and shown in figure 2.1. A dense layer l's weights are organized into a 2-D matrix $\mathbf{W}^l \in \mathbb{R}^{X_i \times Y_o}$, where X_i represents the input activation dimension and Y_o the number of neurons in the layer. Given the input activation vector $\mathbf{A}^{l-1} \in \mathbb{R}^{X_i}$ and output activation vector $\mathbf{A}^l \in \mathbb{R}^{Y_o}$, the operation can be formulated as the matrix-vector computation shown in equation 2.2. $\mathbf{b} \in \mathbb{R}^{Y_o}$ represents the vector of bias terms b, and α, as previously introduced, the activation function applied element-wise to the pre-activation output vector.

$$\mathbf{A}^l = \alpha(\mathbf{A}^{l-1}\mathbf{W}^l + \mathbf{b}) \tag{2.2}$$

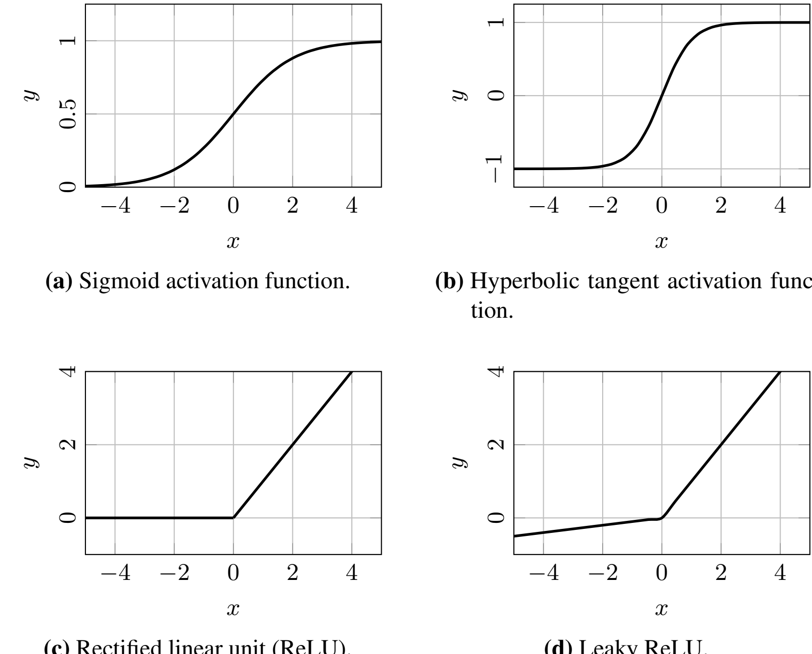
Dense layers remain prevalent in modern neural network architectures, such as transformers and convolutional neural networks [40, 28, 41]. They are typically memory-bound due to the high number of unique weights that are needed for each fully-connected neuron. For this reason, most modern neural network architectures employ dense layers only after the input activation dimensions have been reduced by other preceding layers in the network. Their main function in CNNs is to combine features extracted from preceding layers into classification logits [42].

2.1.2 Activation Functions

DNNs are predominantly composed of linear scale (multiplication) and shift (addition) operations. Compositions of purely linear functions can only result in linear functions. To achieve complex, non-trivial functions, non-linear representations must be captured by DNNs [43]. Non-linear functions are introduced in DNNs at the output of every neuron, as shown in equation 2.1 and 2.2. These are often referred to as activation functions. Historically, the *sigmoid* and *tanh* functions were used to activate neurons. Apart from being complex to execute on hardware, these activation functions were proven to not present any advantages over simpler alternatives such as the rectified linear unit (ReLU) function. The ReLU function simply zeroes out all negative values and allows the positive values to pass unchanged. This piecewise linear function is simple to execute in hardware, introduces activation sparsity which can be exploited for efficient inference, and simplifies the training procedure of DNNs. Although variations of the ReLU, such as the leaky ReLU, have been proposed to allow gradient flow over the negative range of the piecewise linear function, the basic ReLU function remains one of the most popular activation functions in the field. The aforementioned activation functions are shown in figure 2.2.

2.1.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) are specialized neural networks which are well-suited for data with spatial, localized correlation [42]. Unlike fully-connected DNNs, CNNs have an inductive bias by way of their neurons considering small 2-D regions across a third channel dimension, instead of the entire input volume. The discoveries of Hubel and Wiesel in the 1950s and 60s showed that neurons in the visual cortex of small mammals focus on small localized



(a) Sigmoid activation function.

(b) Hyperbolic tangent activation function.

(c) Rectified linear unit (ReLU).

(d) Leaky ReLU.

Figure 2.2: Activation functions used to introduce non-linearities in DNNs. The simple ReLU function is the most commonly used in modern DNNs due to its simplicity in terms of computation and effective training results.

regions of the visual field [44]. Their discoveries also classified neurons of the visual cortex into simple and complex ones, based on the features they react to. For example, simple neurons react to lines and edges of specific orientation, whereas complex neurons identify sub-features and patterns irrespective of orientation. This inspired Fukushima to develop the first neural networks with such inductive biases [39], followed by the modernization of the concept by LeCun et al. who created LeNet [42], laying the groundwork for today's CNNs.

2.1.3.1 Convolutional Layers

The weights of a convolutional layer are organized as a 4-D tensor in $\mathbb{R}^{K_x \times K_y \times C_i \times C_o}$. A set of weights $K_x \times K_y$ form a *kernel*, which defines the 2-D receptive field of the convolution. All kernels along the input channel dimension C_i represent a single *filter*. All filters along the output channel dimension C_o compose the weights of the convolutional layer. Each neuron in a convolutional layer reacts to a $K_x \times K_y \times C_i$ region of the input. The subsequent convolutional layer would effectively have a larger receptive field, as it aggregates simpler features detected by the preceding convolutional layer [40]. This inductive bias not only makes CNNs perform better on localized visual data, but also lowers their weight count compared to a fully-connected DNN. For a 32×32 pixel image with 3 color channels, a single fully-connected neuron would require 3072 weights. In contrast, a typical 3×3×3 convolution filter would have 27 weights, which are then shared among multiple neurons reacting to different regions of the image as the filter moves

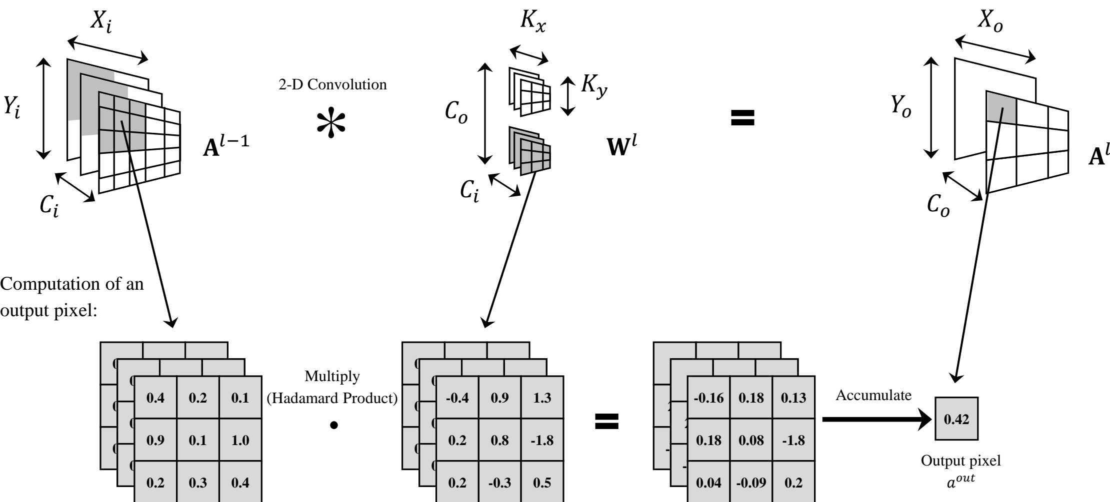


Figure 2.3: Visualization of the convolution operation in CNNs. The computation of a single output pixel is highlighted.

over the input by a stride of s. For completeness, equation 2.3 represents the computation of a single neuron in a convolutional layer (i.e. a single output pixel), which is also visualized in figure 2.3. The input activation tensor of layer l is denoted by $\mathbf{A}^{l-1} \in \mathbb{R}^{X_i \times Y_i \times C_i}$ and the output tensor is $\mathbf{A}^l \in \mathbb{R}^{X_o \times Y_o \times C_o}$. X_i and Y_i are the spatial width and height of the input activation, whereas X_o and Y_o correspond to the width and height of the output activation. Activation tensors in CNNs can also be referred to as feature maps. The two tensors $\mathbf{A}^{l-1}$ and $\mathbf{W}^l$ represent the input feature maps and the weights of the convolutional layer, respectively. The bias addition and batch dimension are not shown for simplicity. Lastly, s represents the stride of the weight kernels over the input feature map.

$$\mathbf{A}^l[c_o][x_o][y_o] = \sum_{c_i}^{C_i} \sum_{k_x}^{K_x} \sum_{k_y}^{K_y} \mathbf{A}^{l-1}[c_i][x_o \cdot s + k_x][y_o \cdot s + k_y] \cdot \mathbf{W}^l[c_o][c_i][k_x][k_y] \tag{2.3}$$

2.1.3.2 Pooling Layers

Pooling layers downsample the spatial dimensions of the intermediate activation maps in the network by applying a striding window operation which collapses the covered region into a single output. This single output pixel is often the maximum value that is present in the pooling window (max-pool), or the average of the overall values in the window (average-pool) [45]. Pooling layers offer many advantages, from reducing the computational complexity and memory demands to regularization effects which control overfitting.

2.1.3.3 Batch Normalization

Batch normalization layers condition the activations to have zero mean and unit variance [46]. This speeds up the training process and generally improves the accuracy of DNNs. Although a definitive explanation for these improvements has not been found, it is thought that the reason lies in the network not needing to learn widely different input distributions for each batch of inputs during training, mitigating the problem of internal covariate shift [46]. Equation 2.4 shows the batch normalization operation applied to one activation pixel a. μ_{bn} is the mean of the batch activations and σ_{bn} is the standard deviation. ϵ_{stab} is an arbitrarily small value appended to maintain numerical stability. γ_{bn} and β_{bn} are scale and shift parameters learned to improve the representation capability of the normalized tensor.

$$a^{norm} = \frac{a - \mu_{bn}}{\sqrt{\sigma_{bn}^2 - \epsilon_{stab}}} \gamma_{bn} + \beta_{bn} \tag{2.4}$$

Batch normalization also allows for high learning rates during training, reduces the emphasis on weight initialization, and improves generalization. These advantages have made batch normalization a fundamental layer in modern CNNs as well as other DNN architectures. Nevertheless, it has some disadvantages, most prominent of which is the increase in computational overhead at run-time, as well as introducing a discrepancy between the model's performance on training and test samples versus real-world data. These reasons have motivated researchers to investigate normalization-free neural networks [47], however the approach still remains prevalent in state-of-the-art DNNs to date.

2.1.3.4 Dilated Convolution Layers

Early CNNs were used for classification tasks, which involve predicting the presence of objects in a scene. With further development, researchers were able reuse the inductive biases of CNNs for other vision tasks, such as object detection, localization, and semantic segmentation. In semantic segmentation, each pixel of the input image is given a classification [48]. This requires far more information about locality than simple classification tasks, since the network needs to precisely *segment* the object in the scene by classifying each pixel belonging to it. In classification-based CNNs, the input spatial dimension is reduced throughout the network as features get aggregated. However, for semantic segmentation CNNs, the input's spatial dimensions must be preserved to produce the desired pixel-wise output. This was initially achieved by using de-convolution layers, where zeroes were introduced into the feature maps for upsampling [48]. More recently, dilated convolutions offered an alternative method of capturing contextual information, without diminishing the input resolution [29]. For dilated convolution, zeroes are inserted into the $K_x \times K_y$ kernel, which increases the receptive field of the convolution without increasing the number of parameters. With dilated convolution, the input spatial dimension can remain large, and the CNN can capture contextual information by spreading out its receptive field. As an added benefit, adding zeroes into the kernel allows for optimization possibilities on hardware, where these computations can be skipped.

2.1.4 Learning and Classifying

Having defined the building blocks of DNNs, the process of *training* and deploying them for *inference* tasks is discussed in this section. As mentioned previously, DNNs are essentially computation graphs with operations performed on the data flowing through them. Their *learning* characteristic is achieved by updating their weights to improve their performance on the target task. Stochastic gradient descent (SGD) is commonly used to learn the weights which suit a particular task. Gradient descent is an optimization technique which nudges parameters in the direction of the steepest slope in a multi-dimensional space with respect to a *cost function*. The cost in the case of a supervised classification task can be the distance between the ground truth and the classification produced by the DNN being trained. The weights are therefore nudged through gradient descent such that the cost of the overall function is minimized. In DNN training, the cost is typically referred to as the *loss* $\mathcal{L}$. To control the size of the nudge in the direction of the steepest slope, the learning rate Γ is multiplied by the computed gradient, before subtracting it from the current weight's value.

$$w_i^{updated} = w_i - \Gamma(\frac{\partial \mathcal{L}}{\partial w_i}) \tag{2.5}$$

Since the weights of the network contribute to the loss at the output of the computation graph, the chain-rule can be applied to find the gradients $\frac{\partial \mathcal{L}}{\partial w_i} \forall i$, where i refers to the index of a particular weight in the neural network. Computing the gradients and nudging the weights to better values is referred to as *backpropagation*, shown in equation 2.5 . In practice, training on complex, large datasets cannot be performed with the standard gradient descent approach. This would imply that the gradients computed for backpropagation result from the *entire* training dataset, which can surpass millions of samples for many common datasets [26]. For this reason, an approximation of the gradient descent approach is typically applied, namely the SGD approach. Here, only a sub-set of the training dataset is considered in each training step, based on which the weights are updated. Once enough training steps cover the entire training dataset, a training *epoch* is complete.

After training, the DNN can be deployed to perform the intended task. When deployed for an inference task, the network is typically only executed in a forward-pass, as a standard computation graph. This process is less computationally intensive compared to training, as no gradient computation or weight updates are necessary. Nevertheless, as the size and computational diversity of modern DNNs grow, inference can still create challenges in real-time, embedded deployment settings.

2.2 Compression and Optimization of Deep Neural Networks

This work is mainly focused on the efficient deployment of DNN algorithms in constrained, edge settings. The building blocks of DNNs introduced in section 2.1 are typically used to compose ever larger networks when targeting complex tasks. This trend is clear when observing the year-over-year improvement in classification performance of DNNs during the ImageNet large scale visual recognition challenge (ILSVRC) and their corresponding growth in computation complexity and memory requirements, shown in figure 2.4. The same trend also appears in

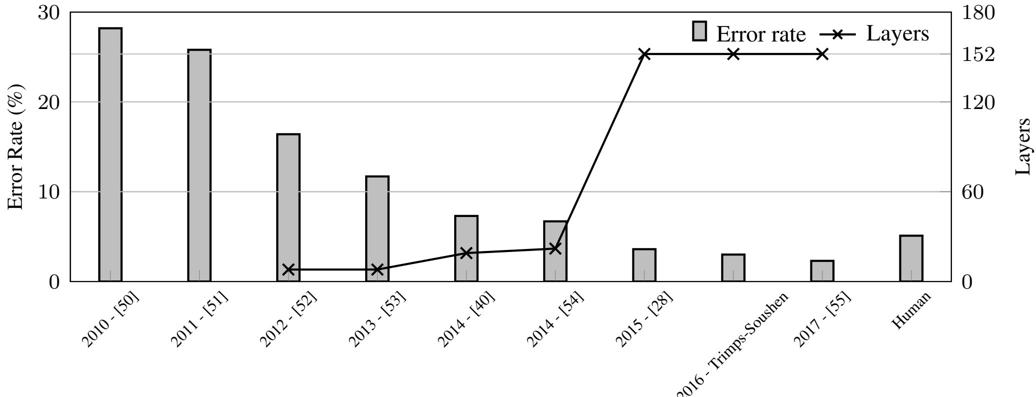


Figure 2.4: Reduction in error rate during the ILSVRC. Models needed more layers and parameters to push the boundaries each year. In 2010 and 2011, classical computer-vision algorithms were used.

other domains, such as large-scale natural language processing, where transformers have already surpassed 530 billion parameters in model size [49]. Next to compute complexity and model size, the variety of layer types and dimensions in modern DNNs introduces algorithmic diversity, which requires flexible hardware components for optimal execution across the network. As a consequence of these software requirements, hardware design becomes more difficult, given the tight area, power, latency, throughput, and safety thresholds typically defined in modern edge applications. In this section, methods for compressing and optimizing DNN algorithms for efficient deployment are presented.

2.2.1 Data Quantization

Data in the context of DNNs can represent weights, activations, intermediate partial sums, or backpropagation gradients. The representation of numerical data in computational hardware heavily affects the computation complexity, memory requirements, precision, learning behavior, and error resilience of the DNN [16]. The representation of a number is composed of the number format and the number of binary bits available. Common number formats include floating-point and fixed-point representations. The single-precision floating-point (FP32) format refers to 32-bits split into 1 sign-bit, 8 exponent bits, and 23 mantissa bits. The expressiveness of the floating-point format enables the representation of very large and very small numbers with a *floating* decimal point [56]. This comes at the cost of more expensive computation logic in terms of complexity, area, power, and latency. The fixed-point format can represent a predefined static range of numbers with *fixed* decimal point precision. For most applications, using the high-precision floating-point format for inference brings no advantages to the accuracy of the DNN [57]. This has made fixed-point and integer formats the standard nowadays on commercial edge inference hardware.

DNN training is typically performed with the FP32 numerical representation in modern training frameworks. The benefit of FP32 is the flexibility to represent large and small values

of weights and activations accurately, as well as the fine gradients computed and applied during backpropagation. The FP32 neural network can then be quantized to a simpler representation, such as the fixed-point or integer representations, before deploying it on constrained, edge hardware. The process of converting a fully-trained FP32 DNN to a quantized, lower precision representation is referred to as *post-training* quantization [57, 58]. Equation 2.6 shows the basic principle of linear quantization for an arbitrary FP32 operand x_f into a more constrained numerical representation $\mathtt{x_q}$.

$$\mathtt{x_q} = clip(round(x_f/v), c) \tag{2.6}$$

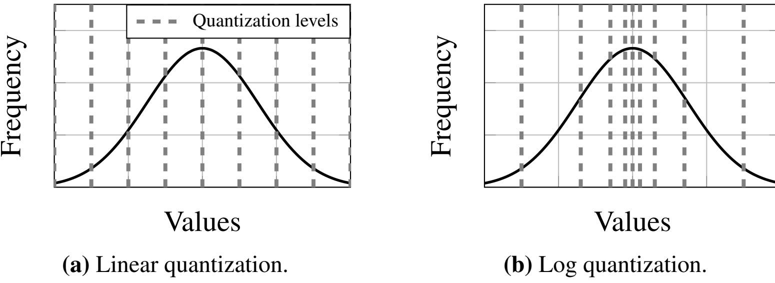
The scaling factor v translates the quantized range of values which $\mathtt{x_q}$ can take into the larger, more precise x_f range. The $round$ operation pushes the smooth values of x_f into the limited values of $\mathtt{x_q}$'s quantized numerical representation. The $clip$ operator cuts-off values of the larger numerical representation of x_f beyond $[-c, c]$, where c is the clipping limit. This maintains symmetric linear quantization.

Different to post-training quantization, the DNN may also be quantized during the training procedure in *quantization-aware* training schemes [57, 35, 36]. This is particularly necessary when the final deployment targets below 8-bits of precision [35]. Here, the training remains in FP32, however, clipping and rounding functions are introduced directly into the DNN's computation graph. During backpropagation, a straight-through estimator (STE) is employed to skip over non-differentiable or zero-gradient operations [59]. This approximates the weight updates and allows the gradients to flow through the layers of the quantized network. The FP32 weights which are updated are referred to as *latent* weights, as they are not the actual weights taking effect in the forward pass or at deployment time. This form of quantization is particularly useful when training 1-bit neural networks, often referred to as binary neural networks (BNNs). For the most basic form of BNNs, a $sign$ function is employed to convert activations and latent weights into the binary format, as shown in equation 2.7, where the float value x_f is converted to the binary representation $\mathtt{x_b}$. BNNs are therefore extremely compute and memory efficient. In the hardware implementation, all weight parameters require 1-bit of memory for storage and data movement, and the -1 values are represented as binary 0's to perform multiplications using simple XNOR logic operations [60]. The trade-off in such low-bit DNNs is the reduced information capacity, which often leads to lower task-related accuracy. Different BNN training schemes have been proposed to deal with these challenges, particularly when both weights and activations are binarized [60, 61, 62, 34].

$$\mathtt{x_b} = sign(x_f) = \begin{cases} 1 & \text{if } x_f \geq 0, \\ -1 & \text{otherwise} \end{cases} \tag{2.7}$$

With any digital numerical representation, 2^b unique values can be represented, where b is the number of bits allotted to the representation. As weight and activation data in DNNs typically follow non-uniform distributions, the unique 2^b values may be used to perform non-linear quantization and better represent the values which appear in the computations of the network. In this case, number ranges which appear frequently in the DNN's computations may be represented with less rounding error than numbers which appear rarely. This can be a logarithmic distribution of the 2^b unique values across a portion of the number line, as shown in figure 2.5.

Another form of non-linear quantization can be a simple look-up table of 2^b values that should be represented [56]. The bits b are then used only as indices to read the true value of the operand from the look-up table, which can be stored at a higher bitwidth than b. The true values stored in the look-up table can be distributed across the number line arbitrarily and learned by the network during training.



(a) Linear quantization. **(b)** Log quantization.

Figure 2.5: Linear quantization represents the numerical distribution with uniformly spaced quantization levels. Log-based quantization represents the more frequently occurring values more finely, while less frequent values are more sparsely represented, with higher rounding error.

Recent works have also investigated mixed-precision DNNs, where substructures, e.g. layers, filters, and/or datatypes, can have different quantization levels [7, 15, 10, 63, 35]. The numerical distribution in substructures can vary largely from one part of the DNN to another. This makes a single quantization scheme sub-optimal for many parts of the network. For example, in CNNs, the fully-connected layers at the end of the network can be represented with much less precision than the feature-extracting convolutional layers [64]. Nevertheless, developing mixed-precision DNNs can be challenging. First, finding the optimal mixed-precision configuration can be formulated as a search problem. The search space for such problems is typically very large; for datatype-wise, layer-wise mixed quantization of an L layer CNN, Q^{2L} solutions exist, where Q is the set of possible quantization levels, i.e. supported bitwidths [9]. Some works propose searching this space using a reinforcement learning (RL) agent [7] or a genetic algorithm (GA) [10], while other works try to find the optimal bitwidth for each layer at training time, without adding any overhead of metaheuristic search agents [15]. Second, the hardware deployment platform must support and gain a speed-up from the proposed Q levels, which typically requires more complex, non-standard arithmetic units and dynamic memory alignment [65, 66, 67, 68, 69].

2.2.2 Parameter Pruning

Parameter pruning is a technique used to slim down DNNs by removing unimportant weights or weight substructures [70, 71, 72]. These can be low-magnitude synaptic weights which have little to no influence on the performance of the DNN, or the weights whose removal leads to a minimal change in task-loss. Reducing the model's size and complexity by removing substructures can have a direct impact on the inference efficiency of the final deployment [31, 73]. Additionally, pruning can introduce a regularization effect which mitigates the network's overfitting on the training data [71].

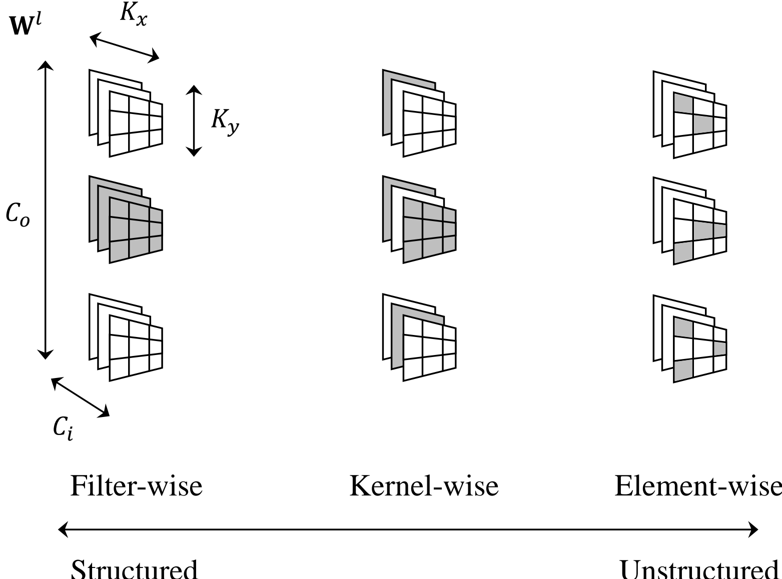


Figure 2.6: Different example pruning regularities showing structured to unstructured parameter removal. Structured pruning can bring benefits to hardware acceleration without any specialized zero-detectors or complex memory management.

Many heuristics emerged to decide *which* DNN substructures can be pruned [70, 71, 72, 73, 31, 74, 75, 76, 77]. L1-norm pruning is a common technique where the norm guides the pruning algorithm to remove substructures of low magnitude parameters, and thereby, low influence on the function of the DNN [74]. Other works identified different heuristics, such as geometric median [75] and lasso regression [76], similarly determining the saliency of neurons based on the guiding metrics. The pruning problem can also be formulated as a search problem, where an algorithm must search for the optimal set of substructures to be removed. Works involving RL agents [31], GAs [12, 11], and other metaheuristics have shown the effectiveness of combining guiding metrics with search algorithms.

Other works perform pruning without the use of heuristics, but instead try to *learn* the saliency of neurons [37, 13, 17]. For example, an autoencoder attached to the target layer can produce a pruning mask during training to decide which neurons can have an effect on the DNN's output [13]. At the end of the training, the produced mask translates to the pruning configuration which can be applied before deployment. Differently, the in-train pruning approach proposed in [17] updates pruning masks through SGD during the training process. An in-train approach has the advantage of allowing the network to learn the task, the optimal pruning masks, as well as other targets such as robustness against adversarial examples, within the training process at no extra GPU-hour costs.

Another important aspect to consider during parameter pruning is the regularity of the substructures being removed. For a given sparsity ratio, this has a direct impact on the accuracy degradation to be expected from the pruning procedure, as well as the hardware benefits that can be exploited at deployment time [31]. Generally, fine-grain, irregular weight pruning results in high compression and maintains high task-related accuracy [73]. However, the irregularity breaks

the structured parallelism in the DNN's computational workloads. Identifying which weights to skip and which ones to execute prohibits general-purpose computation platforms and standard accelerators from achieving a speed-up with this pruning regularity and its irregular memory access patterns [78, 79]. More coarsely, the pruning algorithm may remove larger structures such as entire neurons of dense layers, kernels and channels of convolutional layers, or attention heads in transformers. Pruning large structures generally translates to a change in the tensor's dimension. For example, a 4-D weight tensor of a convolutional layer in $\mathbb{R}^{K_x \times K_y \times C_i \times C_o}$ would maintain the same dimensions if individual weights were pruned. However, removing an input channel would shrink the C_i dimension. The same applies for pruning an output channel and the C_o dimension. Most DNN accelerators are essentially tensor processing units, resulting in a direct improvement in hardware performance when tensors are shrunk in this manner. The downside to coarse, structured pruning is that the task-related accuracy can quickly degrade at high pruning rates [31]. Structured and unstructured pruning examples are visualized in figure 2.6. It is worth mentioning that extracting benefits from irregular parallelism is nevertheless an important field of research in DNN hardware design [78, 79]. Recently, the *Ampere* general-purpose graphics processing unit (GPU) architecture by NVIDIA provided the means for semi-structured pruning, where specialized hardware units can detect up to 2 pruned values out of each 4 elements, offering a trade-off between irregular and structured pruning [80].

2.2.3 Neural Architecture Search

Taking a step back, the neural architecture as a whole can be designed in an optimized, resource-aware manner. Many state-of-the-art models in literature have been handcrafted by machine learning experts, based on heuristics, experience, and existing work [81, 82, 83, 28]. The number of decisions to be made in designing a DNN is very large, making it highly unlikely to ever find the optimal parameters for any given task and hardware platform. Researchers in the field of neural architecture search (NAS) have already employed metaheuristics to solve such problems, however, the challenge of training time remains at the core of this discipline [84, 85, 86, 87, 88].

Over the past decade, specialized layers have been developed to maintain the functionality of classical DNNs while reducing the computational complexity and parameter count [82, 29]. For example, depthwise-separable convolutions involve two stages to replace the classical convolutional layer [82]. The depthwise convolution involves C_i 2-D filters in $\mathbb{R}^{K_x \times K_y \times C_i}$, each convolved with only one 2-D channel from the input. This convolution keeps the output tensor dimensions equal to the input activation tensor. This is followed by pointwise convolution which is a standard convolution with a weight tensor of unit kernel dimensions in $\mathbb{R}^{1 \times 1 \times C_i \times C_o}$. A standard convolution would involve $K_x \times K_y \times C_i \times C_o \times X_o \times Y_o$ multiplications, which is higher than the multiplications of depthwise-separable convolutions $C_i \times X_o \times Y_o(K_x \times K_y + C_o)$. Similarly, other works have developed modules, such as the fire and inception modules, which transform the input to reduce the overall parameter and operation count, while maintaining the function of the standard convolution [89, 54].

Other architectural innovations involve residual connections, which are bypass paths that allow gradients to flow more effectively through the network during backpropagation [28]. These connections unlocked the potential of deeper DNNs for more complex problems and are still found in the most cutting-edge models to date [90]. However, such connections can introduce

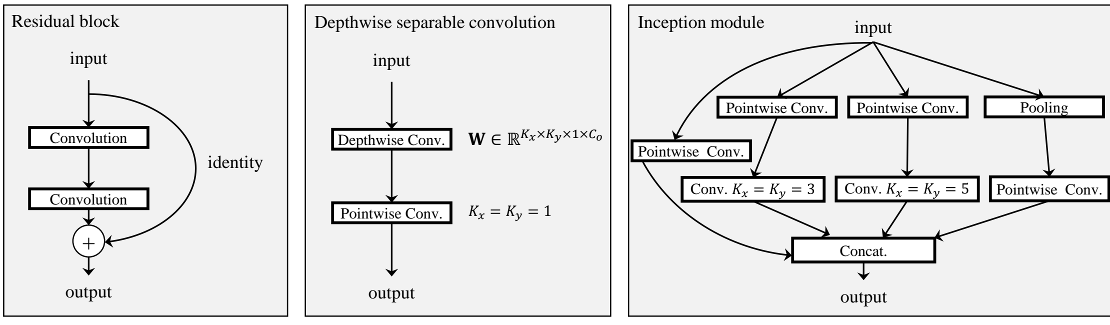


Figure 2.7: Example handcrafted DNN architectural blocks developed to improve training and reduce total computations and parameters.

inference challenges, as activations from past layers need to be stored in memory and reused at deeper stages of the DNN before being discarded. Figure 2.7 shows some popular DNN blocks that were designed through handcrafted NAS.

Depending on how granular the search space is defined, NAS can inherently perform compression using quantization and pruning [84, 88]. For example, if two layers with equal dimensions but different quantization degrees are considered unique solutions in the NAS space, then the search is jointly finding the architecture and its quantization in the same process. Consequently, the efficiency of the architectures being considered can be measured with respect to a target hardware platform. This can be done using hardware-in-the-loop (HIL) setups, differentiable and analytical hardware models or look-up table (LUT) approaches, where measurements are collected on the hardware ahead of the search experiment.

2.2.4 Adversarial Training

An increasingly important metric for DNN deployment in safety-critical scenarios is robustness against adversarial attacks in the form of input perturbations [18]. Adversarial attacks against a neural network can be formulated as an optimization problem of finding the minimal perturbation δ for an input image I that changes the prediction of the neural network $\mathcal{N}$. Attacks can be classified as white-box or black-box attacks, ranging from the attacker having full knowledge of the network parameters and the gradients during backpropagation to no information at all about the neural network. During adversarial training, perturbed examples are introduced to maximize the loss $\mathcal{L}$ with respect to the label Y, within a reasonable perturbation budget ϵ_{bug}, as shown in equation 2.8.

$$\min_{\mathbf{W}} \underset{(I,Y)\sim\mathcal{D}}{\mathbb{E}} \left[\max_{|\delta|\leq\epsilon_{bug}} \mathcal{L}\left(\mathcal{N}(I+\delta, \mathbf{W}), Y\right) \right] \tag{2.8}$$

A set of randomly sampled images from the dataset $\mathcal{D}$ is chosen, where the expected loss $\mathbb{E}$ on the random samples is minimized through an adversarial training scheme. A commonly used attack to introduce imperceptible adversarial perturbations is the fast gradient sign method (FGSM) [91], which was is one of the first white-box attacks to be developed. The advantage of FGSM is that generating an adversarial example is faster than with other attack methods, such as projected gradient descent (PGD) [92]. FGSM in combination with random initialization is

particularly effective to incorporate into the training loop to obtain adversarial training with a small overhead of GPU-hours, as presented in fast adversarial training (FastAT) [93]. For the final evaluation of adversarial robustness, the neural network is typically exposed to an unseen adversarial attack method.

2.3 Hardware Acceleration of Deep Neural Networks

DNNs are simple computation graphs largely composed of the multiply-accumulate (MAC) operation presented in equation 2.1. Given that these simple arithmetic operations are performed on large tensors, the parallelism potential of DNN workloads is very high. In a simple execution schedule, the computation of each output activation within one layer can be parallelized, whereas the layers are executed in sequence. More advanced scheduling schemes introduce inter-layer parallelism, by starting the subsequent layer's computations as soon as sufficient activations are available [94]. DNNs with identity connections, parallel layers, early terminating branches, and diverse workload dimensions result in further scheduling and data movement possibilities, making the scope for hardware optimization as large and complex as the scope for neural network optimization.

2.3.1 Deep Neural Networks on General-Purpose Hardware

A natural candidate for accelerating DNNs is the standard, off-the-shelf graphics processing unit (GPU). GPUs are general-purpose computation platforms typically used for video and image processing tasks. They are equipped with a large number of simple arithmetic logic units (ALUs) for highly parallel computation workloads [80]. The ALUs of a GPU are typically simple, with very minimal control logic. This makes GPUs well-suited for large, uniform, parallel workloads, where many simple cores are tasked with the same job in a single instruction multiple data (SIMD) processing scheme. Most DNN computations can be represented as general matrix multiplications (GEMMs). This transformation of DNN workloads to more general GPU-friendly workloads can sometimes result in replication, redundant memory accesses, and other sub-optimal execution characteristics [56]. GPUs compensate for these redundancies by having large and fast memories, high clock rates, and many ALUs that require minimal programming and control to perform their uniform, parallel task [95]. However, even in the case where the latency is compensated by more capable hardware, the power consumption of such a platform will generally be higher than a custom DNN accelerator solution [96]. Modern GPU architectures have clearly been influenced by the popularity of DNNs in research and industry [95, 97, 80]. NVIDIA's *Volta* architecture incorporated design changes to exploit the advantages of DNN quantization as a compression technique [95]. The introduction of *tensor cores* supported the execution of lower bitwidth workloads at higher throughput in a vectorized manner. The more recent *Ampere* architecture incorporated the 2:4 sparsity scheme to support semi-structured pruning, as mentioned in section 2.2.2 [80].

Another general-purpose execution platform for DNNs is the traditional central processing unit (CPU). CPUs are not the optimal platform for large-scale SIMD processing, which is highly applicable to DNN workloads. However, being the most common and ubiquitous computation

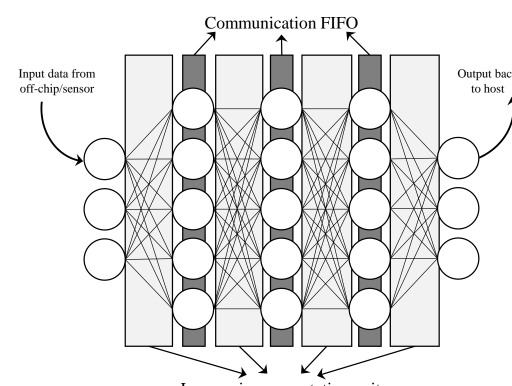


Figure 2.8: Abstract visualization of spatial and dataflow architectures. Spatial accelerators use an array of PEs to perform the parallel operations of a DNN. Dataflow architectures reflect the neural network architecture in hardware and process the layers as a classical dataflow graph.

platform, the execution of DNNs on CPUs is inevitable in some cases. CPUs can be a practical option for smaller DNNs, latency-relaxed applications, or in constrained embedded systems where a dedicated DNN accelerator is not feasible. ML software libraries have also been optimized to exploit the capabilities of modern CPUs, such as hyper-threading and vectorized instructions [98]. For example, CPUs with support for advanced vector extensions (AVX) use the wider SIMD registers to pack more operations when DNNs are quantized to lower bitwidths [99].

2.3.2 Deep Neural Network Accelerators

The highly parallelizable workloads in modern DNNs call for highly parallel compute platforms for low-latency and high-throughput inference. Most specialized DNN accelerators in literature are composed of an array of processing elements (PEs). These are referred to as *spatial accelerators* [56]. PEs of spatial accelerators are typically more capable compared to a GPU's cores or ALUs, having more control logic and, in some cases, direct communication with other PEs over an interconnect, such as a network on chip (NoC). The advantage of having more complex PEs is their programmability, which allows them to effectively handle DNN workloads without replication or transformation into more standard arithmetic workloads such as GEMMs. The communication between PEs also means that data can be reused across the PE array, without having to access the on-chip buffers or a higher-memory hierarchy each time [3]. Additionally, PEs can share intermediate results or partial sums (PSUMs), allowing them to collectively share the effort of the workload more effectively. Finding the optimal dataflow over a PE array for DNN inference is an active field of research and is critical in achieving effective HW-SW co-design, as every workload could have a different movement pattern to achieve its execution targets in terms of energy, latency or both [4, 100, 101, 102]. This will be discussed in more detail in the upcoming subsections. Figure 2.8 shows an example of a generic spatial DNN accelerator.

Other DNN accelerator architectures are based on dataflow graph processing [103]. This type of architecture involves synthesizing the DNN graph directly onto hardware as a traditional

dataflow graph, i.e. a pipeline of nodes communicating over first in, first out (FIFO) buffers, as shown in figure 2.8. Such architectures are well-suited for reconfigurable fabric such as field programmable gate arrays (FPGAs), where a new computation graph can be flashed onto the fabric whenever the DNN needs to be changed. Another advantage here is that no communication is necessary with off-chip memory during computation as the entire graph is on-chip and the intermediate results are passed from one computation node to the next directly. The architecture also unrolls the layers of the DNN, allowing multiple inputs to be processed in different parts of the graph to improve throughput [103]. For example, while input i is being processed by the graph node for layer l, the next input i+1 can already be processed by the preceding graph node for layer l-1. With a well-dimensioned pipeline, this architecture can achieve a high-throughput, low-latency execution. However, there are some disadvantages to an in-hardware graph-based implementation. For large DNNs, it might not be feasible to fully unroll the graph and all its synaptic weights onto the fabric of an FPGA [104]. Additionally, for DNNs with residual paths, a large amount of memory might be required to store the intermediate results of previous nodes of the graph until it can be used by the deeper layers of the graph. This can ultimately stall the pipeline and deplete the memory resources of the programmable logic.

More exotic accelerators have also appeared in research, particularly in the field of neuromorphic computing [105, 106]. However, in this work, the focus remains on more classical graph-based and parallel computing architectures.

In the following subsections, a more detailed discussion on spatial accelerators is presented, as well as an elaboration of the challenge of finding the optimal schedule for these accelerators with respect to different DNN workloads.

2.3.2.1 Spatial Accelerators

Spatial accelerators with contributions covering novel interconnect [107], efficient data access and movement patterns [3], support for run-time sparsity detection [79, 78], and variable precision arithmetic [69, 68, 66] have been proposed in literature. Apart from the individual contributions, these accelerators share some general characteristics which help in identifying the properties of a generic spatial accelerator. Generally, these accelerators have a 3+1 level memory hierarchy, composed of an off-chip memory typically implemented as a dynamic random-access memory (DRAM), an on-chip memory typically implemented as a static random-access memory (SRAM), registers inside each PE, and the combined registers of all the PEs can be considered as yet another memory hierarchy, accessible over the interconnect (hence, 3+1). Referring back to figure 2.8, these memory levels can be seen on the generic spatial accelerator architecture. The requirement for such memory hierarchies stems from the fundamental challenge of unrolling the entire computation load onto the PE array in a single processing pass. To elaborate this point, algorithm 2.1 shows the nested-loop representation of the convolution operation presented earlier in equation 2.3.

Ideally, the data required by the entire loop is accessed only once from an off-chip memory resource and then reused exhaustively by the PEs, without any redundant off-chip accesses. This implies that during all the iterations where a particular data element is involved, all the other data that it is reused against (computed with) also fits in the lowest-level memory. In practice, this approach is not feasible for large algorithms, as fast and efficient near-compute memories,

Algorithm 2.1 Nested loop representation of the convolutional layer execution from equation 2.3

```
Input: A^{l-1}[C_i][X_i][Y_i]
Weights: W^l[C_o][C_i][K_x][K_y], Stride: s
Output: A^l[C_o][X_o][Y_o]                  ▷ Required tensors for the convolution operation
for c_o = 0 ; c_o < C_o ; c_o + + do                           ▷ Output channel iterator
    for c_i = 0 ; c_i < C_i ; c_i + + do                         ▷ Input channel iterator
        for x_o = 0 ; x_o < X_o ; x_o + + do          ▷ Output horizontal spatial iterator
            for y_o = 0 ; y_o < Y_o ; y_o + + do          ▷ Output vertical spatial iterator
                for k_x = 0 ; k_x < K_x ; k_x + + do              ▷ Kernel horizontal iterator
                    for k_y = 0 ; k_y < K_y ; k_y + + do            ▷ Kernel vertical iterator
                        w = W^l[c_o][c_i][k_x][k_y]              ▷ Iterators as tensor indices
                        a^{l-1} = A^{l-1}[c_i][x_o · s + k_x][y_o · s + k_y]
                        A^l[c_o][x_o][y_o]+ = w · a^{l-1}   ▷ Core MAC, Write to output tensor A^l
```

such as PE registers, are usually a limited, precious resource due to manufacturing costs and on-chip area constraints. Nevertheless, data reuse is still possible to some extent through clever scheduling techniques [108, 109, 3]. The main computation is at the core of the inner-most loop, where many elements are accessed in multiple iterations of the higher loops. Specifically, reuse occurs when the indices of the parameters involved in the inner-most computation remain fixed for some loops before iterating in others. In hardware, this translates to a single element being stored at a lower-level memory for multiple iterations before being purged to make space for new data. For optimal reuse to occur, no single element should be read more than once from a higher-level memory.

Loop-tiling is an approach to efficiently exploit the entire memory hierarchy by dividing the nested-loop into shallower loops, which can fit on multiple memory levels. The loop-tiling strategy effectively decides which tiles of the CNN computation will take place in one round of communication with a lower-level memory (on-chip buffer). An example of the C_o loop in algorithm 2.1 being tiled is given in algorithm 2.2. Tiles of size TC_o are sent by the outer loop (off-chip memory) to the inner loop (on-chip memory). Another visual example of loop-tiling is presented in figure 2.9.

The order of the loops can also be manipulated without affecting the algorithm through *loop-reordering*. For example, in algorithm 2.1, the execution iterates over the y_o index before incrementing the x_o index, allowing a set of elements with index x_o to reside longer on the lower-level memory while iterating over all possible elements $y_o \in Y_o$. Swapping these two loops would result in y_o elements residing longer on the lower-level memories. This essentially helps in extracting improved reuse opportunities, since the upper-level loops remain on the lower-level memories of the hardware architecture, thus closer to the compute units.

Finally, *loop-unrolling* is the third loop optimization technique, which can be applied once a memory level is distributed spatially. The degree of unrolling is limited by the parallelism offered by the hardware architecture and the on-chip interconnect. For example, in algorithm 2.1, the kernel's elements K_y can be assigned to p_y spatially distributed PEs, effectively executing several p_y loop iterations in parallel as shown in algorithm 2.2. Another visual example of loop-unrolling

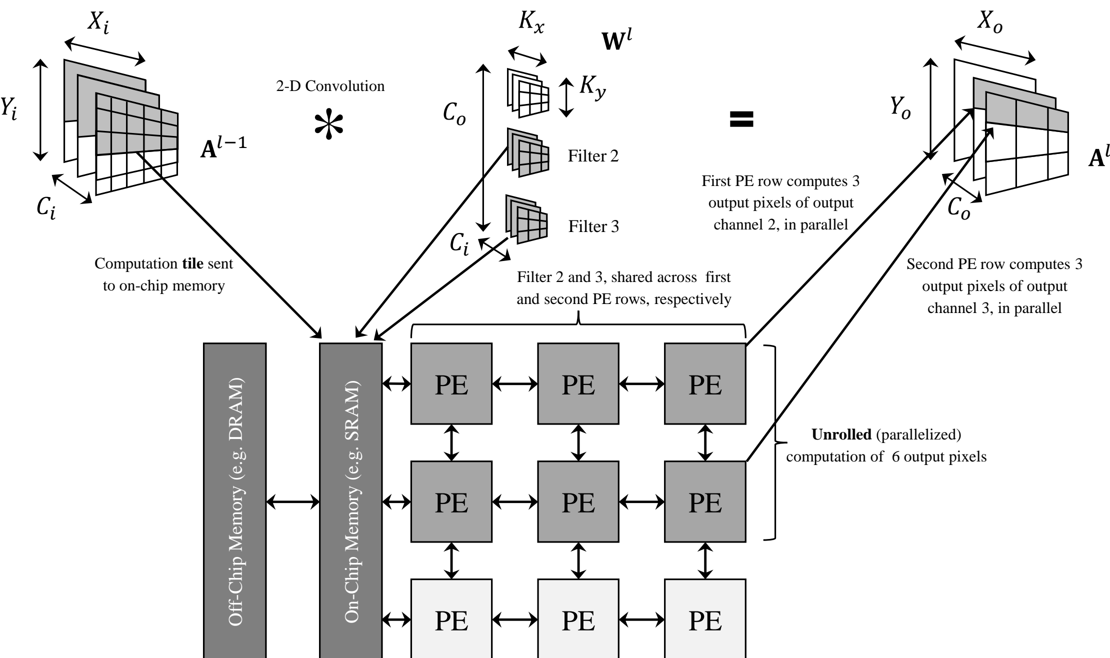


Figure 2.9: Tiling and unrolling example on a spatial accelerator.

is presented in figure 2.9, where six output pixels from two different channels are computed in parallel by six PEs, after the required filters and input pixels are unrolled over them.

Algorithm 2.2 Output channel tiling and weight kernel unrolling example based on algorithm 2.1

Input: $\mathbf{A}^{l-1}[C_i][X_i][Y_i]$
Weights: $\mathbf{W}^l[C_o][C_i][K_x][K_y]$, **Stride:** s
Output: $\mathbf{A}^l[C_o][X_o][Y_o]$ ▷ Required tensors for the convolution operation
for $c_o = 0\ ;\ c_o < C_o\ ;\ c_o{+}= TC_o$ **do** ▷ **Off-chip** tile iterator with tile size TC_o
  ... ▷ Other nested loops
  for $tc_o = 0\ ;\ tc_o < TC_o\ ;\ tc_o{+}{+}$ **do** ▷ **On-chip** output channel tile iterator
    for $k_y = 0\ ;\ k_y < K_y\ ;\ k_y{+}= p_y$ **do** ▷ Unrolling p_y operations in parallel
      $\mathbf{w} = \mathbf{W}^l[tc_o][c_i][k_x][k_y : k_y + p_y]$ ▷ $\mathbf{w}$ and $\mathbf{a}$ vectors for p_y parallel operations
      $\mathbf{a}^{l-1} = \mathbf{A}^{l-1}[c_i][x_o \cdot s + k_x][y_o \cdot s + k_y : y_o \cdot s + k_y + p_y]$
      $\mathbf{A}^l[tc_o][x_o][y_o]{+}= \mathbf{w} \cdot \mathbf{a}^{l-1}$ ▷ Core MAC, Write to output tensor $\mathbf{A}^l$

2.3.2.2 Schedules and Dataflow Mapping

The loop optimization techniques briefly discussed in the previous subsection can be used to define schedules and computation mapping schemes [4, 100]. Tiling and reordering affect which data is available to the PE array, thereby directly influencing the effectiveness of the unrolling scheme. Since these optimization techniques are all codependent, they compose a large search space of schedules, where every possible tile, order, and unrolling degree represents a solution.

The schedule and mapping search space naturally depends on the workload being scheduled as well as the hardware dimensions (memory and PE array size) being considered. Constraints are introduced by disallowing certain communication patterns among PEs, or limiting the memory hierarchy size, which shrinks the search space of valid schedules and simplifies the hardware design.

In terms of loop reordering, three ordering schemes were identified in [110], which influence the reuse of one datatype, before being accessed again from the off-chip memory. These can be categorized into output reuse oriented (ORO), input reuse oriented (IRO), and weight reuse oriented (WRO). For tiling, every possible combination of weights, input activation, and output tiles which fits on the available on-chip memory, results in a legal tiling scheme. Finally, for unrolling, the computations are mapped onto parallel PEs to exploit the parallelism in DNN computations. When unrolling computations onto the PE array, some on-chip data movement considerations can help squeeze more efficiency out of the compute array by sharing the data which is already on the array [109, 3]. For example, PEs operating on the same input feature map pixels, but different weights may share the input pixels among themselves over the on-chip interconnect, rather than individually accessing the more expensive and larger on-chip SRAM memory.

The common taxonomy for on-chip computation mapping and data movements defines which datatype remains stationary in the PEs, and which datatypes traverse the array or are called from the on-chip SRAM buffer. For example, the weight-stationary (WS) dataflow allows each PE to have a unique set of weights in its registers, while input feature map pixels are provided over the NoC and flow through the PEs which require them. In an output-stationary (OS) dataflow, the weights and input feature maps can flow through the array while each PE maintains the PSUM of the output pixels it is responsible for, until they are fully accumulated. For input-stationary (IS) dataflows, the input activations remain on the PE while weights flow through the array and output pixels are computed in a distributed manner. For each of the WS, OS, and IS dataflows, there can be a variety of implementations possible, defining which parts of the data are stationary. For example, a variant of OS may allow each PE to work on a subset of output pixels of the *same* output feature map. Alternatively, each PE may handle an entire output channel on its own [56]. In both cases, the dataflow falls under the OS classification, but is implemented differently.

More complex dataflows also exist in literature, most prominent of which is the row-stationary (RS) dataflow, proposed for the Eyeriss accelerator [3]. Each PE is responsible for a 1-D convolution of one row of input activations against one row of the weight kernel. Vertically, the PEs can share their PSUMs to accumulate the results of their 1-D convolutions, essentially achieving the accumulation across the 2-D kernel's window. Weights, output activations and input activations are shared across the on-chip interconnect horizontally, vertically, and diagonally, respectively. Within each PE, the registers may contain the weights and activations of different channels, allowing for more efficient use of the lowest level memory, and performing more accumulations within the same PE with less data movement.

3 HW-SW Co-Design of Deep Neural Networks

Co-designing hardware and software might seem intuitive, yet the challenge of truly integrating both domains in a holistic implementation is often considered or identified late in the design process. Flexibility in low-level, core design choices becomes limited over the course of hardware or software development. Ideally, HW-SW co-design should maintain a coherent view of both domains throughout the time of development. The challenges of co-design also change as further development progress is made. Initially, ideas and fundamental HW-SW design decisions might be exchanged by human experts through specifications and targets that should be achieved in the final deployment. In the next stages, the implementation details of hardware and software might become too complex to discuss in depth through linguistic specifications. Here, functional models and simulations can guide the development and help in evaluating and making design choices in both domains. Towards the last stages of development, large systems composed of many sub-components make up the design. The inter-dependencies among sub-components and sub-systems can become hard to interpret by human experts. Here a combination of functional models and metaheuristics provide the human designer with near optimal choices, without the designer having to be aware of all the low-level details and interactions of the overall system.

In this chapter, the challenges in the hardware and software domains are discussed in the context of DNN deployments. The targets of ML and HW engineers are presented to understand the contrasting perspectives that might emerge in a co-design problem. The techniques and methods which help ML and HW engineers achieve synergies are briefly introduced. Finally, the problem statement of this dissertation is constructed, along with the proposed paths to achieve effective HW-DNN co-design based on different design problem definitions.

3.1 Contradicting Challenges of DNN and HW Design

Identifying the challenges of HW-DNN co-design requires a clear understanding of the goals and constraints in both domains. As an extreme case, we consider the perspectives of a HW-engineer with minimal information about the target DNN and an ML-engineer with minimal information about the target HW.

An ML-engineer is concerned with developing the most performant algorithms in terms of task-related prediction accuracy. This correlates with more model parameters, layer diversity, higher input resolution, and generally a more computationally complex DNN. Predictability

and certainty are also critical to such algorithms in safety critical settings, contributing to more complex algorithms which have redundancy through batch-processing from multiple sensors and/or having an ensemble of classifiers with a voting mechanism to give the final prediction. Naturally, these also increase the computational overhead required for the application. With equal importance, the robustness of the algorithms against adversarial attacks is yet another concern for the ML-engineer, which motivates the introduction of preprocessing stages to the algorithm, adversarial training, redundancy, and multiple input processing, similar to the measures taken for predictability. Counterintuitively, adversarial training and input filtering for improved robustness against adversarial attacks harms the natural task-related accuracy of the DNN.

In contrast, a HW-engineer seeks other targets. The fabrication cost of an application specific integrated circuit (ASIC) is correlated with the complexity and area utilization of the hardware design. Typically, the larger share of an integrated chip's area is consumed by memory cells, where 6 transistors are required per 1-bit of SRAM. This incentivizes designs with smaller on-chip buffers. Smaller and simpler compute logic with fewer registers also helps in lowering the area cost. Hardware design on FPGA has similar challenges, as the utilization of the design may not exceed the LUT, digital signal processing (DSP) blocks, and block random-access memory (BRAM) resources available on the programmable logic. Along with shrinking the on-chip memory to save area, a contradicting objective is to reduce power consumption. Smaller on-chip buffers would require more frequent communication with off-chip DRAM, which is costly in terms of energy consumption and potentially latency, when off-chip communication results in stalls. Lastly, the HW-engineer also has their concerns with respect to safety and security. For example, the HW-engineer may employ dynamic voltage and frequency scaling (DVFS) to save power, but low-voltage operation could result in bit-flips in logic and memory. This could lead to critical errors at the task level. Bit-flips may also be caused by ionization or aging, neither of which can be guaranteed by the designer. Therefore, precautions must be taken through redundancy approaches, which again stress the challenge of minimizing hardware area and power consumption.

From this brief look at the targets of HW and ML engineers, it is clear that each domain is complicated in its own right, with design decisions affecting contradicting targets within the domain itself. A deeper look further reveals cross-domain impacts of the design decisions. Employing an ensemble of classifiers for algorithmic redundancy could lead to increased latency, area, and power consumption on hardware. In cases where DVFS, aging, or ionization cause bit-flips, the accuracy and predictability of the algorithm cannot be guaranteed. Finally, large complex DNNs executing on hardware with small on-chip buffers increases the amount of computation tiling and off-chip communication required, thereby increasing the energy cost per classification. There is a clear motivation for the two domains to consider their own challenges *alongside* the targets of the other domain, in order to reach solutions that ease the design effort in both hardware and software.

3.2 Compromises and Extending the Hand of Truce

Achieving the difficult targets in the hardware and software design domains can be made easier through optimization techniques presented in chapter 2. Quantization, pruning, NAS, loop-tiling,

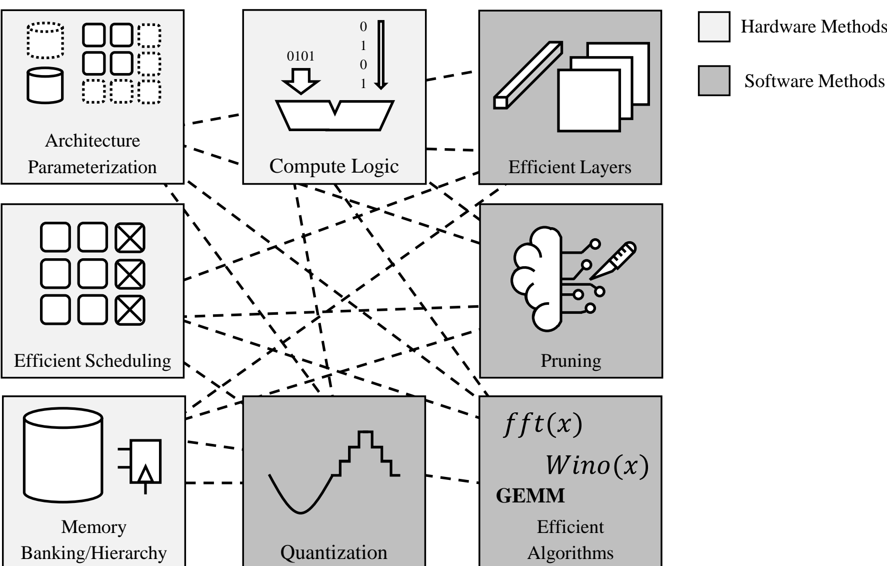


Figure 3.1: Inter-dependencies of hardware and software DNN optimization techniques.

reordering, and unrolling are some techniques that achieve improved deployments of DNNs. However, optimization techniques may be performed without fully extracting the expected benefits.

ML and HW engineers can potentially compromise their own targets to help each other. Yet without properly integrating their optimization techniques, the solution would end up in compromises in both domains with little to no benefits. To elaborate this point three basic examples can be considered.

Pruning and Execution Schedules. Removing parameters from a DNN can be performed at different regularities and with different sparsity targets for each layer. For a CNN, convolutional layers may be pruned in element-wise, channel-wise, or filter-wise regularities. Pruning at a fine regularity which is not supported by the memory access patterns on hardware would result in task-related accuracy degradation without improving the hardware execution metrics. Deciding *which* parameters to prune can be done based on a heuristic, while a search algorithm can decide *how much* to prune each layer, i.e. find the sparsity ratios of each layer. In the hardware domain, the dataflow supported by a generic spatial accelerator may be optimized for weight-dominated or activation-dominated layers, i.e. output-stationary or weight-stationary. A typical image classification CNN has more activations in the initial layers and fewer in the deeper layers. The opposite is true for weights, where initial layers have fewer filters, and deeper layers tend to have more filters and input channels. Overall, the decision on whether to prune initial layers or deeper layers should not only be done to maintain task-related prediction accuracy, but also to benefit the dataflow which the hardware supports. If the dataflow efficiently unrolls the computations of the initial layers, and cannot effectively map the computations of the deeper layers, then the pruning algorithm must take such subtleties into consideration and prune the deeper layers. Pruning layers which are already efficiently executed on the hardware might lead to task-related accuracy degradation without any improvements in execution latency. Pruning a layer can also

change the scheduling and mapping search space. Compute workloads which previously did not fit on the on-chip memory may become possible after pruning, allowing further mapping options on hardware. Therefore, the execution reward returned from the scheduler and mapper of the hardware is also important for the pruning algorithm to decide how much to prune a particular layer. For example, pruning 10 channels or 15 channels may result in the same schedule on a particular hardware design, which does not change the latency of the execution. However, pruning one more channel, e.g. 16 channels, may result in a new tiling scheme becoming possible, which drastically reduces the latency of the execution.

Quantization, Memory and Compute Logic. Similar to pruning, quantization can be decided independently for each layer to provide more numerical precision for some critical layers of the DNN, and low-precision, fast execution for other layers. The numerical precision used to represent weight and activation data can also be different *within* the same layer. A search algorithm can be applied to this problem as well, tasked with finding the optimal bit allocation for each datatype in each layer. Assuming the search algorithm is given the freedom to choose between 1 to 16 bits for each datatype in each layer, the hardware must equivalently be able to extract the benefits at each of those quantization levels. This might involve designing bit-serial PEs, and flexible data-packing in the memory's word-length. The interconnect must also flexibly transport the necessary data at all supported bitwidths, without under-reads or over-reads. If data-packing and memory alignment allocates 32 bits for 8-bit parameters, the benefits of quantization for memory movement are not achieved. Similarly, if the arithmetic unit performs the same operation with the same latency and throughput for all bitwidths, no computation speed-up is achieved. Finally, similar to pruning, quantization might unlock new legal schedules which fit on the on-chip memory, thereby directly influencing the latency and power due data movement, as well as the expected benefits at the compute level. In terms of a vectorized PE, which might support a subset of quantization levels, the search algorithm must be constrained to make decisions which are supported by the hardware. The effects of each decision on the dataflow, mapping, and schedule must be fed back as signals to the search algorithm, guiding it to choose strategies which truly benefit the target hardware's capabilities as well as maintain a high task-related accuracy.

Adversarial Robustness and Bit-Flip Resilience. Considering a safety-critical deployment of DNNs, different threat models must be anticipated by ML and HW engineers. Adversarial attacks can be seen as algorithmic threats which exploit internal paths of a DNN to produce high-confidence, incorrect predictions. To improve the robustness of a DNN against adversarial input perturbations, the ML engineer can introduce adversarial examples during the training, allowing the DNN to learn such input-based threats. Differently, the HW-engineer must consider the consequences of hardware-based errors, which may occur due to low-voltage operation, aging, or exposure to radiation. Protection against such hardware errors or bit-flips can be achieved by introducing redundant hardware modules and/or redundant computations, both of which are very costly in terms of latency, power, and potentially area. In essence, both sides are attempting to achieve the same goal, which is the correct operation of the DNN in the presence of errors or perturbations. However, by improving the robustness of the DNN against adversarial examples, its behavior under hardware-based bit-flip errors is affected. The HW-engineer's target is not necessarily to eliminate *all* errors, but to provide sufficient redundancy such that a reasonable amount of computation errors can be overcome by the DNN's internal algorithmic redundancy. If the ML-engineer considers the effect of bit-flips during the adversarial training scheme, a

solution which reduces the effort in hardware redundancy might be achieved. In contrast, a DNN which is adversarially trained without any consideration to hardware errors might be too sensitive to internal computation bit-flips, making the targets of the HW-engineer significantly more challenging to achieve.

The three examples represent cases where working on two domains separately can lead to sub-optimal compromises, but working jointly on solutions makes the overall deployment meet its targets while reducing the individual efforts of HW and ML engineers. Figure 3.1 shows more dependencies that exist between hardware and software DNN optimization techniques.

3.3 Problem Statement and Paths to Effective Co-Design

The challenges, arguments, and examples presented in the previous sections can be boiled down to the following problem statement:

> AI on edge necessitates tightly-coupled HW-SW co-design. Isolated efforts of HW and SW optimization cannot fully capture the potential of efficient inference. This leads to sub-optimal compromises in HW or SW design, without reaping the expected benefits. To achieve true co-design, multiple paths towards HW-awareness and optimization must be laid out parallel to SW design and development. The different co-design paths should be applicable to different stages of development and the current design challenge.

In this dissertation, three core concepts inspired by the field of very-large-scale integration (VLSI) design are presented and adapted to different HW-DNN co-design problems:

- **Co-Design Methodologies.** Defining multiple paths towards HW-DNN co-design for a plurality of optimization methods and design challenges.
- **Executable Models.** Developing effective HW-awareness injection techniques through models, to facilitate the task of DNN optimization and provide tangible benefits in the final deployment.
- **Abstraction Levels.** Defining multiple abstraction levels to facilitate co-design at different stages of development and provide a divide-and-conquer approach to tackle large HW and DNN design spaces.

The three concepts are essential to the works presented in the next chapters. A summary of all works under the scope of this thesis and their use of the three VLSI-inspired concepts is shown in figure 3.2.

3.3.1 Co-Design Methodologies

The need for different paths to achieve co-design can be justified by understanding how co-design challenges look like at different stages of development. Design problems range from being abstract to well-defined in different development phases. Accordingly, this work classifies three methods to co-design: handcrafted, semi-automated, and fully-automated co-design.

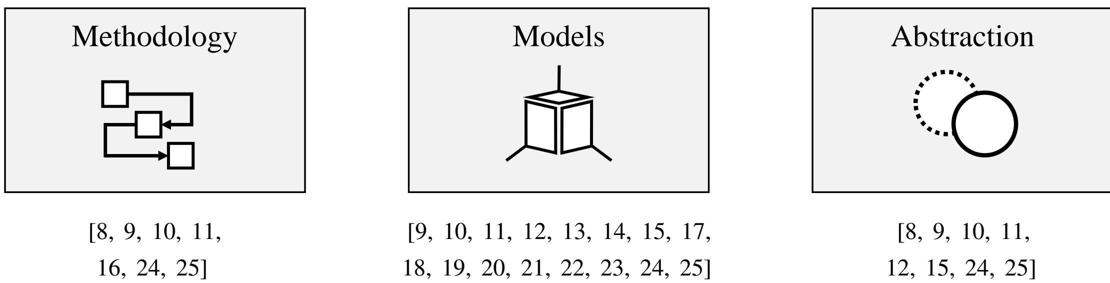


Figure 3.2: Works published under the scope of this thesis categorized with respect to concepts used from the VLSI design domain.

Handcrafted co-design involves human-experts reaching an agreement on design decisions based on an understanding of complex theoretical concepts and analytical thinking. These are design decisions that may be difficult to formulate into a search problem or define for an agent to solve. An example would be designing low-level hardware description language (HDL) components tailored for a specific DNN operation [8], or understanding the effect of adversarial training and bit-flip robustness to develop improved DNN training schemes [16]. The ML and HW human-experts must exchange ideas succinctly to reach a solution through this methodology. Naturally, handcrafted co-design is not feasible for large-scale, search oriented problems. Chapter 4 presents two works using this methodology [8, 16].

In semi-automated co-design, components developed through handcrafted methods may be integrated into larger systems in a semi-automated manner [25, 24]. A system might be largely represented with well-defined models that can be automatically optimized, while other decisions must be made by a human-expert. For example, the DNN's computation graph can be implemented as a pipeline of computation blocks. These can be represented as a synchronous data flow (SDF) model, where the computation blocks are actors communicating their results over FIFO channels. This system may be developed and automatically optimized by high-level synthesis (HLS) pipeline functions, however, the human designer might still need to decide what throughput should be targeted with the pipeline and what resource utilization must be maintained. The human designer may also allocate more compute resources to bottleneck actors. Based on this, the allocation of memory in the communication channels can be automatically chosen. Chapter 5 covers Binary-LoRAX [25] and BinaryCoP [24], which incorporate semi-automated co-design.

Finally, a fully-automated design methodology requires little to no intervention from a human designer. This is only feasible when the system can be represented by well-defined, predictable, executable, high-fidelity, parameterizable models. It is also important to assert that there are no errors or corner cases in the executable models, which might allow the automated design agent to make decisions which cannot be realized in practice. The automated design agent can be a GA, an RL agent, or any similar metaheuristic or search agent. As the automated design agent traverses the design space with the help of the parameterizable, executable models, it finds the parameter settings which achieve the hardware and software design goals. This methodology is particularly effective towards the late stages of development, when reliable, well-defined, executable models

are available. In chapter 6, HW-FlowQ [10] and AnaCoNGA [9] are discussed as examples of fully-automated design frameworks in the scope of this work.

3.3.2 Executable Models

As HW and ML engineers try to meet their co-dependent goals, continuous integration of their work needs to be evaluated. In this context, models can form a bridge between the two domains, where sufficient hardware and software details can be exchanged without hindering the independent, complex development cycles in either domain. Apart from integration and testing, effectively arriving at a solution that meets an application's constraints typically requires exploration of a large and complex HW-DNN solution space. This motivates the design of lightweight, easily reconfigurable models, which can be used to evaluate design choices in these complex design spaces. Such models allow the continued development of the hardware *parallel* to the DNN design phases, without having to finalize and/or synthesize the hardware in the early stages of development.

Executable models of DNN hardware accelerators can be constructed based on the deterministic execution of their graph structures. Considering a spatial accelerator, a particular tiling and unrolling strategy can be translated into a precise schedule which provides high-fidelity latency and energy estimations to guide the design decisions of human experts and automated design agents or metaheuristics. These models do not necessarily have to be cycle-accurate representations of the hardware, but rather *analytical* models which can provide the total number of memory accesses and computations required to complete a workload with respect to a given scheduling strategy. Once the execution schedule is known, the cost (latency and/or energy) of every action (memory read, write or computation) can be multiplied by the total number of times that action will take place, producing the estimates. Such models were developed and used in publications within the scope of this dissertation [9, 10, 11, 12, 13]. A graph-based, dataflow accelerator can be represented with a more traditional SDF model, which can also return execution metrics to the design methodology. Several works within the scope of this dissertation used such models [22, 25, 24, 23].

For black-box or general-purpose, non-deterministic hardware, a LUT-based approach may be considered. A large number of relevant computation workloads can be executed on the hardware, and the resulting execution metrics, e.g. latency, throughput, or power, can be collected in a LUT. This can then be used in optimization loops of the DNN, where neural architecture, pruning or quantization decisions can be evaluated based on the pre-collected performance measurements. Therefore, the LUT can guide the optimization and design of the DNN [19, 20]. The measurements of a LUT can also be used to train a regression model, which eventually can predict the execution metrics of unseen workloads or hardware configurations. The regression model can also be formulated in a differentiable manner, e.g. a Gaussian process regressor, which can then be integrated *directly* into the training scheme of the DNN and help the SGD algorithm directly optimize the weights for the task-related accuracy, as well as, pruning masks or quantization levels for hardware performance [15]. LUT and regression-based hardware models were used in published works [19, 20, 15] contributing to this dissertation.

When using models for design space exploration, it is critical to assert their estimation fidelity. A high-fidelity model may even be preferred to an accurate model in cases where choosing the

best solution is more important than measuring the performance of a single solution. Particularly for automated design agents and metaheuristic search techniques, a model with low-fidelity might heavily misguide the direction of search space traversal, ending up in optimal solutions with respect to the model, but sub-optimal implementations on real hardware. The speed of the model execution is also important. Design space exploration loops for DNNs already suffer from long GPU-hours for training and accuracy evaluation of neural network configurations. Adding further delays to consider the hardware design exacerbates this issue and would severely extend the search time. However, a fast executable model might be used to eliminate hardware-inefficient DNN configurations early, before GPU training and evaluation, thereby shortening the overall search time *and* injecting hardware-awareness to the design exploration loop [9].

3.3.3 Abstraction Levels

Viewing complex systems at different levels of abstraction is fundamental to design methodologies in the field of VLSI. The divide-and-conquer approach to manage design complexity can be achieved by viewing only a limited set of design details at a time, allowing the design process to systematically consider simpler problems at each level of abstraction before moving on to the next. In the VLSI domain, the Gajski-Kuhn diagram defined different *views* to a system and multiple abstraction levels with respect to each view [111]. Based on this diagram, design transitions are defined, such as synthesis, implementation, abstraction, refinement, optimization and others, which allow the design phase to move through abstraction levels and design views in different ways. Figure 3.3 shows the Gajski-Kuhn diagram as well as the transitions mentioned to traverse the views and abstraction levels. With these transitions, the design *flow* might take a top-down, bottom-up, or meet-in-the-middle approach. A top-down design flow implies that the design starts at a high abstraction level and slowly gets refined as more components are developed. The opposite is true for a bottom-up design flow, where engineers can work on low-level components first, and then integrate them into larger, more complex systems. Finally, a meet-in-the-middle approach allows two (groups of) designers to work in tandem at the system level and the logic or register-transfer level (RTL). The system designers can specify the high-level architectural targets, while the component designers implement the modules required for the target architecture, eventually meeting each other midway in the levels of abstraction.

In this work, these concepts from the VLSI domain are reintroduced in the context of HW-DNN co-design [10, 11]. The divide-and-conquer approach can be useful in maintaining reasonable complexity in the large search spaces of hardware design, NAS, and DNN compression. The design of DNN accelerators can also benefit from the levels of abstraction, where a processing element can be designed and optimized to support some necessary DNN operations and then replicated and integrated into larger designs, such as dataflow accelerators. This transition in abstraction levels was performed during the course of this work, from designing OrthrusPE [8] to then injecting it into larger hardware designs in Binary-LoRAX and BinaryCoP [24, 25]. Combining abstraction levels and executable models is also possible, as demonstrated in section 6.1 based on the work in [10] and [11]. For example, a high abstraction level execution model of a spatial accelerator may first consider ideal conditions, large on-chip buffers and 100% PE utilization (optimal loop-tiling and unrolling). Such a model can help in determining the theoretical ideal case of execution at an early design stage. Then, constraints on the interconnect and the on-chip

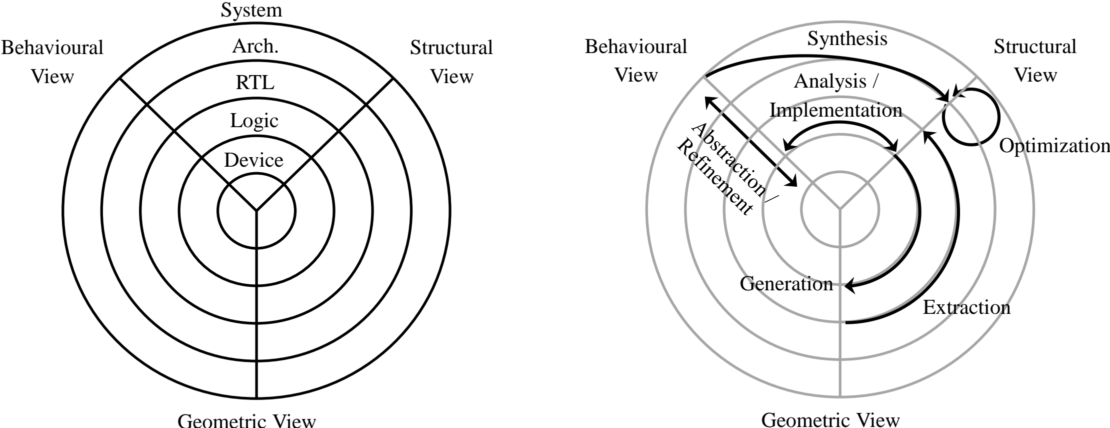


Figure 3.3: The Gajski-Kuhn diagram (left) and the possible transitions to traverse the views and abstraction levels (right).

buffer size may be introduced after *refinement*, helping the designer understand the loop-tiling characteristics of the DNN workloads with respect to said constraints. Finally, a more refined abstraction level can further consider the PEs' arithmetic capabilities, register sizes, and communication patterns, and implement the entire execution schedule, returning more accurate estimates of latency and energy consumption. The next chapters showcase six examples where these concepts are used in handcrafted, semi-automated, and fully-automated design methodologies.

4 Handcrafted Co-Design

DEEP understanding of hardware and software enables human-experts to craft tailored solutions for complex co-design problems. This is particularly useful in cases where a design challenge is conceptual and hard to concretely define mathematically or formulate into a search space for algorithms or metaheuristics to solve. The designers' conceptual understanding of the challenge is itself the problem formulation, which can be solved by applying their knowledge, expertise, and creativity. In this chapter, two examples of handcrafted co-design are presented. In OrthrusPE [8], a complex form of BNNs is considered, which replaces expensive multiplication and addition operations with hardware-friendly XNOR and popcount operations. However, to maintain high task-accuracy, the BNNs still require some fixed-point arithmetic operations. This motivates the conception of a PE which supports both fixed-point and binary operations, with minimal hardware overhead. In *Mind the Scaling Factors* [16], hardware and software threat models are investigated in the form of on-chip bit-flips and input-based adversarial attacks. By understanding the theoretical worst-case effect of a bit-flip in the numerical representation on hardware, the neural network training hyperparameters are tuned to improve adversarial robustness *and* bit-flip error resilience. In both works, understanding the conceptual design challenge or the theoretical aspects of the execution led to the human-engineered, handcrafted co-design solutions.

4.1 OrthrusPE: Runtime Reconfigurable Processing Elements for Binary Neural Networks

Recent advancements in BNN training methods and architecture search have pushed the prediction accuracy of single-bit neural networks closer to their full-precision counterparts [62, 34]. These advancements were brought about by introducing additional computations, in the form of scale and shift operations in the fixed-point numerical domain and convolutions with multiple weight and activation bases in the binary domain [62]. OrthrusPE[1] is a runtime reconfigurable PE which is capable of executing all the operations required by modern BNNs without introducing large resource utilization overheads or sacrificing power efficiency. More precisely, the DSP48 blocks on off-the-shelf FPGAs are used to compute binary Hadamard products (for binary convolutions) and fixed-point arithmetic (for scaling, shifting, batch-norm, and non-binary layers), thereby utilizing the same hardware resource for two distinct, critical modes of operation. The experimental results show that common PE implementations have 67% higher dynamic power consumption, while requiring 39% more LUTs, when compared to an OrthrusPE implementation.

[1] In Greek mythology, Orthrus is a two-headed dog.

4.1.1 BNN Training Challenges and Motivation for Reconfigurable PEs

The memory and compute advantages of data quantization have made it an essential compression technique in edge DNN deployment scenarios [35, 36]. Floating-point weights and activations are superfluous for most inference tasks, making low-bitwidth fixed-point representations an attractive alternative [57]. However, as the number of bits for a fixed-point representation decreases, accuracy degradation ramps up and more of the critical, learned information is lost. Nonetheless, research into fully binarizing DNNs has flourished in recent years, producing training schemes and BNN architectures which achieve accuracy target that rival high-precision implementations [60, 112, 61, 62, 34].

Different implementations of BNNs exist, with binary weights *or* binary activations, as well as binary weights *and* activations. Intuitively, having both weights and activations in the binary format leads to the highest loss of information, but provides the most memory, energy, and compute efficient solution. The billions of multiplications typically required for forward passes are reduced to simple XNOR logic operations, while the accumulations are implemented as popcounts [34]. The main issue with such a low information representation is that the cumulative effect of the finely-tuned, learned weights of the network is lost. Early attempts at realizing fully binarized DNNs resulted in accuracies far below those achieved by state-of-the-art DNNs on complex problems such as ImageNet [60]. To tackle this problem, Lin et al. [62] approximated activations and weights using *binary bases* to create accurate binary convolutional networks (ABC-Nets). Using this method, BNNs have achieved accuracies only 4-5 p.p. below their full-precision counterparts for Top-1 and Top-5 results on ImageNet. This method elaborated in section 4.1.3.

Although binary bases present a solution to the information loss problem, they also introduce new operations to be executed on hardware. Furthermore, the first layer of the neural network is critical and it maintains fixed-point values for weights and activations to avoid severe loss of information at the input of the network. Lastly, the effect of batch normalization on improving the accuracy and training time of BNNs makes it an essential layer that must be supported by the hardware [112].

OrthrusPE provides a hardware solution applicable to virtually all[2] Xilinx FPGAs for accelerating accurate binary neural networks, without employing additional hardware for the non-binary operations that need to be performed at intermediate stages. Off-the-shelf FPGAs with DSP48 or DSP58 blocks can be reconfigured at runtime to compute the highly parallel binary Hadamard products required for binary convolution operations, as well as the fixed-point operations that occur intermediately. The synthesis and implementation results of OrthrusPE are compared against other configurations with equivalent throughput. The method is orthogonal to dataflow optimizations and the overall accelerator architecture, and can be implemented as an extension to any existing, compliant FPGA accelerator.

The contributions of this work are summarized as follows:

- Developing a flexible computation unit to accelerate a wide range of BNNs (e.g. table 4.1) and executing SIMD-based binary Hadamard product operations on FPGA hard blocks.

[2] All 7-series, Ultrascale and Ultrascale+ FPGAs, Zynq SoCs (DSP48E1 and DSP48E2), and the recent Versal platform (DSP58). The entry level Spartan-6 presents the only exception (DSP48A1).

- Reusing FPGA hard blocks to create novel, runtime reconfigurable processing elements, which dynamically support binary and fixed-point computations.
- Formalizing the relationship between computation mode switching and partial result memory for BNN layers with multiple binary bases.

4.1.2 Related Work

Several accelerators have been developed with processing elements designed to exploit performance boosts due to variable quantization levels [69, 67, 68, 66]. Other accelerators were designed to solely execute BNNs [113, 114, 103].

Bit Fusion [69], UNPU [67], Stripes [68], and Loom [66] are all based on ASIC designs. UNPU, Stripes and Loom offer single-bit operations while the Bitbricks structure used in Bit Fusion allow for the execution of operations at fixed quantization levels, making the smallest possible precision bounded by the size of a single Bitbrick. UNPU, Stripes and Loom are capable of performing both binary and fixed-point operations, however, with a non-negligible overhead due to the support of variable quantization levels. Further ASIC-based works, BRein [114] and YodaNN [113], were developed precisely to accelerate BNNs. However, they do not implement the binary bases required for accurate binary nets.

FINN [103] is a popular framework for accelerating BNNs on FPGAs. The framework is geared towards BNNs in [60]. FINN compiles HLS code from a BNN description to generate a hardware implementation. FINN is not compatible with multiple binary bases, but rather simpler BNNs suited for problems such as MNIST, CIFAR-10 or SVHN. Other FPGA-based BNN accelerators [115, 116, 117] also execute binary operations purely on LUTs and utilize DSPs for fixed-point operations, where they are supported.

The authors of Double MAC [118] extract more functionality from FPGA hard blocks. The signals are preconditioned before entering the DSP blocks such that two multiplications can be obtained with some post-processing. This leverages quantization, since the two results obtained at the output are calculated from operands that are smaller than the maximum possible precision supported by the DSP. The work virtually turns DSPs into SIMD multipliers. Similarly, OrthrusPE utilizes DSPs as SIMD binary Hadamard product processing units. Double MAC is orthogonal to OrthrusPE, making it possible to include Double MAC as a *third* operating mode.

Efficient exploitation of hard blocks on FPGAs can play a key role in lowering the efficiency gap between ASIC and FPGA implementations [119]. This is evident in the recent trend of FPGA manufacturers adding more hard blocks to their chips aimed at accelerating DNN applications [120].

4.1.3 Accurate Binary Convolutional Neural Networks

To identify the operations that must be supported by PEs, the basic building blocks of accurate BNNs are presented in this section. Different operations of BNNs proposed in literature are summarized in table 4.1. The weights and activations of a BNN are constrained to $\{-1,1\}$. In the hardware implementation, the '-1' values are mapped to '0', allowing the execution of multiplication and accumulation as hardware-friendly XNOR and popcount operations during

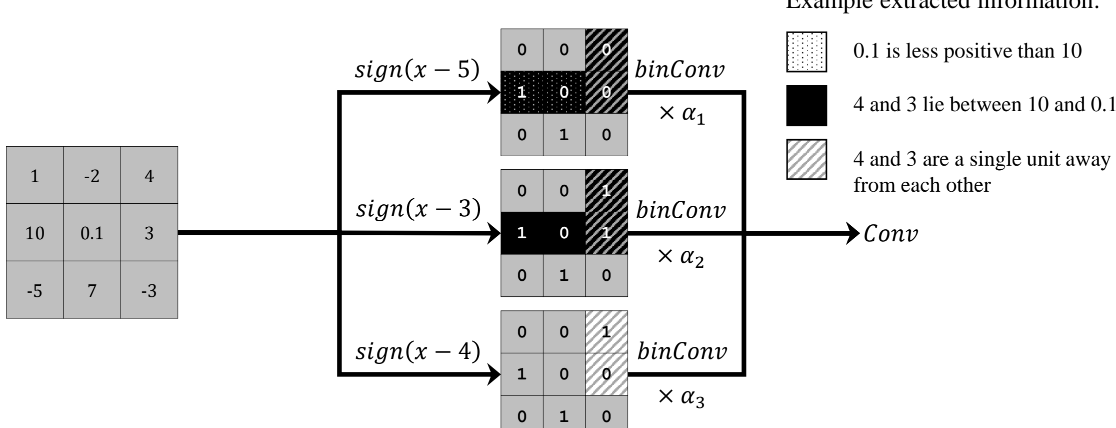


Figure 4.1: Binary bases can differentiate the values of the full-precision kernel more accurately by preserving more information through linear transformations.

inference. Throughout this section, this affine transformation is considered and the binary numerical space is denoted with $\mathbb{B} = \{0,1\}$.

Accurate BNNs use *multiple* weight and activation bases to approximate a full-precision layer, which reduces the gap in prediction accuracy between the two implementations. Different to simpler BNNs where values are binarized using the $sign()$ function (recall equation 2.7), Lin et al. [62] presented a solution where each base is produced by scaling and shifting the original values to different degrees before binarization. This produces multiple unique binary bases which preserve more information collectively due to the additional operations performed before obtaining them. Figure 4.1 shows a simple example of a 3×3 kernel binarized into 3 bases. Within one base, it is possible to differentiate whether a value is positive after a shift or not. Shifting 0.1 and 10 by -5, leads to 0 and 1 binarization respectively, indicating 0.1 is smaller than 5 and 10 is larger. Combining two bases captures information on how certain values fall between other values, for example, 3 and 4 must lie between 0.1 and 10, since the 0.1 and 10 remained 0 and 1 respectively after shifting by -5 *and* -3, while 3 and 4 were binarized to 0 after shifting by -5 and turned to 1 in the binary base where they were shifted by -3. Finally, combining all three bases, we can extract that the difference between kernel elements 4 and 3 is 1 to 2, as both were binarized to 0 when the shift was -5 or both to 1 when the shift was -3, but they had different binarization when the shift was by -4.

To discuss the complexity of a convolution operation with binary bases, we recall the notation presented in section 2.1. $\mathbf{A}^{l-1} \in \mathbb{R}^{X_i \times Y_i \times C_i}$ is an activation tensor of a convolutional layer $l \in [1, L]$ in an L-layer CNN, where X_i and Y_i indicate the input spatial dimensions, and C_i is the number of input channels. The weights $\mathbf{W}^l \in \mathbb{R}^{K_x \times K_y \times C_i \times C_o}$ are the trainable weights of the layer. The sign, scale, and shift functions are used to find an appropriate binarization for $\mathbf{A}^{l-1}$ and approximate it into $\mathbf{H}^{l-1} \in \mathbb{B}^{X_i \times Y_i \times C_i \times N}$, having N binary bases. Similarly, $\mathbf{W}^l$ is approximated as $\mathbf{B}^l \in \mathbb{B}^{K \times K \times C_i \times C_o \times M}$, where M is the number of weight bases. Equation 4.1 represents the multi-base binary convolution.

Table 4.1: Requirements of most common binary neural networks and the respective hardware operations for execution.

Method	Binary Weights \Activations	Weight Scale α_m	Activation Scale β_n	Batch-Norm	Multiple Weight Bases M	Multiple Activation Bases N
BNN [60]	✓	✗	✗	✓	✗	✗
XNOR-Net [34]	✓	✓	✓	✓	✗	✗
CompactBNN [121]	✓	✗	✓	✓	✗	✓
ABC-Net [62]	✓	✓	✓	✓	✓	✓
Hardware Operation	XNOR-Popcount	Multiplication	Multiplication	Multiplication-Shift	MAC	MAC

$$\mathbf{A}^l = \text{Conv}(\mathbf{B}^l, \mathbf{H}^{l-1}) \tag{4.1}$$

Equation 4.2 demonstrates the binary convolution using sub-tensors $\mathbf{B}^l_m \in \mathbb{B}^{K_x \times K_y \times C_i \times C_o}$ and $\mathbf{H}^{l-1}_n \in \mathbb{B}^{X_i \times Y_i \times C_i}$, where m and n indicate a single base in M or N respectively. The scaling factors α_m and β_n are trainable parameters to extract more information from each binary base representation, however, they introduce fixed-point operations to the binary convolution.

$$\mathbf{A}^l = \sum_{m=1}^{M} \sum_{n=1}^{N} \alpha_m \beta_n \texttt{BinConv}(\mathbf{B}^l_m, \mathbf{H}^{l-1}_n) \tag{4.2}$$

Consider a binary weight tensor slice $b^l \subset \mathbf{B}^l_m$, where $b^l \in \mathbb{B}^{K_x \times K_y}$. The activation slice $h^{l-1} \subset \mathbf{H}^{l-1}_n$, where $h^{l-1} \in \mathbb{B}^{X_i \times Y_i}$ is defined accordingly. Equation 4.3 shows the XNOR operation performed on b^l and h^{l-1}, which is the core binary operation. The XNOR operations are grouped as binary Hadamard products of the sliding kernel windows. Next, the partial sum p_{m,n,c_i} is the accumulation of the intermediate XNOR operations over the kernel dimensions $K_x \times K_y$.

$$\begin{aligned} \underbrace{p_{m,n,c_i}}_{\text{integer}} &= \text{PopCnt}(\text{XNOR}(b[k_x][k_y], h[x_i + k_x][y_i + k_y])) \\ &= \sum_{k_x=1}^{K_x} \sum_{k_y=1}^{K_y} \text{XNOR} \underbrace{(b[k_x][k_y], h[x_i + k_x][y_i + k_y])}_{\text{binary}} \end{aligned} \tag{4.3}$$

Finally, to compute a single output pixel $a_{m,n}$ relative to bases m and n, the popcount values p_{m,n,c_i} need to be accumulated across the input channels C_i , as shown in equation 4.4.

$$a_{m,n} = \sum_{c_i=1}^{C_i} (p_{m,n,c_i}) \tag{4.4}$$

4.1.4 OrthrusPE

OrthrusPE is a processing element which operates in two modes, a binary and a fixed-precision mode. In the binary mode, OrthrusPE executes SIMD binary Hadamard products and the

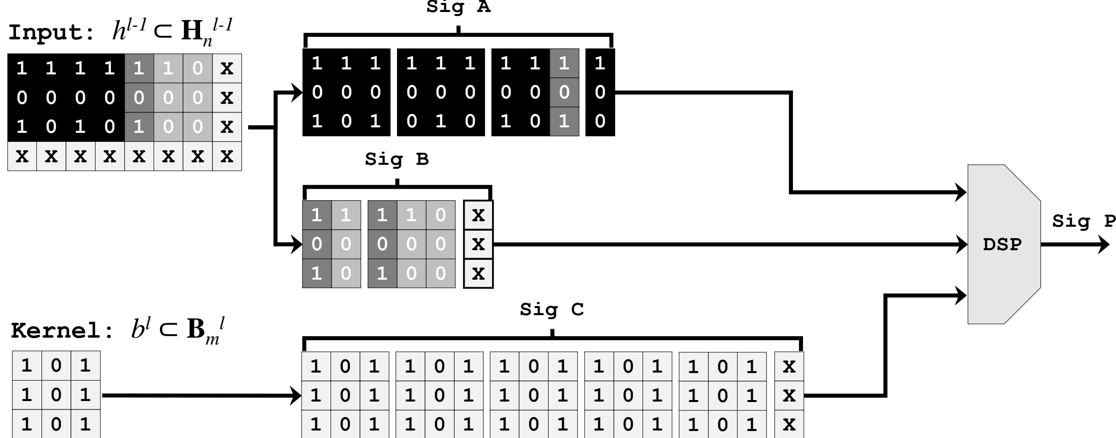


Figure 4.2: Preconditioning signals A, B and C to compute five 3× 3 Hadamard products. Pixels represented with an `X` are not relevant for this cycle of operation.

subsequent popcount-accumulation operations in equation 4.3. The fixed-precision mode is activated for the scaling and accumulation operations shown in equation 4.2 and equation 4.4 as well as shifting, batch normalization, and non-binary layers. OrthrusPE leverages a single DSP which is reconfigured at runtime to achieve efficient execution in both modes of operation.

For efficient pipelining and in cases where power and resource utilization are less critical, we propose a static variant of OrthrusPE. Here, two DSP blocks are instantiated in each PE. The first DSP is fixed in the binary mode while the second operates as a typical DSP for executing the fixed-point operations. This implementation is referred to as OrthrusPE-DS (Dual-Static).

4.1.4.1 SIMD Binary Hadamard Product in Binary Mode

In the example dataflow shown in figure 4.2, OrthrusPE receives two tensor slices, h^{l-1} and b^l. When the kernel b^l is slid over the partial input feature map h^{l-1}, a new tensor slice from $\mathbf{H}_n^{l-1}$ can be read from the memory.

The DSP48 (DSP48E1) slice is presented in figure 4.3. The concatenation of signals A and B is used to fit part of the tensor slice h^{l-1} and select it through the X multiplexer, forming a 48-bit wide signal. The DSP's multiplier needs to be set to its dynamic mode to allow the use of the concatenated A:B signal, as well as the individual A and B signals during regular multiplication. The multiplier in the DSP is asymmetric, as signals A and B are 30 bits and 18 bits, respectively. These signals are preconditioned before entering the DSP, such that their concatenated value represents multiple sliding windows of h^{l-1}. The tensor slice b^l, in the form of signal C, is available at multiplexers Y and Z (as well as W on DSP48E2). Signal C is preconditioned to hold the b^l kernel, which is to be operated with the h^{l-1} pixels. By setting the ALUMODE signal and activating the correct multiplexers through OPMODE, the DSP is essentially transformed into a SIMD binary Hadamard product module. Note that there exist multiple combinations of ALUMODE and OPMODE states that provide the same result.

Considering that the most frequently occurring kernel size in modern CNNs is 3×3 pixels [28], each cycle generates five individual Hadamard products at the output of the DSP. In this case,

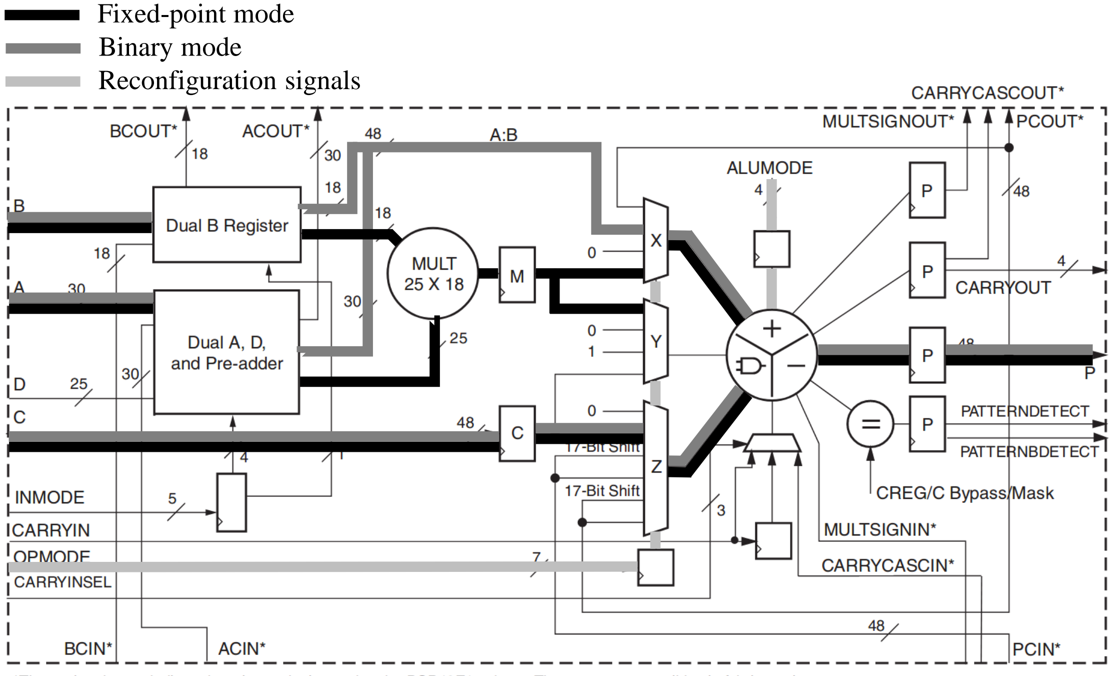


Figure 4.3: The DSP48E1 Slice [1]. Appended bold paths illustrate the relevant signals for our operating modes.

each Hadamard product requires 9 bits out of the 48 bits in signal C and signal A:B, leaving 3 unused bits after the calculation of the 5 results. However, this still presents a high utilization of 94%. This is due to the fact that we can fully fit $\lfloor \frac{48}{K_x \times K_y} \rfloor$ Hadamard products in a single DSP cycle's output, where $K_x \times K_y$ is the number of bits per Hadamard product. The solution to always retain a utilization of 100% is to allow partial operations to take place in each cycle, while small additional logic rearranges the successive results before being processed by the popcount logic. In this manner, any arbitrary window size can also be implemented with 100% utilization of the DSP's processing bitwidth. Intuitively, for FC layers, the utilization is always 100%. Figure 4.4 shows the DSP utilization for two possible configurations of OrthrusPE.

Preconditioning the signals, concatenating them, and performing the wide XNOR operation does not infer a DSP slice in the synthesis tool, but rather generates a regular LUT solution. Therefore, the DSP slice was instantiated manually and the signals were explicitly passed to the module.

4.1.4.2 Arithmetic Operations in Fixed-Precision Mode

The analysis of accurate BNNs in section 4.1.3 revealed the need for regular fixed-point arithmetic operations. As shown in equation 4.3, after a binary Hadamard product is calculated, its popcount reduces the result to a single integer value. All subsequent calculations required for an output pixel which involve this popcount are carried out in fixed-point and/or integer arithmetic. This makes full-reliance on binary operations infeasible. Even in naive, inaccurate BNNs, some critical

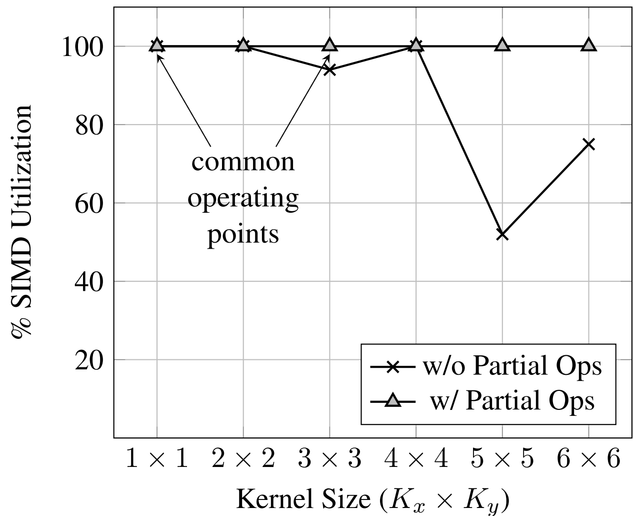


Figure 4.4: SIMD register utilization of the DSP48 in OrthrusPE, with and without partial operations.

layers, such as the input layer and batch normalization layers, require fixed-point arithmetic. The additional scaling operations in accurate BNNs also introduce fixed-point multiplications, which are most efficiently executed on DSPs. This is the motivation for reconfiguring the DSP back to its regular operation mode, by resetting the ALUMODE and OPMODE signals at runtime. With this solution, we exploit the same hardware resource for two distinct modes of operation.

The work in Double MAC [118] can be appended to OrthrusPE, giving it a further mode to operate in. This is particularly useful for accumulating popcounts, which have a smaller bitwidth compared to the fixed-point values used for the non-binarized inputs of the network and the batch normalization layers.

Figure 4.5 shows a schematic of OrthrusPE's internal components. The Bin_mode register holds a flag indicating the mode of operation. The value of Bin_mode influences the DSP_Reconfig signals ALUMODE and OPMODE, which are fed into the DSP to reprogram it as shown in figure 4.3. Bin_mode also functions as a selector for 3 multiplexers (A_MUX, B_MUX and C_MUX), allowing the input feature map pixels, weight pixels and partial sums to be passed directly to the DSP (fixed-point mode) or taken after preconditioning them for the binary Hadamard product operation as shown in figure 4.2 (binary mode). The result produced from the DSP is passed to another multiplexer, where it can be written directly to the partial sum register or postprocessed with a popcount operation, then written to the register.

4.1.4.3 Mode Switching and Partial Sum Accumulation

While in binary Hadamard SIMD mode, OrthrusPE can generate five 3× 3 Hadamard products per cycle. These products are passed to popcount logic, generating five integer values. Referring back to section 4.1.3, each of the M binary weight bases needs to be convolved with each of the N binary activation bases. A single input activation channel and a single weight filter channel produce $M \times N$ partial sum maps. Those $M \times N$ maps need to be scaled by α_m and β_n, then

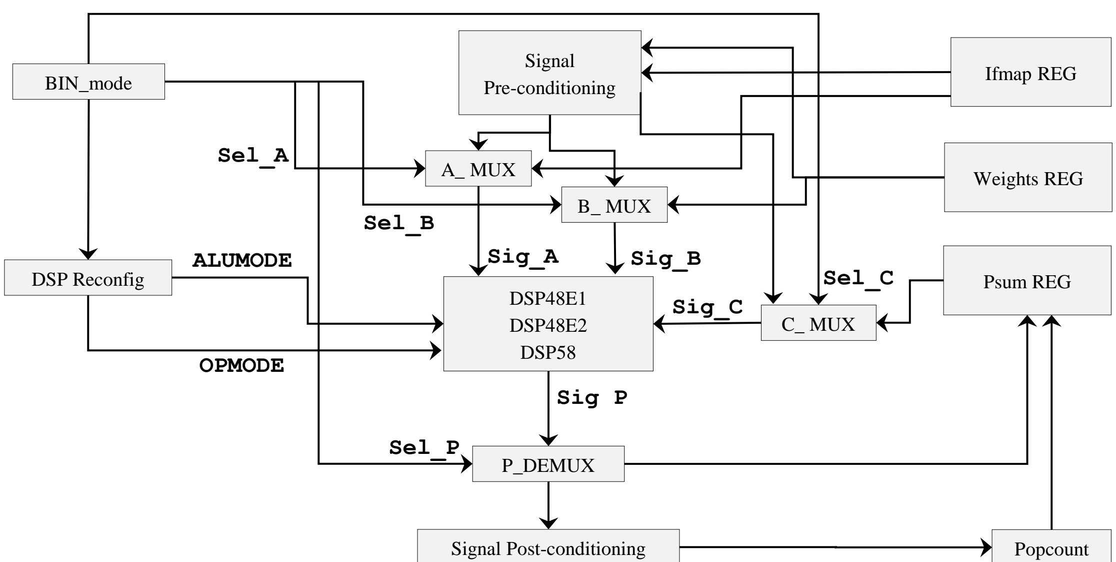


Figure 4.5: Block diagram showing the main components of the OrthrusPE

collapsed into a single partial sum map. The $M \times N$ maps represent parasitic partial sums which will require a considerable amount of memory if not accumulated for an extended time during execution. To minimize this, OrthrusPE can perform $P \times M \times N$ Hadamard products, where P is a set of p_{m,n,c_i} pixels (recall equation 4.4), then switch to its MAC mode for scaling and accumulation, before moving on to another spatial region of the map. Decreasing P reduces the required memory for parasitic partial sums as shown in equation 4.5. In the last term, the kernel dimensions dictate the popcount's bitwidth along with an added sign-bit.

$$Mem_{psum} = N \times M \times P \times (\lceil log_2(K_x \cdot K_y) \rceil + 1) \tag{4.5}$$

The trade-off is that accumulating the $P \times M \times N$ popcounts requires switching the mode of OrthrusPE more often. The number of mode switches per input channel map is expressed in equation 4.6. X_o and Y_o are the dimensions of a single output channel. Since it is possible to switch the ALUMODE after 1 cycle of operation for non-pipelined DSPs, this trade-off does not represent a large overhead and can be exploited to reduce partial result memory requirements in an accelerator.

$$SwitchCount = 2 \times \left\lceil \frac{X_o \cdot Y_o}{P} \right\rceil - 1 \tag{4.6}$$

P can be chosen with some analysis using equation 4.5 and equation 4.6. Figure 4.6 shows the effect of P on partial result memory and switch count for a single input channel of binary ResNet18 [28] in each of its convolutional layers, with $M = N = 3$. The layers are grouped based on their output spatial dimensions.

Layer 1 is not considered as it is not binarized. The actual scratchpad size depends on other factors such as dataflow, loop unrolling and loop interleaving. The analysis shown is only with respect to the minimum requirement necessary for a basic dataflow which maintains the partial

results within a PE until a single input channel is completely processed against a single filter kernel. Layers $2-5$ should dictate the memory requirements as they generate the highest volume of parasitic partial results. This is due to their output dimensions requiring 56×56 pixels per channel.

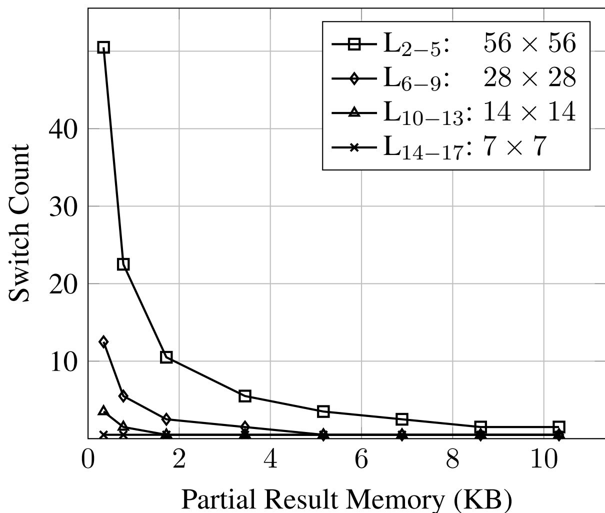


Figure 4.6: Switch count and partial result memory analysis for a single input channel from different convolutional layers of binary ResNet18, with $M = 3$, $N = 3$. Each point represents a different configuration of P.

4.1.5 Evaluation

4.1.5.1 Experimental Setup

The proposed implementations in section 4.1.4 (OrthrusPE and OrthrusPE-DS) are compared to two implementations with equivalent functionality. Typically, BNN processing elements employ two or more distinct types of resources for the operations described in table 4.1. On FPGAs, the straightforward approach is to map all binary operations to LUTs and execute the supported fixed-point operations on DSPs. This translates to a single PE execution spanning two different types of hardware resources. We refer to this implementation as the “Hybrid” implementation. For completeness, we compare a fourth implementation that restricts execution of the operations to the FPGA’s LUT resources.

All four implementations were synthesized and implemented using the Xilinx Vivado 2018.1 synthesis tool targeting the Zynq UltraScale+ MPSoC ZCU102. Correct functionality of OrthrusPE was confirmed by the Xilinx Vivado Simulator. Power estimates are obtained using the Xilinx Power Estimator and the Vivado Power Analysis tool, built into the Vivado Design Suite.

In order to fairly compare the four implementations, we fix the throughput to 1 MAC per cycle or 48 XNORs per cycle, i.e. a single OrthrusPE’s throughput. Higher performance of all implementations is possible by replicating the structures. The processing elements were synthesized across multiple target frequencies to show compatibility with any potential accelerator

Table 4.2: Resource Utilization results of the tested implementations.

Implementation	F=770MHz			F=160MHz		
	LUTs	**FF**	**DSP**	**LUTs**	**FF**	**DSP**
All-LUT	559	160	0	516	160	0
Hybrid (Common)	230	253	1	166	253	1
OrthrusPE	**165**	210	1	**111**	210	1
OrthrusPE-DS	**120**	229	2	**87**	229	2

which might utilize them. A BNN accelerator would typically require *hundreds* of PEs, therefore, the presented results scale gracefully based on the underlying accelerator.

4.1.5.2 Resource Utilization Analysis

Implementation of the individual PEs yielded the utilization results presented in table 4.2.

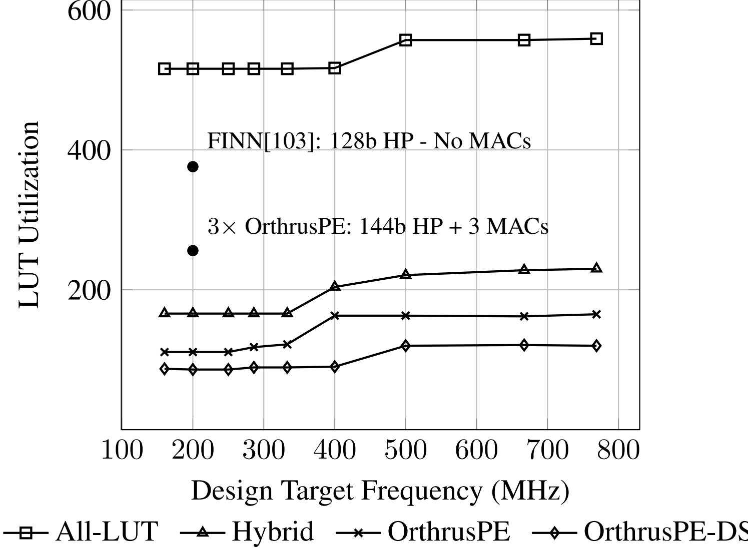


Figure 4.7: Synthesis results for LUT utilization across different design target frequencies. Each plot point represents a different synthesis run.

The results show that OrthrusPE can operate at the maximum frequency of the DSP48 block, i.e. the added functionality comes without any cost of latency. Practically BNN accelerators operate at lower frequencies, therefore OrthrusPE can be implemented on any BNN FPGA accelerator.

The LUT utilization differences within each implementation are minimal when synthesizing at frequencies above 500 MHz, as shown in figure 4.7. However, from 500 MHz down to 400 MHz, three of the implementations enjoy some relaxation in the parallelism required to meet the timing constraints. Another such relaxation occurs when the target frequency is lowered from 400 MHz

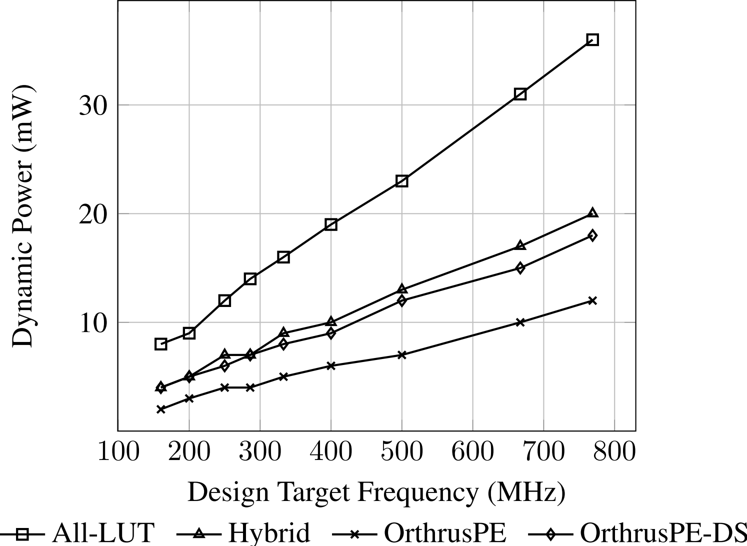


Figure 4.8: Dynamic power estimation at different design target frequencies. Each plot point represents a different synthesis run.

down to 333 MHz. Overall, our OrthrusPE and OrthrusPE-DS implementations, both executing SIMD binary Hadamard product on DSPs, result in the lowest LUT utilization cost.

A further plot point is added showing the resource utilization quoted in FINN [103] for popcount-accumulation of 128-bits at a target frequency of 200 MHz. The closest matching OrthrusPE implementation, in terms of bitwidth, provides 16 more bit accumulations and 3 parallel MAC operations (through runtime reconfigurability), while requiring 32% fewer LUTs.

4.1.5.3 Dynamic Power Analysis

Figure 4.8 shows the power estimates for the implementations at different design target frequencies. The results demonstrate that using a single OrthrusPE for MAC operations and binary Hadamard products presents the most efficient solution among the evaluated implementations. The OrthrusPE-DS solution also offers the second best power efficiency among the 4 configurations. In practice, OrthrusPE-DS can execute both types of operations concurrently which makes it well-suited for a pipelined accelerator.

The results show that DSPs present a good choice for accelerating binary operations in OrthrusPE and OrthrusPE-DS. Default implementations relying purely on LUTs or hybrids of LUTs and DSPs were less efficient in all of our experiments. Exploiting DSPs as in OrthrusPE improves the utilization of hard blocks already employed by accurate BNN accelerators. This does not prevent the design from employing further LUTs for further binary operations, yet it allows hard blocks to contribute to more types of computations.

4.1.6 Discussion

The development of OrthrusPE was aimed at achieving an efficient execution of accurate BNNs at the compute level. Analyzing the computational complexity of accurate BNNs allowed the hardware designer to use their conceptual understanding of the capabilities of the DSP block on FPGA to find a creative solution for the problem of supporting two types of numerical operations in the binary and fixed-point domains. To implement the solution, a handcrafted reprogramming of the DSP block was required to enable the desired functionality. This led to the HDL description of OrthrusPE, which wraps around the DSP block and allows the user to switch between the functions, and access the scratchpads accordingly. The runtime reconfigurable PE satisfies all the functions required by accurate BNNs, while capitalizing on resource reuse. Accurate BNNs cannot be achieved without fixed-point operations and reliance on DSP blocks. Instead of separating binary and fixed-point computations to two types of hardware resources, OrthrusPE improves the efficiency of the computation by executing both on FPGA hard blocks. Two configurations were evaluated, OrthrusPE and OrthrusPE-DS, across multiple target accelerator frequencies. Both solutions achieved improved resource utilization *and* power efficiency compared to typical BNN accelerator processing elements. Accurate BNNs solve many of the computation and memory challenges for deep neural network workloads on edge devices. Efficiently executing their mixed-precision computations can further exploit the advantages they offer at the hardware level.

4.2 Mind the Scaling Factors: Resilience Analysis of Quantized Adversarially Robust CNNs

The state-of-the-art status of DNNs in prediction, classification, and generative problems has led to their use in an increasing number of application domains, including those which are safety-critical. Using such algorithms in autonomous driving settings or space and defense applications emphasizes the importance of understanding their resilience against all forms of targeted and non-targeted errors. Most commonly, such errors can occur due to random hardware faults or adversarially-generated attacks. Understanding the outcome of an error in the execution of a DNN is a complicated process which depends on the underlying hardware, the DNN architecture and training, the input data, the location of the error in the DNN, the datatype it affects (i.e. weights or activations), the frequency of its occurrence, the numerical precision of the data, the location of the error within the numerical representation, among other factors. The inter-dependencies which lie among these factors can be extracted by performing large-scale experiments and ablation studies. However, explaining and interpreting the inter-dependencies requires the knowledge of a human-expert who analyses the data and connects the observations to the theoretical and conceptual understanding of the hardware and software implementation at hand. Equipped with this knowledge, the hardware and software designers can make conscious decisions which result in a more adversarially robust and hardware-error resilient deployment of the DNN.

In this handcrafted co-design example, adversarially trained DNNs are shown to be more susceptible to failure due to hardware errors when compared to vanilla-trained models. Large differences are identified in the quantization scaling factors of fault-resilient and fault-susceptible DNNs. Adversarially trained DNNs learn robustness against input attack perturbations, which widens their internal weight and activation numerical distributions. A larger numerical distribution results in a larger change in magnitude for a bit-flip in the respective numerical representation covering that distribution. Based on this understanding of quantization, arithmetic hardware, and DNN training, a simple weight decay solution is proposed for adversarially trained models to maintain adversarial robustness and hardware resilience in the same DNN. This improves the fault resilience of an adversarially trained ResNet56 by 25% for large-scale bit-flip benchmarks on activation data while gaining slightly improved accuracy and adversarial robustness.

4.2.1 Hardware Fault Resilience and Adversarial Robustness

Next to adversarial robustness and interpretability, resilience against hardware errors must be guaranteed before placing CNNs safety-critical settings. Understanding the failure cases for logic transient errors on datatype, frequency, bit-position, and number of affected computation units is important in carefully introducing hardware redundancy in a reasonable and cost-effective manner. Moreover, the method by which the CNN was trained affects its behavior in the presence of hardware errors [122, 123]. Consequently, it is also important to study the influence of compression [31] or adversarial training techniques [93] on fault resilience. Existing works in this domain have several limitations. Some only focus on robustness against input adversarial attacks without considering fault resilience [93, 124], others focus on random errors in different parts of the hardware with little attention to CNN training [125]. Works using aged CNNs without frequent, intermediate batch normalization layers have an exaggerated error-amplification

effect for bit-flips [126], while others using targeted bit-flip attacks (BFAs) construct network-specific attacks which are extremely unlikely to happen at random [127, 128, 123]. This work holistically investigates hardware fault resilience and adversarial robustness with large-scale resilience analysis on differently trained CNNs and identifies clear relationships between training-time CNN statistics and their deployment-time effect on scaling factors and clipping limits. The results of this work show that the common denominator for all resilient CNNs is small inter-layer data distributions, which result in smaller scaling factors at deployment. This allows small scaling factors to naturally introduce resilience by attenuating the largest possible perturbation.

The contributions of this work can be summarized as follows:

- Across $\sim$10M bit-flip experiments, regularly trained, adversarially trained, batch-norm free, weight decayed and pruned CNNs are considered. The hardware bit-flip module allows for testing a wide range of bit-flip patterns to analyze the effect of training/compression on hardware fault resilience.

- An in-depth analysis on the layer-wise data distributions of the considered CNNs is performed, by observing the differences in the scaling factors required for their quantization. Key insights are provided by studying the effect of batch normalization and weight decay, to harness scaling factors for improved fault resilience.

- Weaknesses in adversarially trained CNNs are identified, which open a backdoor for injecting faults of large magnitude. A simple weight decay remedy is proposed to shrink the quantization scaling factors, which improves resilience against faults in activation pixels by 25% on FastAT ResNet56, while preserving natural accuracy and adversarial robustness.

4.2.2 Related Work

4.2.2.1 Hardware Fault Resilience Analysis

He et al. [125] analyze the effect of logic transient errors using abstracted, high-level hardware models. The authors emphasized the importance of investigating faults on control and compute components compared to the limited analysis on memory-based bit-flips in existing works. No conclusions were drawn on the training scheme, compression, and adversarial robustness of the neural networks. Rakin et al. [127] introduced a progressive search technique to find optimal BFAs that break CNNs. In a CNN with 93M-bits of weights, the authors found 13 precise bit-flips which completely break the network. However, the probability of such an event happening at random is infinitesimal. Such non-random, specific cases can be referred to as *targeted* bit-flips. He et al. [123] performed resilience investigations on differently trained CNNs while employing such targeted BFAs. Several conclusions were drawn based on empirical results without further analysis to explain the underlying cause of the observations. Moreover, hardware designers cannot benefit from BFA analysis, as these are tightly optimized attacks for one considered CNN. To add hardware redundancy in an effective manner, large-scale resilience analysis can cover more general error cases and aid in making design decisions that benefit all CNN workloads. Lastly, BFA-based investigations only apply to bit-flips on a CNN's weights. In practice, logic

transient errors may happen in any part of the logic, including input pixels, partial sums, or output activations [125].

4.2.2.2 Fault Resilient Training and Adversarial Robustness

Hoang et al. [126] proposed to improve error resilience of CNNs by clipping activations. The investigations were limited to memory-based bit-flips on weights and only aged CNN architectures were tested, which have no batch normalization after each convolutional layer. Errors in such CNNs are typically exaggerated compared to modern CNNs, as batch normalization naturally reduces the activation distribution and scaling factors (as shown in figure 4.10). In an adversarial attack scenario, input noise is propagated and amplified through the layers causing a misclassification. Liao et al. [129] proposed to mitigate the amplification error by using a denoiser to reduce input perturbations. Lin et al. [124] applied Lipschitz regularization to limit the error amplification in quantized CNNs. Both works focused on mitigating attacks injected at the input, but did not consider inter and intra-layer faults (depicted in figure 4.11). Zahid et al. [122] introduced a fault-injection layer at training time. The work focused on a class of permanent errors and did not consider adversarial attacks. A defense method against targeted DRAM bit-flip attacks was proposed by Li et al. [128], where weights were preprocessed to limit their change of value. The method was limited to weight-based, memory-only, targeted BFA and did not consider input-based adversarial attacks.

4.2.3 Methodology

4.2.3.1 Problem Formulation: Quantization and Bit-Flips

Recalling equation 2.3, a single weight value w multiplied by an activation pixel a produces a partial result in the convolution operation of the weight tensor $\mathbf{W}^l$ and input activation tensor $\mathbf{A}^{l-1}$ in layer l of an L-layer CNN. At training time, $\mathbf{A}^{l-1}$ and $\mathbf{W}^l\ \forall l \in L$ are represented by high-precision FP32 values to maintain smooth training and fine adjustments through backpropagation. During inference, the values are quantized to reduce their memory footprint and arithmetic computation complexity on embedded HW. The 8-bit signed integer (INT8) representation is one of the most common numerical representation formats for lean deployment on resource constrained devices. Equation (2.6) showed the basic principle of linear quantization of x_f (either w or a) into a more constrained numerical representation $\mathtt{x_q}$.

The scaling factor v projects the quantized range of INT8 $[-128, 127]$ onto the real range of values which $x_f \in \mathbf{X}_f$ can take with respect to the $clip$ operator. Note that $\mathbf{X}_f$ is either $\mathbf{W}^l$ or $\mathbf{A}^{l-1}$. The $round$ operation pushes the smooth values of $\mathbf{X}_f$ into the limited 256 integer values of INT8. As mentioned previously, the $clip$ operator cuts-off values of the $\mathbf{X}_f$ range beyond $[-c, c]$, maintaining symmetric linear quantization, even in cases where layers such as ReLU leave only positive activations, and weights use only a small portion of the negative number scale. By observing the statistics of weight and activation distributions of a layer, the *calibration* process sets c and v, such that the range of values that appear in a certain layer can be covered by the INT8 static range [58]. Therefore, c and v are directly influenced by the dataset, the weight values of the CNN (e.g., learned through vanilla or adversarial training, regularized or not) and

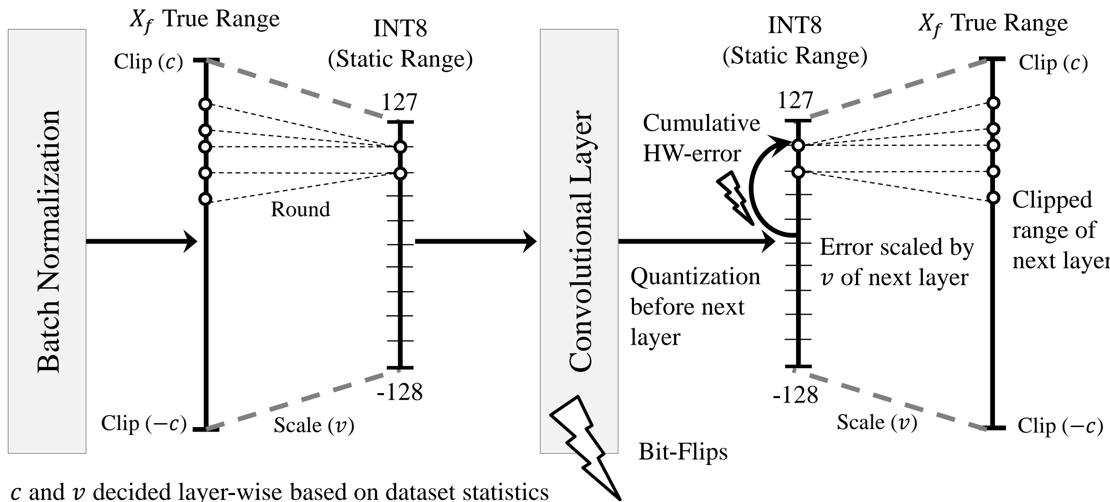


Figure 4.9: Batch-norm limits activation range at *training time*, effectively lowering v and c of the subsequent convolutional layer at *deployment time* (on hardware). Errors in the convolutional layer can at most grow in magnitude to the defined clip c of the next layer.

its structure (e.g., existence of batch-norm layers). The described quantization of $\mathbf{X}_f$ to INT8 is visualized in figure 4.9.

A runtime reconfigurable bit-flip module is implemented to change the value of any position in the 8-bit representation, for weights and activations, and for any subset of multipliers in a standard spatial DNN accelerator [96]. Flipping the n-th bit of an operand at the input of any affected multiplier translates to a 2^n absolute change in magnitude within the static INT8 range $[-128, 127]$. However, it is more important to analyze the precise severity of a 2^n flip with respect to the values of the projected *real range* of $\mathbf{X}_f$, i.e. after applying scaling factors v.

With this conceptual understanding of quantization and bit-flips, some general insights can be made:

- **Quantization naturally improves bit-flip resilience.** Quantization clips the largest possible perturbation when projecting a larger, dynamic representation, such as FP32 into a more constrained range of INT8. As the clip limits c and scaling factors v are decided based on statistics *before* deployment on hardware, a single or multiple bit-flips on hardware cannot perturb the network beyond c of the next layer (figure 4.9). This is an inherent improvement in bit-flip resilience over float/dynamic numerical representations.

- **Batch normalization improves quantized CNN bit-flip resilience.** Resilience analysis on aged CNNs (LeNet, VGG and AlexNet), which do not employ batch normalization after every convolutional layer, cannot be extended to modern CNNs. The lack of batch normalization layers in aged CNNs aggravates the effect of bit-flips, as their activation and weight distributions are much larger than modern CNNs. Consequently, INT8 variants of

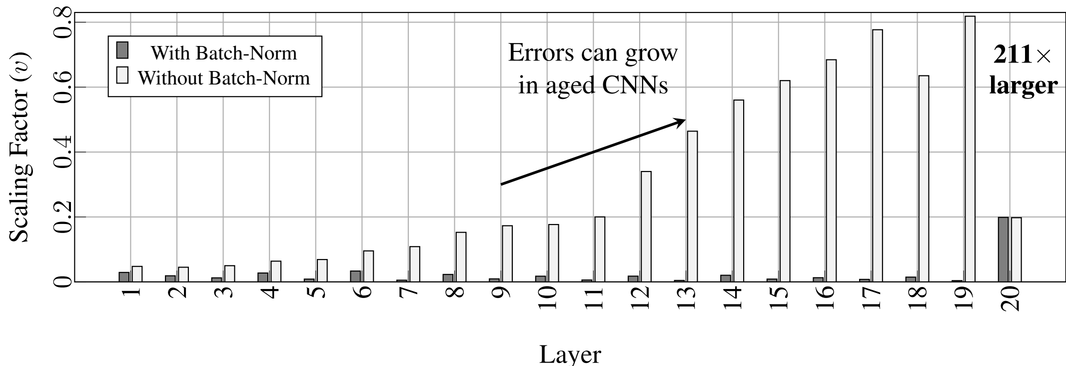


Figure 4.10: Layer-wise scaling factors v of ResNet20 CNNs trained on CIFAR-10, with and without batch-norm. Works investigating bit-flips on aged CNNs (without batch-norm after every layer) cannot be extended to modern CNNs.

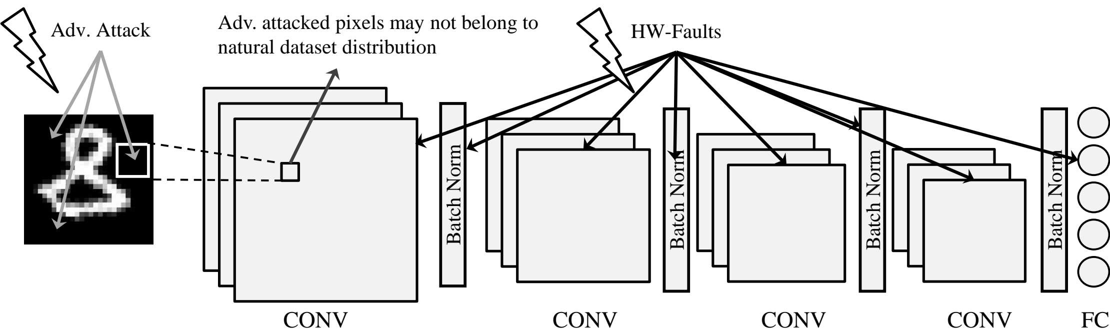


Figure 4.11: Adversarial attacks apply input perturbations to cause incorrect classifications. Training for such attacks implies training for pixel value distributions outside of the natural dataset. Differently, hardware faults can occur at any point within the CNN, and are not limited to the input of the network.

aged CNNs will have large scaling factors v to accommodate the activations that appear in the convolutional layers, resulting in a much larger true magnitude error for any bit-flip. The scaling factors v of ResNet20 with and without batch-norm are shown in figure 4.10 to visualize this problem. Errors in aged CNNs can also propagate and get amplified, as the scaling factors grow in deeper layers.

- **Adversarial training can affect a quantized CNN's bit-flip resilience.** As elaborated in section 2.2.4, adversarially trained CNNs need to be robust against input perturbations which may not follow the statistical distribution of the original training dataset. This affects the statistical distribution of the learned weights in adversarially robust CNNs compared to vanilla trained ones, thereby influencing the scaling factors v during hardware deployment, even when calibrating on non-adversarial, natural data.

4.2.3.2 Error Model and Benchmark Phases

Bit-flips at the compute level fall under logic transient errors [125], and capture a broader range of error patterns compared to memory-based faults. A memory-based fault on a weight parameter w implies all computations using w are affected. With logic transient errors, memory-based faults can be replicated, *as well as* every other case where a subset of w's computations are affected. This provides finer granularity in error injection control for the large-scale bit-flip benchmarks planned in this work. The large-scale bit-flip benchmarks are made possible by exploiting the flexibility of a run-time reconfigurable bit-flip injection hardware module implemented on the accelerator as part of this work. Large-scale statistical fault injection is an established approach to analyzing errors in logic [125]. However, it is often infeasible due to slow RTL simulations. RTL simulations are circumvented in this work by directly implementing the bit-flip module on NVDLA [96] and injecting the desired bit-flip patterns on the running hardware. The benchmarks are developed with well-defined bit-flip patterns, to better understand the effect of bit-flip characteristics such as position in numerical representation, frequency of occurrence, affected datatype, and affected percentage of multipliers.

The benchmark is defined in steps, where each successive step changes one aspect of the bit-flip pattern. The bit-flip pattern is maintained and the accelerator performs inference of an entire test set of input images. Once the test set is exhausted, the next step begins with a new bit-flip pattern and the test set is passed once more. The benchmark steps are shown as a nested-loop in algorithm 4.1.

First, the frequency f of bit-flip occurrence is set. The frequency indicates the rate of bit-flip injection per computation, i.e., if f is set to 0.1, a bit-flip is introduced at every 10-th computation of the affected hardware component. Next, the affected datatype t is set, as in activations $\mathbf{A}$ or weights $\mathbf{W}$. Third, the loop goes over the bit-flip position b, indicating the severity in magnitude change for the value of the input operand of the affected computation. Finally, the inner-most loop chooses the number of affected multipliers m, as a percentage of the accelerator's total MAC units. Figure 4.12 visualizes these bit-flip characteristic parameters. At the core of the nested-loop in algorithm 4.1, the characteristics are programmed into the bit-flip module, then the accelerator is allowed to perform inference over the *entire* test set. Here, system failures are defined as those cases when the prediction with hardware errors disagrees with that of the *same* CNN without any bit-flips. Therefore, failures are not counted based on the accuracy of the model or the true label of the input image. This definition aligns with existing work [125], and is fair when comparing different networks, as their underlying baseline accuracy is orthogonal to their resilience against hardware errors.

4.2.4 Evaluation

The experiments are performed on the CIFAR-10 dataset, using 50K images for training and 10K test images for evaluation. The test set also serves as the hardware fault test set in algorithm 4.1. ResNet20 and ResNet56 represent shallow and deep baseline models for the CIFAR-10 dataset. If not otherwise mentioned, all hyper-parameters specifying the task-related training were adopted from ResNet's base implementation [28]. Pruned variants are obtained by re-implementing the reinforcement-learning-based pruning agent proposed in AMC [31]. The fault resilience of pruned

Algorithm 4.1 Large-scale Resilience Analysis Benchmark

```
P = EvaluateTestSet()                              ▷ Get Predictions w/o Bit-Flips
for f in F do                                      ▷ Frequency of Bit-Flip Occurrence
    for t in T do                                  ▷ Datatype (Weight, Activation)
        for b in B do                              ▷ Bit Position in INT8 Value
            for m in M do                          ▷ Affected MAC Units
                ProgramBitFlipModule(f,t,b,m)
                P̂ = EvaluateTestSet()
                FailRate = Count(P ≠ P̂)/TestSetSize
```

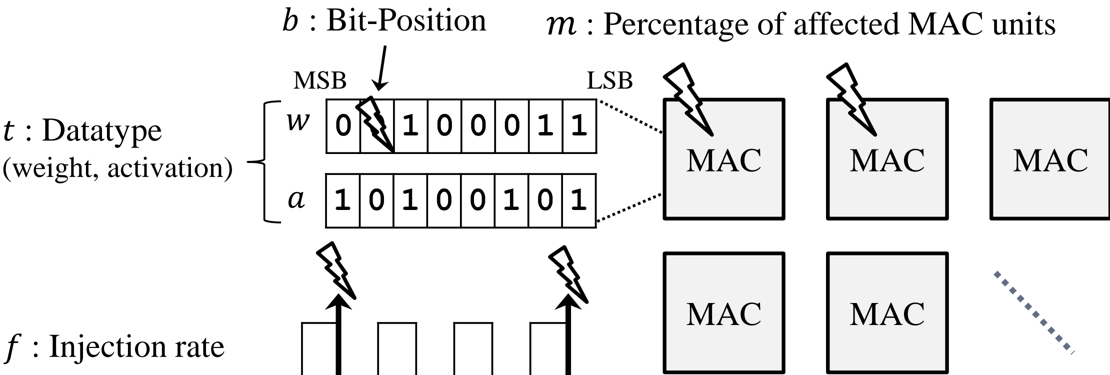


Figure 4.12: Parameters to determine bit-flip characteristics of the benchmark.

CNNs is investigated with 50%-60% fewer operations remaining compared to their unpruned variants. For defensive training against adversarial attacks, we use the popular FastAT [93] approach and the training hyper-parameters described in the paper. To evaluate adversarial robustness, we apply a strong unseen PGD [92] adversarial attack on all considered CNNs with 20 iterations and a perturbation budget ϵ_{bug}=2. The entropy-based calibrator of TensorRT is used to find the optimal v and c for each layer of the full-precision CNNs, before INT8 execution. This persistently gave better accuracy over the naive min-max calibrator. As an added benefit for the bit-flip experiments, the entropy-based calibrator provides smaller clip ranges than the naive min-max method, which benefits the fault resilience of all considered CNNs. All CNNs are calibrated on the same dataset, i.e., the same images are passed to compute v, c of each layer of each CNN, before deployment on hardware.

We synthesize a 64 MAC unit variant of the NVDLA accelerator on the Xilinx ZCU102 board. The bit-flip module is written in Verilog and wraps around the MAC units without adding any delays to any critical paths of the accelerator design. The sets in algorithm 4.1 are $\mathcal{F} = \{0.1, 0.02, 0.01, 0.005, 0.002\}$, $\mathcal{T} = \{A, W\}$, $\mathcal{B} = \{5, 6, 7\}$, and $\mathcal{M} = \{25\%, 50\%, 100\%\}$, where the benchmark loops over the elements in the order they are presented here. Bit-position b = $7 \in \mathcal{B}$ indicates a flip in the sign-bit of INT8. The sets $\mathcal{F}$, $\mathcal{T}$, $\mathcal{B}$, and $\mathcal{M}$ were chosen after an ablation study on the considered networks. The ranges for each bit-flip characteristic adequately represent weak-to-strong influence on CNN fault rate for the purpose of our analysis.

4.2.4.1 Large Scale Resilience Analysis

The results of the benchmark detailed in algorithm 4.1 are shown in figure 4.13. The following observations can be made:

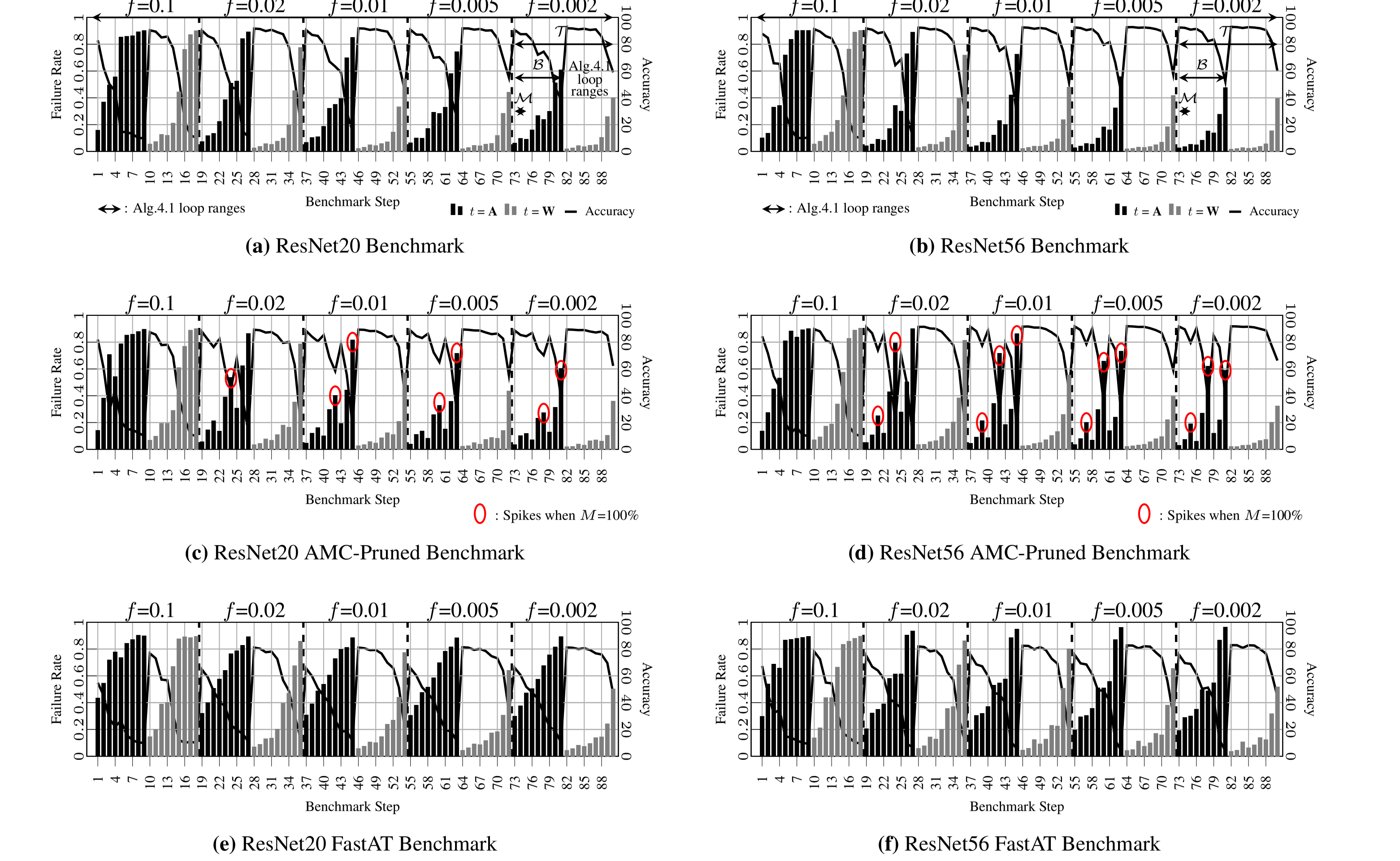
- **Activation sensitivity.** Flipping bits of input activations **A** is more likely to cause failures compared to flipping weight bits in **W** at any bit-flip position, on any number of multipliers and any frequency of bit-flip injection. Many memory-based and targeted bit-flip works only flip the weights of the CNN, without investigating input activations [126, 128], which are persistently more vulnerable in all our tested CNNs, and all bit-flip patterns of the benchmark.

- **Sign-bit sensitivity.** An expected and common observation is the high impact of the sign-bit in deciding the probability of failure. However, it is interesting to note the *degree* of its importance; in almost all cases, flipping the sign-bit in 25% of the multipliers is more potent than flipping the 6-th bit on 100% of the multipliers, at any given frequency, for both weights **W** and activations **A**. Flips on the 5-th bit (or lower, based on observations not shown for brevity) are almost negligible at low injection rates, even on 100% of the MAC units.

- **Adversarially robust CNNs are vulnerable to hardware errors.** There is a clear degradation in fault resilience for adversarially robust CNNs, particularly for activation-based bit-flips. We address this observation more closely in the next section. Pruned CNNs exhibit resilience properties close to their unpruned counterparts. This is justified as their scaling factors v are similar to the original (vanilla) unpruned network. However, spikes of high failure rates (marked in figure 4.13) occur when $m = 100\%$, indicating that injecting many perturbations in a CNN with fewer computations (due to pruning), leads to slightly weaker fault resilience.

- **Deep CNNs with batch normalization are resilient.** Deeper CNNs (56-layers) have improved fault resilience over their shallow (20-layers) counterparts for vanilla, pruned, and adversarially robust variants. The errors introduced in the early layers of the network do not *grow* with the depth of CNN. This can be credited in part to the batch normalization layers which take place after every convolutional layer, regulating the maximum possible perturbation that can pass to the next layer, (1) due to calibration-time statistics (helps in lowering scaling factors v of the layers) and (2) run-time normalization. He et al. [123] show benefits of batch normalization against *targeted* (search-based) bit-flip attacks. We further show the benefits of batch normalization more generally against any hardware-based faults (non-targeted). The two ResNet20 variants presented in figure 4.10 (Vanilla and No Batch-Norm) are evaluated in table 4.3. The overall mean failure rate is *doubled* in the variant without normalization, due to its high scaling factors which amplify errors in the CNN.

The results in figure 4.13 can shed light on parsimonious hardware-error resilience options. For example, the designer may apply a redundancy method on the computations against the sign-bit or allocate resilient memory holding activation bits (e.g. 8T-SRAM). More conservatively,



(a) ResNet20 Benchmark

(b) ResNet56 Benchmark

(c) ResNet20 AMC-Pruned Benchmark

(d) ResNet56 AMC-Pruned Benchmark

(e) ResNet20 FastAT Benchmark

(f) ResNet56 FastAT Benchmark

Figure 4.13: Bit-flip experiments following algorithm 4.1 on vanilla, pruned and adversarially trained ResNet20 and ResNet56. Each bar represents the failure rate of a particular bit-flip setting $\{f, t, b, m\}$ tested over 10K test images. Each sub-figure comprises 900K bit-flip experiments.

the designer may apply that redundancy to only a subset of multipliers, e.g. 50% of the MAC array, further saving resources and area-on-chip. Such design decisions can be made based on large-scale resilience experiments, and would not be possible based on targeted bit-flip attacks [127, 123, 128].

4.2.4.2 In-depth Analysis of Adversarially Trained CNNs

The conceptual understanding of the problem definition in section 4.2.3.1 helps in interpreting the observation of reduced fault resilience of adversarially trained CNNs. Quantization to a constrained numerical representation (INT8 or similar), implies that few discrete values must represent a wider range of dynamic real-values. The true range covered by the INT8 representation depends on the scaling factor v and clipping limit c. Since adversarially trained and vanilla CNNs are structurally identical, the first point of investigation is the data distributions of these CNNs. In figure 4.14, the scaling factors v for each convolutional layer of the CNN are shown, obtained through the entropy-based calibrator on the *same* calibration dataset. A clear difference can be observed, where adversarially trained (FastAT) layers can have up to $\sim 7\times$ higher scaling factors compared to the respective vanilla-trained CNN.

Regularization loss $\mathcal{L}_{reg}$ is an auxiliary loss typically added to the cross-entropy loss $\mathcal{L}_{ce}$ to penalize weights with high magnitude during neural network training. As shown in equation 4.7, this loss is scaled with the weight decay (α_d) hyper-parameter, to strengthen/weaken its effect on the overall loss formulation $\mathcal{L}_{\text{total}}$ during backpropagation and weight update.

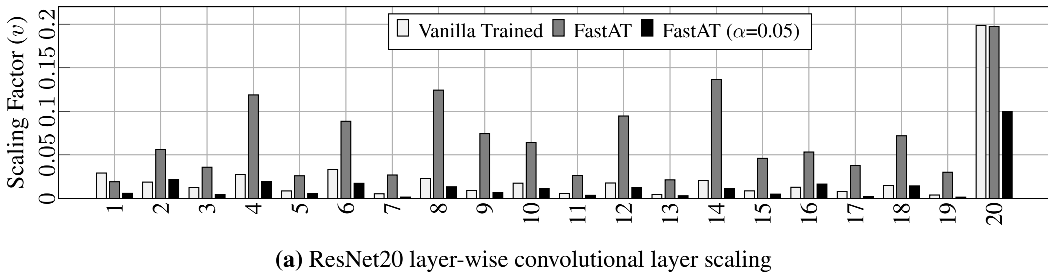
$$\mathcal{L}_{total} = \mathcal{L}_{ce} + \alpha_d \mathcal{L}_{reg} \tag{4.7}$$

Weight decay α_d was set equivalently for both vanilla and FastAT training (α_d = 0.0005 and α_d = 0.0004, respectively, based on original papers [28, 93]). For the same $\mathcal{L}_{reg}$ and α settings, the FastAT CNN incorporated large inter-layer data distributions to achieve its high robustness against adversarial attacks. Following the third insight made in section 4.2.3.1, adversarial training introduces pixel values which do not fall under the distribution of the standard training dataset. This forces the CNN to learn them to achieve higher adversarial robustness, *stretching* its trainable parameter distributions (weights and batch normalization parameters). When performing calibration on natural, unattacked data, the scaling factors v *grow* accordingly (figure 4.14). This opens a *backdoor* to inter- and intra-layer hardware perturbations (bit-flips) during execution, which end up having *high true magnitude* as a consequence of the larger scaling factors (v) in adversarially robust CNNs.

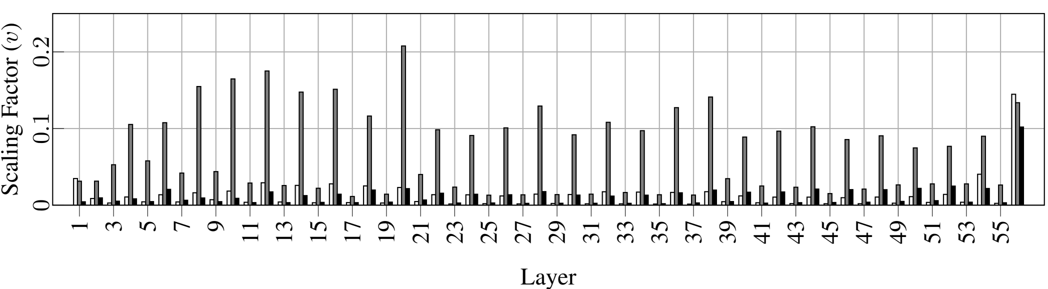
Strengthening the effect of $\mathcal{L}_{reg}$ during training pushes the weights to a more constrained distribution, and correspondingly, the activations resulted by those weights. As an initial remedy, we propose increasing the weight decay during training, which naturally shrinks the weight distributions. In figure 4.14, we show the scaling factors of FastAT-trained ResNet20 and ResNet56 CNNs with high weight decay α_d = 0.05, bringing them *back* to vanilla training levels.

4.2.4.3 Results and Conclusions

In table 4.3, the results of figure 4.13 are summarized, as well as the weight decayed variants of FastAT (α_d = 0.05). As a coarse indicator of hardware fault resilience, the mean failure rate (MFR)



(a) ResNet20 layer-wise convolutional layer scaling



(b) ResNet56 layer-wise convolutional layer scaling

Figure 4.14: Convolutional layer scaling factors for vanilla trained and adversarially robust variants of ResNet20 and ResNet56. High weight decay (α_d = 0.05) brings the high scaling factors v of FastAT back to vanilla levels.

is provided for each CNN over the entire benchmark in algorithm 4.1 (Overall). Additionally, to help in understanding the effect of individual characteristics of the bit-flip patterns, one bit-flip characteristic is fixed (f, b, m, or t) and the MFR over all steps varying the other bit-flip parameters is measured.

The observations made in section 4.2.4.1 are supported by the MFR presented in table 4.3. For FastAT CNNs, a 62% and 95% degradation in overall MFR can be observed for ResNet20 and ResNet56, respectively, compared to their vanilla-trained variants. When increasing α_d, the FastAT CNNs improve by up to 16% in overall MFR. More specifically, the fault resilience against activation bit-flips t = **A** is improved by 27% and 25% for the high weight decay FastAT ResNet20 and ResNet56, compared to the regular FastAT implementation. Although baseline accuracy is considered orthogonal to fault resilience analysis (explained in section 4.2.3.2), it is interesting to discuss the trade-offs that can be achieved in fault resilience, adversarial robustness, and natural accuracy. In general, adversarial training techniques in literature incur a degradation in natural accuracy when trying to learn adversarial attacks as well as their target classification task [93]. An observation can be made for the smaller FastAT ResNet20 suffering a further drop of 3.8 p.p. in accuracy after applying high α_d. However, the larger ResNet56 has a slightly improved accuracy after weight decay compared to the regular FastAT implementation. Weight decay can be harsh, particularly on smaller CNNs, as more weights approach zero and lose their feature representation capability. ResNet56 has sufficient redundancy to compensate for this (and even benefits through regularization); however, the smaller ResNet20 loses some of its natural accuracy. Although weight decay is proposed as an initial, simple remedy for the

Table 4.3: Summary of results on shallow (ResNet20) and deep (ResNet56) CNNs as vanilla, pruned, and adversarially trained variants. Percentage improvement shown for FastAT α_d = 0.05 over regular FastAT.

Model	Train/Config	Baseline (INT8) Acc. [%]	PGD-20 Atk. Acc.[%]	Mean Failure Rate (MFR) - Lower is better: Overall	$f = 0.005$	$f = 0.1$	$b = 5$	$b = 7$	$m = 25\%$	$m = 100\%$	$t = \mathbf{W}$	$t = \mathbf{A}$
ResNet20 CIFAR-10	Vanilla	92.03	1.04	0.29	0.21	0.53	0.09	0.55	0.19	0.37	0.19	0.39
	No BatchNorm	79.12	5.01	0.60	0.57	0.69	0.48	0.72	0.53	0.70	0.40	0.81
	60% Pruned	89.59	1.21	0.27	0.17	0.56	0.11	0.47	0.15	0.41	0.19	0.35
	FastAT [93]	81.58*	72.85	0.47	0.40	0.67	0.27	0.70	0.39	0.57	0.30	0.64
	FastAT α_d=0.05	77.72	70.36	0.40 (15%)	0.31 (23%)	0.65 (3%)	0.20 (26%)	0.62 (11%)	0.24 (38%)	0.55 (4%)	0.33 (-10%)	0.47 (27%)
ResNet56 CIFAR-10	Vanilla	92.94	4.53	0.22	0.13	0.49	0.06	0.46	0.13	0.32	0.17	0.28
	50% Pruned	92.04	2.66	0.28	0.19	0.54	0.10	0.47	0.15	0.44	0.19	0.37
	FastAT [93]	82.71*	72.72	0.43	0.35	0.66	0.21	0.69	0.31	0.54	0.30	0.56
	FastAT α_d=0.05	83.37	74.72	0.36 (16%)	0.25 (29%)	0.65 (2%)	0.17 (19%)	0.63 (9%)	0.25 (19%)	0.48 (11%)	0.31 (-3%)	0.42 (25%)

*: Accuracy degradation from vanilla-training is common in state-of-the-art adversarial training to achieve high adv. robustness (see accuracy after PGD attack)

adversarial training and fault resilience problem, the analysis provided in this work identifies a larger challenge in bringing robustness of both domains (adversarial attacks and hardware faults) in the same CNN. It is also important to note that adversarially trained CNNs, even with the proposed high α_d, are still less fault resilient than vanilla CNNs. The reason being that weight decay indeed shrunk the convolutional layers' scaling factors, but the batch normalization trainable parameters (γ_{bn}, β_{bn}) are not directly affected by weight decay, leaving their scaling factors large due to adversarial training.

4.2.5 Discussion

This work highlighted the importance of scaling factors for maintaining hardware-fault resilience of efficient, quantized CNNs. The importance of scaling factors was verified by performing large-scale bit-flip experiments on regularly trained, adversarially trained, batch-norm free, weight decayed, pruned, deep and shallow CNNs. Extracting key insights from the results generated by the large-scale experiments required human expert-knowledge in ML and hardware concepts, such as adversarial training, quantization, calibration, and neural network data distributions. Only by conceptually understanding both sides, the execution on hardware and the network's training properties, the results were made interpretable and used to develop an intermediate solution based on this understanding. If an automated agent were provided access to all training parameters (learning rate policy, scheduler and value, momentum, batch-size, epochs, loss formulation, optimizer type, regularization type, weight decay, etc.) as well as the large-scale resilience benchmark, it would take a prohibitively long time for it to find out how each training hyper-parameter affects bit-flip resilience. Keeping in mind that each time the automated agent reconfigures the training setup to test out a different training hyper-parameter configuration, an *entire*, costly, GPU-based training run is required before deployment, followed by large-scale bit-flip experiments, to collect the reward/result for that particular training configuration. On the other hand, the human expert required only two experiments (ResNet20 with and without batch normalization) to prove their *hypothesis* which connects scaling factors to bit-flip resilience. After confirming the hypothesis, the same idea extended itself to adversarially-trained CNNs, proving that their large scaling factors open a backdoor for bit-flips with large true magnitude perturbations. The human designer then used their theoretical understanding of how each training hyper-parameter affects data distributions in a CNN, which indirectly affects the scaling factors

at deployment time on a quantized accelerator. The relevant hyper-parameter, weight decay, was tweaked to improve the resilience of an adversarially trained ResNet56 by 25% on activation faults. This succinctly captures the process of *handcrafted* HW-CNN co-design.

5 Semi-Automated Co-Design

CO-DESIGN problems can involve a multitude of components, where handcrafting all parts of the solution is challenging, and formulating the whole problem into a feasible, traversable, and well-defined search space is not possible. In such cases, certain computation models may be used to *aid* the human designer in optimizing some components. Additionally, low-level, handcrafted components may be integrated in an automated manner into a larger system, which is then optimized by an agent or a model of computation (MoC). In this chapter, two examples are introduced, where neural network accelerators reuse the handcrafted components from chapter 4 in a larger hardware design, which can be represented as a data flow graph (DFG). When implementing the neural network as a computation graph on a dataflow hardware architecture (recall figure 2.8), the architecture of the neural network defines the complexity of the graph and the computation effort in each node (layer). To a large extent, the neural network is *itself* the hardware design. This context forms the HW-CNN co-design problem for this chapter. Based on the layer-wise computation effort, the designer must accordingly specify the resources to be allocated in different parts of the graph. Here, the allocation not only has to respect the resources available on the target embedded FPGA platform, but also consider the throughput and efficiency of the computation pipeline resulting from the synthesized graph. Dataflow architectures can be optimized in a semi-automated manner when compiling the graph in HLS; the human designer must specify the resources for nodes in the graph, but the allocation of FIFO communication buffers and the computation pipeline is automatically generated. This form of co-design was used to fit highly efficient BNNs on a semi-autonomous prosthetic hand in Binary-LoRAX [25], and enabled accurate, privacy-preserving, edge-based face-mask wear and positioning detectors during the COVID-19 pandemic in BinaryCoP [24].

5.1 Binary-LoRAX: Low-power and Runtime Adaptable XNOR Classifier for Semi-Autonomous Grasping with Prosthetic Hands

Intelligent, semi-autonomous prostheses take advantage of combining autonomous functions and traditional myoelectric control. With the help of visual and environment sensors, intelligent prostheses achieve a level of autonomy which relieves the user from generating elaborate electromyographic (EMG) signals for grasp type and trajectory. To achieve the desired functionality, the semi-autonomous prosthesis must efficiently process the incoming environmental data at a high rate, with low power and high accuracy. This work proposed Binary-LoRAX, a low-latency runtime adaptable classifier for the semi-autonomous grasping task of prosthetic hands. The

classification task is offloaded to an efficient BNN accelerator which performs high-throughput XNOR operations on DSP blocks. To tailor the classifier's performance to the current application scenario, a frequency scaling approach is proposed, which dynamically switches between two modes of operation, high-performance and power-saving. At high-performance, classifications are performed with a low latency of 0.45 ms, high-throughput of 4999 frames per second (FPS) and power consumption of $\sim$2.15 W. This enables functions such as object localization and batch classification. Switching to power-saving mode, a latency of 80 ms is maintained, with up to 19% improved classifier battery-life. Our prototypes achieve a high accuracy of up to 99.82% on a 25-class problem from the YCB graspable object dataset [130].

5.1.1 HW-DNN Co-design for Intelligent Prosthetics

The design of low-power, performant, intelligent systems emphasizes the importance of an efficient deployment of deep learning algorithms on embedded hardware. Specifically for autonomous applications, including robotic or prosthetic devices, real-time interpretation of sensor data is essential for responsiveness. When visual sensors such as cameras are used, the processing of the high-bandwidth input data is challenging, especially for battery-powered systems. In prosthetic hands, the implementation of semi-autonomous functions is enabled through in-hand visual perception, which requires efficient embedded processing to avoid insecure, high-latency external compute services. The complete control algorithms, including image recognition, must be executed on in-hand embedded processing hardware.

Such contradictory objectives of maximizing performance while minimizing power and resource utilization, assign a decisive role to HW-DNN co-design. Binary-LoRAX is an efficient, runtime adaptable BNN classifier for the semi-autonomous grasping task of prosthetic hands. The challenges arising from the prosthetic hand's application constraints are tackled through the following contributions:

- Training BNNs for the graspable object classification task, enabling the efficient deployment of neural networks on intelligent prostheses with a task-related accuracy of 99.82% on a 25-class problem from the YCB object dataset [130], adding 12 classes compared to existing work [131].
- Achieving low-latency classifications of 0.45 ms, consuming $<$1% of the optimal controller delay [132] and achieving a 99.7% reduction in latency compared to existing work [131].
- Efficiently executing XNOR operations on an FPGA's DSP blocks in a vectorized manner, freeing more LUT resources and allowing larger BNNs to fit onto embedded FPGAs.
- Dynamically adapting the frequency of the accelerator, offering high-performance and low-power modes to target different application scenarios (dangerous/delicate objects, batch processing, object localization, low battery, prosthetic movement), improving the classifier's battery-life by up to 19% compared to [103].

5.1.2 Related Work

5.1.2.1 Efficient Intelligent Prosthetics

Computer vision-based, semi-autonomous control of prosthetic hands has been proposed in several recent works [133, 134, 135, 136, 137, 138, 139, 140]. The implementation of semi-autonomous hand functions reduces the complexity of required user-control commands typically generated by EMG signals. The constrained set of commands can be invoked with lower cognitive effort by the user [136, 141]. The simplified commands also require less complex electrode setups, as high accuracy and long-term stable EMG-pattern recognition is not required.

The KIT Prosthetic Hand proposed in [137] employs an in-hand camera to achieve its semi-autonomous functions. Parts of the grasp action are automated with the help of visual, environmental information. To execute a semi-autonomous grasp, visual information about the object is obtained through the camera, such as the object class and/or dimensions. This information is then used to select a suitable grasp from a database, where parameters can include finger trajectories and grasp force. In [131], a two-step classification system for the KIT Prosthetic Hand was proposed, where an object classification algorithm and an acknowledgment from the user triggers a second semantic segmentation neural network.

In the first version of the KIT Prosthetic Hand, an ARM Cortex M7-based microprocessor was used. The newer design shown in figure 5.1 houses a custom, Zynq Z7010-based printed circuit board (PCB).

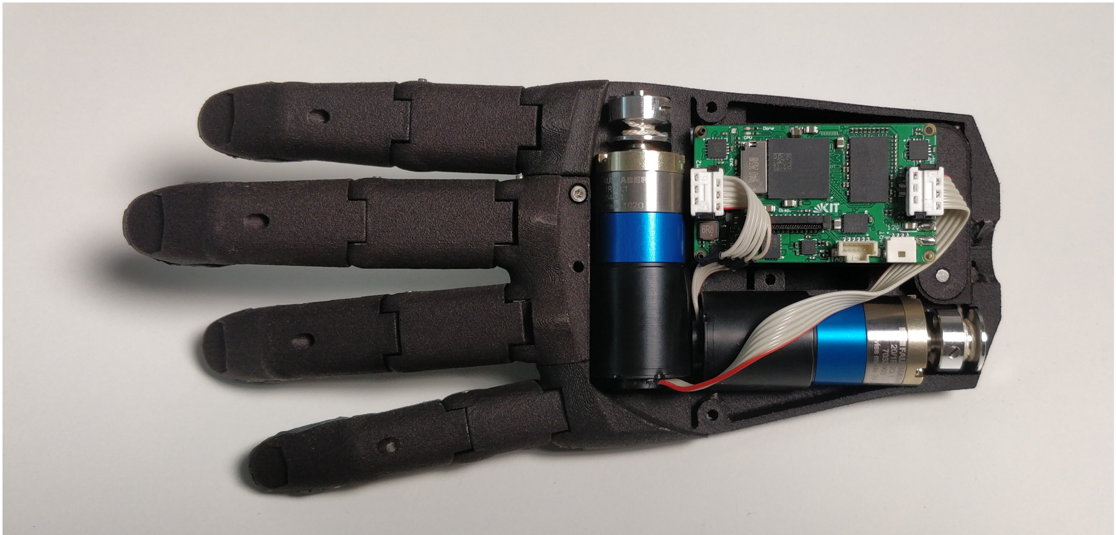


Figure 5.1: KIT Prosthetic Hand (50$^{\text{th}}$ percentile female) with Zynq Z7010-based processing system

5.1.2.2 Binary Neural Networks for Intelligent Prosthetics

Works such as [34, 62, 142, 143] have focused on adding algorithmic or structural complexity to BNNs to achieve classification performance close to full-precision CNNs on complex tasks [144].

However, simpler tasks with lower scene complexity can be handled with more efficient BNNs [60, 24].

In the context of semi-autonomous prosthetic hands, the camera input at the instance before the grasp operation takes place is expected to have one central object in the field-of-view. In that regard, the task's complexity resembles that of popular datasets, such as the German Traffic Sign Recognition Benchmark (GTSRB) [145], Street View House Numbers (SVHN) [146] or CIFAR-10 [27], all of which have the object of interest in the forefront of the scene, with minimal random background complexity when compared to autonomous driving scenes such as Cityscapes [147]. It is important to note that BNNs have shown high accuracy and good generalization on the mentioned datasets [60, 103]. Considering the power, memory, accuracy, and latency requirements of the target application, along with the limited battery-life and compute capabilities of the small, edge compute device on the prosthesis, BNNs represent good candidates for the graspable object classification problem.

5.1.2.3 The XILINX FINN Framework

FINN [103] is a popular framework for accelerating BNNs on FPGAs. Although the framework is designed for BNNs presented in [60], it also supports 2-bit weights and/or activations. FINN compiles HLS code from a BNN description to create a hardware design for the network. The generated streaming, dataflow architecture consists of a pipeline of individual hardware components instantiated for each layer of the BNN. OrthrusPE [8], as presented in section 4.1, investigates the effectiveness of deploying binary operations onto DSPs as SIMD binary Hadamard product processing units. For Binary-LoRAX prototypes the FINN architecture is infused with handcrafted, reprogrammed DSPs, which are switched *statically* to the binary operation mode presented in section 4.1. For LUT constrained devices such as the Z7010, this allows larger and/or faster accelerator designs, by spreading out computations to DSPs. This is further extended with runtime frequency scaling to achieve different modes of operation, for different latency requirements and power consumption rates, based on the current application scenario of the prosthetic hand.

5.1.3 Methodology

5.1.3.1 Training and Inference of Simple BNNs

For efficient approximation of weights and activations to single-bit precision, the BNN method by Courbariaux et al.[60] is used. A brief recap of these simple BNNs is provided in this section. At training time, the network parameters are represented by full-precision latent weights **W** allowing for a smoother convergence of the model [59]. It is important to note that the input and output layers in this implementation are not binarized, to avoid a drop in classification accuracy.

During the forward-pass for loss calculation or deployment, the weights $w \in \mathbf{W}$ are transformed into the binary domain $b \subset \mathbf{B} \in \mathbb{B}^{K_x \times K_y \times C_i \times C_o}$, where $\mathbb{B} = \{-1, 1\}$. Note that this simple form of binarization does not involve multiple binary bases (M, N) as those discussed in section 4.1.3. In the hardware implementation, the -1 is represented as 0 to perform multiplications as XNOR logic operations. The weight and input feature maps are binarized by the $sign()$ function (recall equation 2.7).

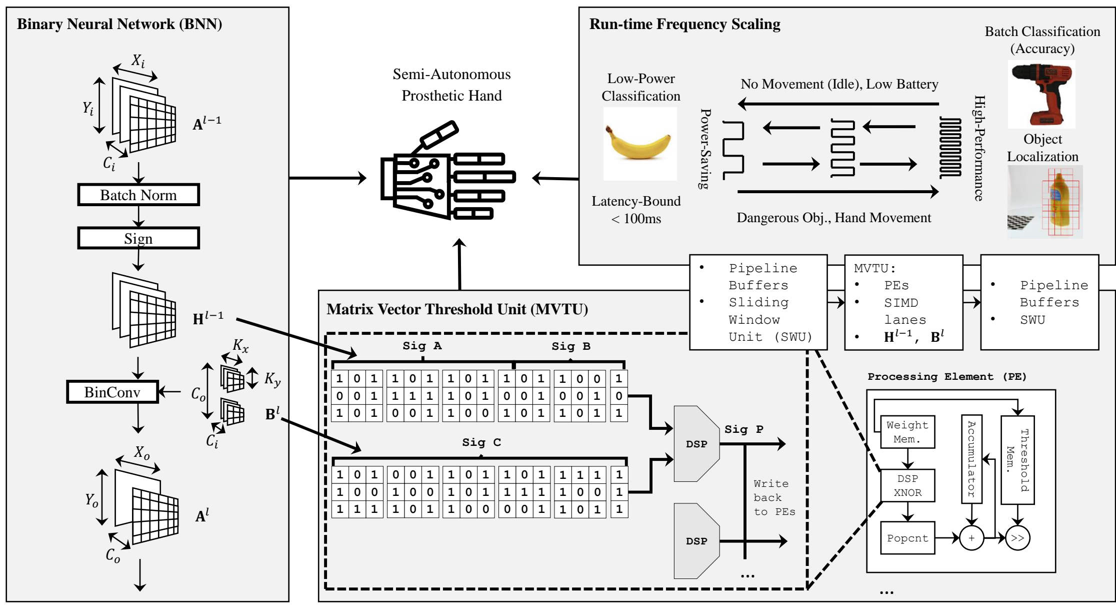


Figure 5.2: Overview of Binary-LoRAX: BNN tensor slices are fed into DSP blocks which perform high-throughput XNOR operations. DSP results are forwarded to the PEs of a matrix-vector-threshold unit (MVTU). A single MVTU of the pipeline is shown for compactness. Runtime frequency scaling allows high-performance functions, or power-saving mode.

The $sign()$ function blocks the flow of gradients during training due to its derivative, which is zero almost everywhere. To overcome the gradient flow problem, the $sign()$ function is approximated during back-propagation by a STE [59].

In the simplest case, the estimated gradient g_b could be obtained by replacing the derivative of $sign()$ with the hard $tanh$, which is equivalent to the condition $g_w = g_b$ when $|w| \leq 1$ [60], as shown in equation 5.1.

$$g_w = g_b 1_{|w| \leq 1} \tag{5.1}$$

As mentioned previously, batch normalization of the input elements $a^{l-1} \subset \mathbf{A}^{l-1}$, before the approximation into the binary representation $h^{l-1} \subset \mathbf{H}^{l-1} \in \mathbb{B}^{X_i \times Y_i \times C_i}$ is crucial to achieve effective training. An advantage of BNNs is that the result of the batch normalization operation will always be followed by a $sign()$ operation (as shown in figure 5.2). The result after applying both functions is always constrained to two values, $\{-1, 1\}$, irrespective of the input. This makes the precise calculation of batch normalization wasteful on embedded hardware. Based on the batch normalization statistics collected at training time, a *threshold* point τ_{thold} can be defined, where an activation value $a^{l-1} \geq \tau_{thold}$ results in 1, otherwise -1 [103]. This allows the implementation of the typically costly normalization operation as a simple magnitude comparison operation on hardware.

5.1.3.2 Hardware Architecture

The baseline hardware architecture is provided by the Xilinx FINN framework [103]. The hardware design space has many degrees of freedom for compute resources, pipeline structure, number of PEs and SIMD-lanes, among other parameters. The streaming architecture is composed of a series of matrix-vector-threshold units (MVTUs) to perform the XNOR, popcount and threshold operations mentioned in section 5.1.3.1. In figure 5.2, a single MVTU is shown in detail, containing two PEs with 32 SIMD-lanes each. A detailed view of a single PE is also provided in the same figure. For convolutional layers, a sliding-window unit (SWU) reshapes the binarized activation maps $\mathbf{H}^{l-1} \in \mathbb{B}^{X_i \times Y_i \times C_i}$ into interleaved channels of $h^{l-1} \subset \mathbf{H}^{l-1}$, to create a single wide input feature map memory, that can efficiently be accessed by the subsequent MVTU and operated upon in a parallel manner. Max-pool layers are implemented as Boolean OR operations, since a single binary “1” value suffices to make the entire pool window output equal to 1.

A single MVTU is solely responsible for a single layer in the BNN, and is composed of single or multiple PEs, each having their own SIMD-lanes. The SIMD-lanes determine the throughput of each PE for the XNOR operation.

The choice of PEs and SIMD-lanes determines the latency and hardware resource utilization of each layer (i.e. MVTU) on the hardware architecture. Instantiating too many PEs can result in many underutilized FPGA BRAMs, while too few PEs result in a slower processing rate with better BRAM utilization. Increasing the number of PEs beyond a certain number causes the synthesis tool to map the memories to LUTs instead of BRAMs, since each PE gets a smaller slice of the total weights **B**. This adds another dimension of design complexity, as the target FPGA’s LUT and BRAM count can be balanced against throughput and utilization efficiency. A layer’s poorly dimensioned MVTU can result in an inefficient pipeline, leading to poor overall throughput. Throughput in a streaming architecture is heavily influenced by the slowest MVTU of the accelerator, as it throttles the rate at which results are produced when the pipeline is full. On the other hand, latency is dependent on the time taken by all the MVTUs of the architecture as well as the intermediate components between them (e.g. SWU, pooling unit, etc.).

Choosing the correct number of PEs and SIMD-lanes for each layer becomes a design problem of balancing the FPGA’s resources, the pipeline’s efficiency (throughput and latency), and potentially the choice of layers in the BNN (i.e. task-related accuracy). The number of resources on the FPGA is limited, especially in the context of low-power prosthetics, making these aspects important in planning the deployment with a HW-DNN co-design approach.

5.1.3.3 Runtime Dynamic Frequency Scaling

In the previous section, the importance of defining the number of layers (BNN design) and PE/SIMD-lanes per MVTU (HW design) was outlined. To enable efficient performance of the semi-autonomous prosthesis, a further aspect must be considered next to resource utilization and latency, namely the power consumption of the classifier. Prosthetic devices are meant to be used on a day-to-day basis, making high power consumption a prohibitive aspect to their practicality. For this reason, the classifier is adapted with the ability to change its operating frequency dynamically at runtime. The purpose is to avoid running the classifier continually at its

full capacity, but rather scale down its performance (in terms of latency) for more efficient use of the available energy supply. Dynamic power in complementary metal-oxide-semiconductor (CMOS) scales roughly with frequency following $P_{dyn} \approx \alpha_{sw} f \cdot CV_{dd}^2$, where α_{sw} is the switching activity, f is the frequency, C the effective capacitance and V_{dd} the supply voltage.

In case of our target Xilinx Zynq system-on-chip (SoC) boards, the programmable logic (PL), on which the hardware acceleration is implemented, is clocked through phase-locked loops (PLLs) controlled by a CPU-based processing system (PS). The PS can manipulate the PL's clock by writing into special registers, whose values act as frequency dividers to the PLLs. As an example, the motion of the prosthetic hand can be captured through simple sensors which are monitored by the PS. Based on this motion, the PS can drive up the frequency of the classifier and prepare for a low-latency, high accuracy classification (based on a mean classification of a batch of frames). In case of a fragile or perilous object, the lower risk of a false classification can reduce the chances of an improper grasp. The PS can also trigger the object localization task by splitting the view into multiple small images and classifying them with high throughput. This is elaborated in section 5.1.4.3. These high-performance features may extend the use of Binary-LoRAX to other semi-autonomous prostheses and/or applications. Conversely, the PS may monitor the remaining battery power or system temperature and switch the classifier to low-power mode.

5.1.3.4 SIMD Binary Products on DSP Blocks

In resource constrained platforms, the available hardware must be used effectively. Smaller FPGAs that have a few thousands of LUTs can easily run into synthesis issues, even with small network architectures. Since DSP blocks are not heavily utilized when synthesizing FINN-based accelerator designs, they presented a good alternative to LUT resources for executing the parallel XNOR operations of the accelerator.

A script was developed to parse through the HDL files generated by HLS, to find all the signals involved in XNORs on the accelerator. The connections between the operand signals and the output registers are removed, then primitive DSP modules are instantiated with the correct wiring to operate in the binary mode in a handcrafted manner, as described section 4.1. The operand signals of h^{l-1} and b^l are arranged into the aforementioned A:B and C signals and connected into the DSPs. The wide output P signal is then split and passed back into the next stages of the PE.

5.1.4 Evaluation

5.1.4.1 Experimental Setup

Binary-LoRAX is evaluated on 25 objects from the YCB dataset [130], improving upon previous work by 12 objects [131]. The dataset is augmented through scale, crop, flip, rotate and contrast operations. The masks provided with the dataset are used to augment the background with random Gaussian noise. The dataset is expanded to 105K images for the 25 classes. The images are resized to 32×32 pixels similar to the CIFAR-10 [27] dataset. The BNNs are trained up to 300 epochs, unless learning saturates earlier. Evaluation is performed on a 17.5K test set. The BNN architectures v-CNV, m-CNV, and μ-CNV, detailed in table A.1, are trained according to the method in [60]. Each convolutional and fully-connected layer is followed by batch normalization

and activation layers except for the final layer. Convolution groups “1” and “2” are followed by a max-pool layer. The target SoC platforms for the experiments are the XC7Z020 (Z7020) for v-CNV and m-CNV prototypes, and XC7Z010 (Z7010) for μ-CNV. All prototypes are finally deployed on the Z7020 SoC. Power, latency and throughput measurements are taken directly on a running system. The power is measured at the power supply of the board (includes both PS and PL). Latency measurements are performed end-to-end on the accelerator covering the classifier’s total time for an inference, while throughput is the classification rate when the accelerator’s pipeline is full. Note that throughput is higher than the latency rate due to the streaming architecture working on *multiple images* concurrently in different parts of its pipeline when it is full.

5.1.4.2 Design Space Exploration

Considering two embedded SoC platforms, the Z7020 and the more constrained Z7010, three Binary-LoRAX prototypes were investigated: v-CNV, m-CNV and μ-CNV. The CNV network is based on the architecture in [103] inspired by VGG-16 [40] and BinaryNet [60]. m-CNV and μ-CNV have a similar architecture, with fewer channels, for faster inference and to fit the Z7010 respectively. For the prosthetic hand, latency is more critical than throughput. On the Z7010, the number of PEs and SIMD lanes were chosen to minimize end-to-end latency accordingly.

In table 5.1, we report the details of the CNV network with (1,2) and (2,2) bits for weights and activations respectively. The fully-binarized CNV (1,1) network achieved a comparable accuracy of 99.82% on the YCB graspable object dataset, showing the effectiveness of BNNs for this task, and the potential to add more classes in future work.

In the bottom half of table 5.1, the hardware utilization for the Binary-LoRAX prototypes is provided. For the v-CNV network, a reduction of 2386 (9%) LUTs can be observed from the regular CNV [103]. For the constrained Z7010, such reductions can make a previously non-synthesizable design realizable after moving XNOR operations to DSPs. The increase in DSP usage can be justified as they are not the bottleneck for synthesizable designs in our case. It is important to note that μ-CNV was synthesizable on the Z7010 *only after* moving the XNOR operations to the DSPs, as proposed in section 4.1.

5.1.4.3 Runtime Dynamic Frequency Scaling

Prosthetic devices used on a daily basis must offer high performance for safety and convenience, while minimizing power dissipation to increase the continuous usage time before charging. Referring back to table 5.1, two values are reported (↕) for power, latency and throughput per Binary-LoRAX prototype, for high-performance and power-saving modes. At 2 MHz, Binary-LoRAX’s v-CNV achieves a reduction of up to 16% in power consumption with runtime frequency scaling compared to standard CNV [103]. This translates to an improvement in battery-life of up to 19%. In high-performance mode, a latency of only 0.45 ms is consumed by the m-CNV network at 125 MHz. This reduces latency by 99.7% compared to the work in [131]. Considering the performance/Watt efficiency metric, Binary-LoRAX’s m-CNV achieves 2318 frames/Watt compared to 20 frames/Watt in [131]. With an optimal controller delay for myoelectric prostheses of 125 ms [132], all Binary-LoRAX prototypes consume $<1\%$ of the total time, leaving more

Table 5.1: Hardware results of design space exploration. Power is averaged over a period of 100 seconds of operation.

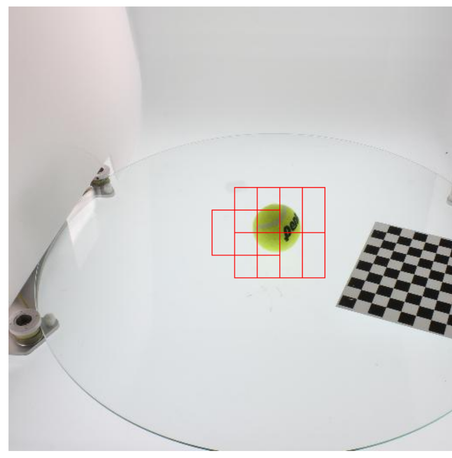
Configuration (W,A)-bits|BNN	**Freq.** MHz	**LUT**	**BRAM**	**DSP**	**Power** [W]	**Latency** [ms]	**Throughput** [FPS]	**Acc.** [%]
(8,8) - [131]*	400	-	-	-	0.446	115	9	96.51*
(2,2) - CNV**	100	35718	140	32	2.217	4.87	860	99.91
(1,2) - CNV	100	40328	131.5	26	2.241	1.63	3049	99.89
(1,1) - CNV [103]	100	26060	124	24	2.212	1.58	3049	99.82
Binary-LoRAX: DSP XNOR + Frequency Scaling:								
(1,1) - v-CNV	$\updownarrow$ 2 / 111	23675	124	72	$\updownarrow$ 1.857 / 2.172	78.93 / 1.42	61 / 3388	**99.82**
(1,1) - m-CNV	$\updownarrow$ 0.7 / 125	21972	44.5	66	$\updownarrow$ 1.879 / 2.157	80.22 / **0.45**	28 / **4999**	98.99
(1,1) - μ-CNV	$\updownarrow$ 1 / 100	**11738**	**14**	**27**	$\updownarrow$ 1.824 / 2.028	80.64 / 0.81	16 / 1646	90.58

*: Running on ARM Cortex M7 (CPU frequency reported), accuracy for 13 classes, 72×72 input
**: Less PEs and SIMD lanes to fit the SoC

slack for post-processing, actuators and other parts of the system. In power-saving mode, the Binary-LoRAX prototypes run at 0.7-2 MHz and achieve an $\sim$80 ms latency, still leaving more than 36% of the allocated delay for the controller. It is important to note that in all the reported power measurements, roughly 1.65 W of power is consumed by the Z7020's ARM-Cortex A9 processor (PS) and the board. This leaves the isolated accelerator's power at roughly 0.2 W in power-saving mode for all configurations, making it very energy efficient. However, we report the overall power since the accelerator is still dependent on processor calls and preprocessing operations on the CPU. In future work, the PS power consumption can also be optimized to further reduce the classifier's overall power requirement.

In addition to the low latency of the high-performance mode, the high throughput of up to 4999 FPS can be used to improve the quality of the application. Instead of providing a single classification, the accelerator can pipeline the inference of many images (potentially from different sensors) and perform batch classification. The batch classification result will represent the highest class over all classifications, which in practice compose of slightly different angles, lighting and distance to the object, improving the chances of a correct classification. Multi-camera prosthetics proposed in [148] can benefit from the high throughput, as more data is gathered through the multiple camera setup.

Another use of the high-performance mode is object localization in multi-object scenes. A large input image can be sliced into several smaller images and reclassified [103]. The image can be reconstructed with bounded high confidence classifications. Figure 5.3 demonstrates the described function on Binary-LoRAX. This can help the prosthesis predetermine the location of different objects in a *far* scene, when the hand is not yet close to the graspable object. The approach also fits the training scheme, as the BNNs are trained on up-close images of the object



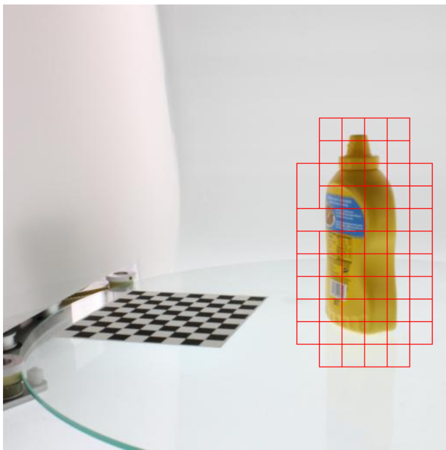


Figure 5.3: The large input image is sliced into smaller images and reclassified. High confidence classifications are bounded.

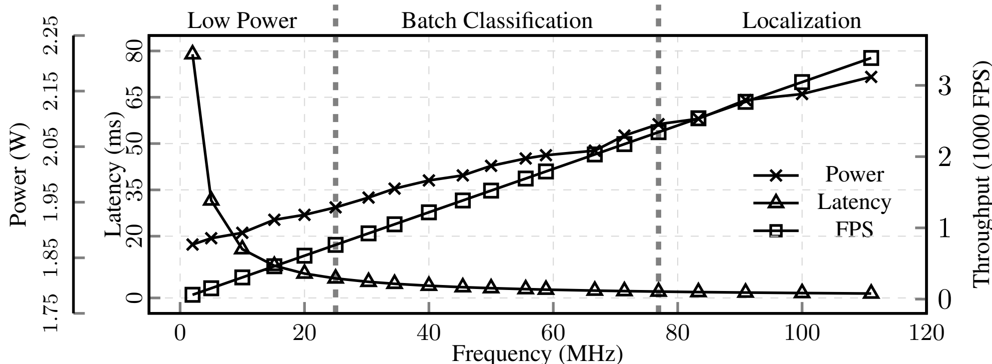


Figure 5.4: Runtime frequency scaling ranging from 2MHz to 111MHz for the υ-CNV prototype.

(soon before the grasp), while far scenes with no central object would be unrecognizable to the BNN. The individual slices of a far scene are similar to the up-close train images.

In figure 5.4, we perform a frequency sweep on the υ-CNV prototype, identifying different points of operation for different application requirements. The low-power region is considered to be below 1.90 W, while localization would require classification rates of above 2250 FPS for an input resolution of 320×240. Batch classification can be triggered in critical scenarios where a latency of $<$10 ms is needed.

We demonstrate the application of runtime frequency scaling in figure 5.5. The total power of the chip is measured for a duration of 80 seconds. At time = 15 s, we introduce a stimulus representing a dangerous object or similarly a signal from a motion sensor on the hand. The event triggers the classifier to high-performance mode for an observation period of 35 seconds. If no further event occurs, the classifier winds down to low-power mode at time = 50 s. Naturally, the intermediate frequencies shown in figure 5.4 can all be triggered for other scenarios or operating modes.

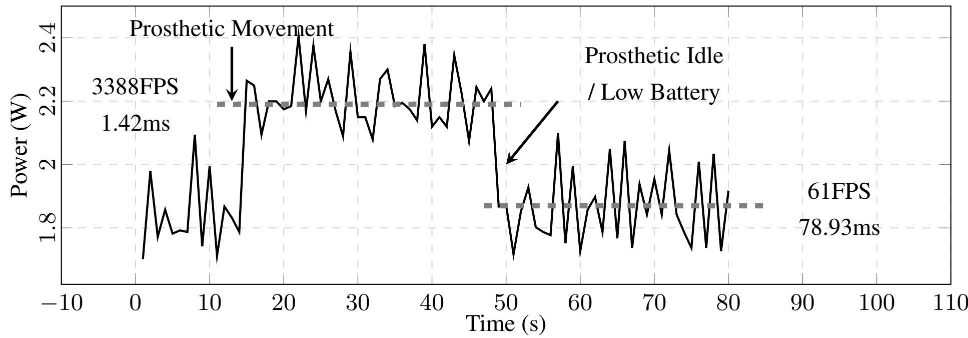


Figure 5.5: Runtime change in operation mode based on application scenario, e.g. motion, delicate object or low battery.

5.1.5 Discussion

A daily-used device, such as a prosthetic hand, must operate in different modes to suit daily application scenarios. This work presented a low-latency runtime adaptable XNOR classifier for semi-autonomous prosthetic hands. The high-performance and power-saving modes were enabled through runtime adaptable frequency scaling. Binary-LoRAX prototypes achieved over $\sim$99% accuracy on a 25-class problem from the YCB dataset, a maximum of 4999 FPS, and a latency of 0.45 ms. The low-power mode can potentially improve the battery-life of the classifier by 19% compared to an equivalent accelerator running continuously at full-power. This work demonstrated the use of expert knowledge and automation in a semi-automated HW-DNN co-design formulation. Particularly for the μ-CNV prototype, which can be synthesized on the heavily constrained Z7010's FPGA, the neural network had to be dimensioned such that the total number of MVTUs resulting from the layers did not consume LUT resources beyond those available on the PL, but still maintained high-accuracy for the graspable object classification task. Then, the PEs and SIMD-lanes of those MVTUs had to be chosen carefully to maintain the performance of the pipeline, but fit on the constrained FPGA. HLS performs the automated optimizations, based on the generated pipeline, to create the HDL components. Handcrafted, reconfigured DSPs were injected into each MVTU, to execute the highly parallel XNOR operations of the BNN and further reduce the total LUTs required by the accelerator. The injection of handcrafted DSPs into MVTUs was performed by an automated HDL parser script. This combination of handcrafted design of the BNN and the DSP, along with the automated pipeline optimizations of HLS and HDL parsers, led to a highly effective, co-designed solution which brought high-performance, intelligent classifiers to the semi-autonomous prosthetic hand.

5.2 BinaryCoP: Binary Neural Network-based COVID-19 Face-Mask Wear and Positioning Predictor on Edge Devices

Face masks have long been used in many areas of everyday life to protect against the inhalation of hazardous fumes and particles. They also offer an effective solution in healthcare for bi-directional protection against air-borne diseases. Wearing and positioning the mask correctly is essential for its function. CNNs can be an excellent solution for face recognition and classification of correct mask wearing and positioning. In the context of the ongoing[1] COVID-19 pandemic, such algorithms can be used at entrances to corporate buildings, airports, shopping areas, and other indoor locations, to mitigate the spread of the virus. These application scenarios impose major challenges to the underlying compute platform. The inference hardware must be cheap, small, and energy efficient, while providing sufficient memory and compute power to execute accurate CNNs at a reasonably low latency. To maintain data privacy of the public, all processing must remain on the edge-device, without any communication with cloud servers. To address these challenges, BinaryCoP, a low-power BNN classifier for correct facial-mask wear and positioning was proposed. The classification task was implemented on an embedded FPGA accelerator, performing high-throughput binary operations. Classification can take place at up to $\sim$6400 FPS and 2 W power consumption, easily enabling multi-camera and speed-gate settings. When deployed on a single entrance or gate, the idle power consumption is reduced to 1.65 W, improving the battery-life of the device. An accuracy of up to 98% for four wearing positions of the MaskedFace-Net dataset is achieved. To maintain equivalent classification accuracy for all face structures, skin-tones, hair types, and mask types, the algorithms were tested for their ability to generalize the relevant features over a diverse set of examples using the gradient-weighted class activation mapping (Grad-CAM) approach.

5.2.1 Efficient Deployment of CNNs for Mask Detection

The ongoing COVID-19 pandemic presents new challenges, which can be solved with the help of state-of-the-art computer vision algorithms [149, 150]. One of the simplest ways of mitigating the spread of the COVID-19 disease is wearing a face-mask, which can protect the wearer from direct exposure to the virus through the mouth and nasal passages. A correctly worn mask can also protect other people, in case the wearer is already infected with the disease. This bi-directional protection makes masks highly effective in crowded and/or indoor areas. Although face-masks have become a mandatory requirement in many public areas, it is difficult to ensure the compliance of the general public. More specifically, it is difficult to assert that the masks are worn correctly as intended, *i.e.* completely covering the nose, mouth and chin [151].

CNNs can provide better accuracy on problems with diverse features without having to manually extract said features [152]. This holds true only when the training dataset has a fair distribution of samples. Correctly identifying a mask on a person's face is a relatively simple task for these powerful algorithms. However, a more precise classification of the exact positioning of the mask and identifying the exposed region of the face is more challenging. To maintain

[1] at the time of writing

equivalent classification accuracy for all face structures, skin-tones, hair types, and mask types, the algorithms must be able to generalize the relevant features over all individuals.

The deployment scenarios for the CNN should also be taken into consideration. A face-mask detector can be set at the entrance of corporate buildings, shopping areas, airport checkpoints, and speed gates. These distributed settings require cheap, battery-powered, edge devices which are limited in memory and compute power. To maintain security and data privacy of the public, all processing must remain on the edge-device without any communication with cloud servers.

Minimizing power and resource utilization while maintaining a high classification accuracy is yet another HW-DNN co-design challenge which is tackled in this work. In this context, BinaryCoP (Binary COVID-mask Predictor) is an efficient BNN-based real-time classifier of correct face-mask wear and positioning. The challenges of the described application are tackled through the following contributions:

- Training BNNs on synthetically generated data to cover a wide demographic and generalize relevant task-related features. A high accuracy of $\sim$98% is achieved for a 4-class problem of mask wear and positioning on the MaskedFace-Net dataset [153].
- Deploying BNNs on a low-power, real-time embedded FPGA accelerator. The accelerator can idle at a low-power of 1.65 W on single entrances and gates or operate at high-performance ($\sim$6400 FPS) in crowded multi-gate settings, requiring $\sim$2 W of power.
- The BNNs are analyzed through Grad-CAM to improve interpretability and study the features being learned.

5.2.2 COVID-19 Face-Mask Wear and Positioning

Correctly worn masks play a pivotal role in mitigating the spread of the COVID-19 disease during the ongoing pandemic [154]. Members of the general public often underestimate the importance of this simple yet effective method of disease prevention and control. Researchers and data scientists in the field of computer vision have collected data to train and deploy algorithms which help in automatically regulating masks in public spaces and indoor locations [155, 156]. Although large-scale natural face datasets exist, the number of real-world masked images is limited [155]. Wang et al. [156] extended their masked-face dataset with a Simulated Masked Face Recognition Dataset (SMFRD), which is synthetically generated by applying virtual masks to existing natural face datasets. Cabani et al. [153] improved the generation of synthetically masked-faces by applying a deformable mask-model onto natural face images with the help of automatically detected facial key-points. The key-points of the deformable mask-model can be matched to the key-points of the face, allowing the application of the mask in a variety of ways. This allows the dataset generation process to further generate examples of incorrectly worn masks, such as chin exposed, nose exposed, or nose and mouth exposed.

5.2.3 BNN Interpretability with Grad-CAM

The output of the convolutional layers in a CNN contains localized information of the input image, without any prior bias on the location of objects and features during training. This

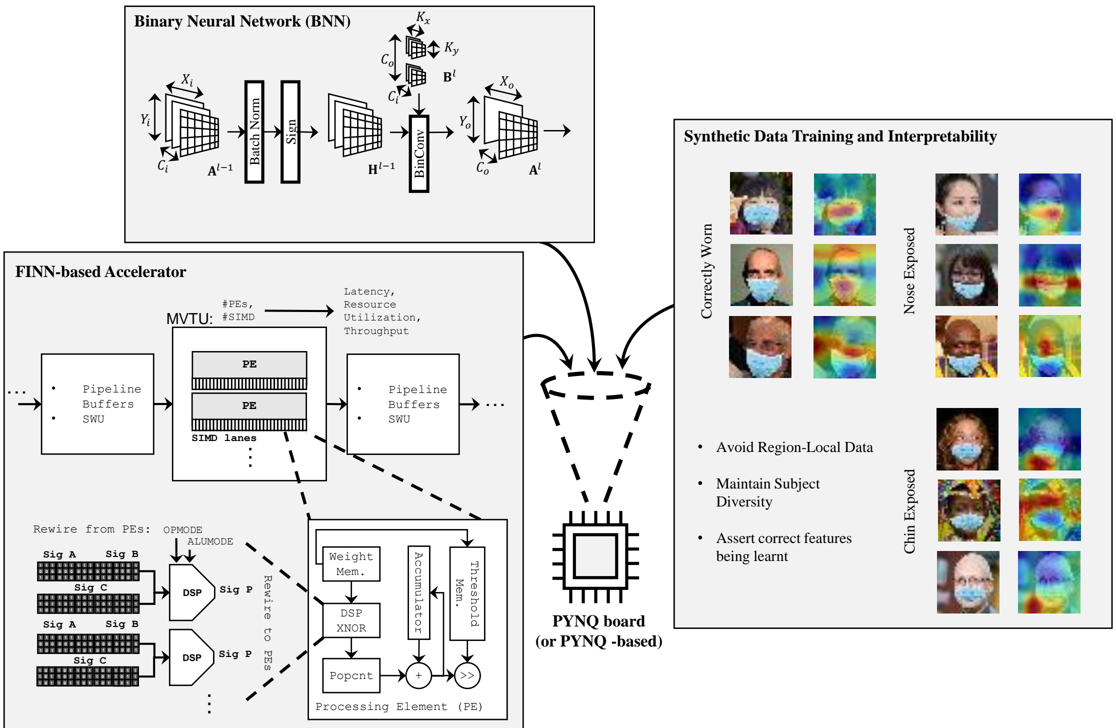


Figure 5.6: Main components of BinaryCoP. The BNN requires low memory and provides good generalization. The FINN-based accelerator allows for privacy-preserving edge deployment of the algorithms without sacrificing performance. The synthetic data helps in maintaining a diverse set of subjects and Grad-CAM can be used to assert the features being learned.

information can be captured using class activation mapping (CAM) [157] and Grad-CAM [158] techniques. To apply CAM, the model must end with a global average pooling layer followed by a fully-connected layer, providing the logits of a particular input. The BNN models investigated in this work operate on a small input resolution of 32×32, and achieve a high reduction of spatial information without incorporating a global average pooling layer. For this reason, the Grad-CAM approach is better-suited to obtain visual interpretations of BinaryCoP's attention and determine the important regions for its predictions of different inputs and classes.

To obtain the class-discriminative localization map, we consider the activations and gradients for the output of the Conv_2_2 layer (shown in table A.1), which has spatial dimensions of 5×5. We use average pooling for the corresponding gradients and reduce the channels by performing Einstein summation as specified in [158]. With this approach the base networks do not need any modifications or retraining. Due to the synthetically generated dataset used for training, we expect BinaryCoP models to generalize well against domain shifts.

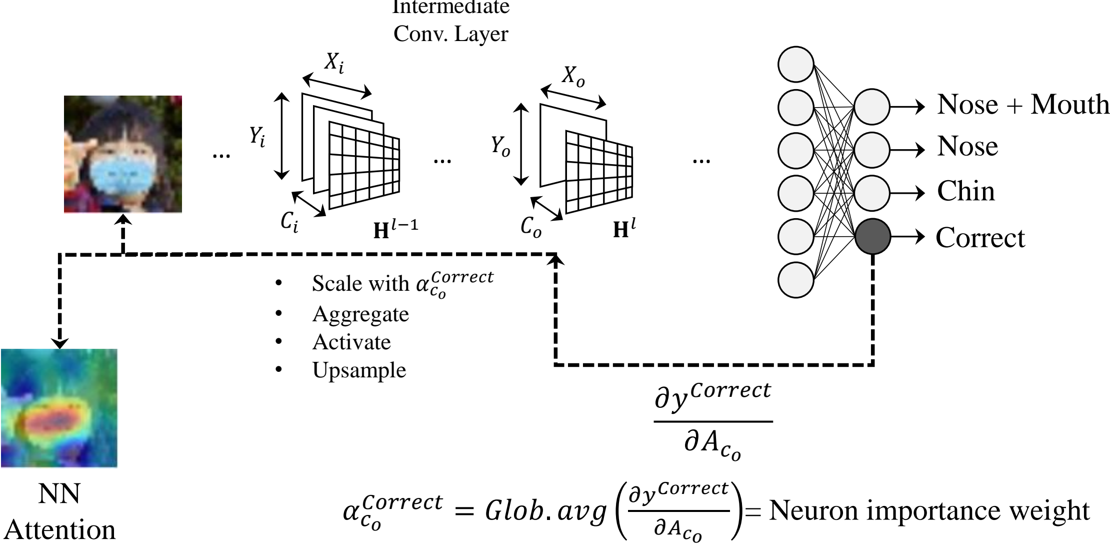


Figure 5.7: The Grad-CAM approach used to assert that correct and reasonable features are being learned from the synthetic data.

5.2.4 Evaluation

5.2.4.1 Experimental Setup

BinaryCoP is able to detect the presence of a mask, as well as its position and correctness. This level of classification detail is possible through the more detailed split of the MaskedFace-Net dataset [153] from 2 classes, namely correctly masked face dataset (CMFD) and incorrectly masked face dataset (IMFD), to 4 classes of CMFD, IMFD Nose, IMFD Chin, and IMFD Nose and Mouth. The dataset suffers from high imbalance in the number of samples per class. From the total 133,783 samples, roughly 5% of the samples are IMFD Chin, and another 5% samples are IMFD Nose and Mouth. CMFD samples make up 51% of the total dataset while IMFD Nose makes up 39%. The dataset in its raw distribution would heavily bias the training towards the two dominant classes. To counter this, we randomly sample the larger classes CMFD and IMFD Nose to collect a comparable number of examples to the two remaining classes, IMFD Chin and IMFD Nose and Mouth. The evenly balanced dataset is then randomly augmented with a varying combination of contrast, brightness, gaussian noise, flip and rotate operations. The final size of the balanced dataset is 110K train and validation examples and 28K test samples. The images are resized to 32×32 pixels, similar to the CIFAR-10 [27] dataset. The BNNs are trained up to 300 epochs, unless learning saturates earlier. The FP32 variant used for the Grad-CAM comparison is trained for 175 epochs due to early learning saturation (98.6% final test accuracy). We trained the BNN architectures shown in table A.1 according to the method described in section 5.1.3.1. The target SoC platform for the experiments is the Xilinx XC7Z020 (Z7020) chip on the PYNQ-Z1 board. The μ-CNV design can also be synthesized for the more constrained XC7Z010 (Z7010) chip, when XNOR operations are offloaded to the DSP blocks as described in section 4.1.4.

Table 5.2: Hardware results of design space exploration. Power is averaged over 100s of operation.

Prototype	LUT	BRAM	DSP	Power [W] Idle	Power [W] Inf.	Thr.put [FPS]	Latency [ms]	Acc. [%]
CNV	26060	124	24	1.65*	2.212	3049	1.58	**98.10**
n-CNV	20425	**10.5**	14	1.65*	2.122	**6460**	0.31	93.94
μ-CNV	**11738**	14	27	1.65*	2.028	1646	0.81	93.78

*Required by the board and ARM-Cortex A9 processor. Accelerator is idle.

Power and throughput measurements are taken directly on a running system, in the same manner described in section 5.1.4.1.

5.2.4.2 Design Space Exploration

Three BinaryCoP prototypes are evaluated, namely CNV, n-CNV and μ-CNV. Architectural details of the networks can be found in table A.1. The CNV network is based on the architecture in [103] inspired by VGG-16 [40] and BinaryNet [60]. n-CNV is a downsized version for a smaller memory footprint, and μ-CNV has fewer layers to reduce the size of the synthesized design. All designs are synthesized with a target clock frequency of 100MHz.

Referring back to table A.1, the PE counts and SIMD-lanes for each layer (i.e. MVTU) are shown in sequence. For BinaryCoP-n-CNV, the most complex layer is Conv_1_2 with 3.6M `XNOR` and `popcount` operations. In figure 5.8, this layer is marked as the throughput setter, due to its heavy influence on the final throughput of the accelerator. Allocating more PEs for this layer's MVTU increases the overall throughput of the pipeline, so long as no other layer becomes the bottleneck. Enough resources are allocated for Conv_1_1 to roughly match Conv_1_2's latency. The FINN architecture employs a weight-stationary dataflow, since each PE has its own pre-loaded weight memory. When the total number of parameters of a given layer increases, it becomes important to map these parameters to BRAM units instead of logic. The deeper layers have several orders of magnitude fewer operations (OPs), but more parameters. For these layers, increasing the number of PEs fragments the total weight memory, leading to worse BRAM utilization and no benefit in terms of throughput. Here, choosing fewer PEs, with larger unified weight memories, leads to improved memory allocation, while maintaining rate-matching with the shallow layers (as seen figure 5.8), leaving the throughput gains from the initial PEs unhindered. The CNV architecture in [103] follows the same reasoning for PE and SIMD allocation. For μ-CNV, fewer PEs are allocated for the throughput-setters, as this prototype is meant to fit on embedded FPGAs with less emphasis on high frame rates.

In table 5.2, the hardware utilization for the BinaryCoP prototypes is provided. With μ-CNV, a significant reduction in LUTs is achieved, which makes the design synthesizable on the heavily constrained Z7010 SoC. The trade-off is a slight increase in the memory footprint of the BNN, as the shallower network has a larger spatial dimension before the fully-connected layers, increasing the total number of parameters after the last convolutional layer. The choice of

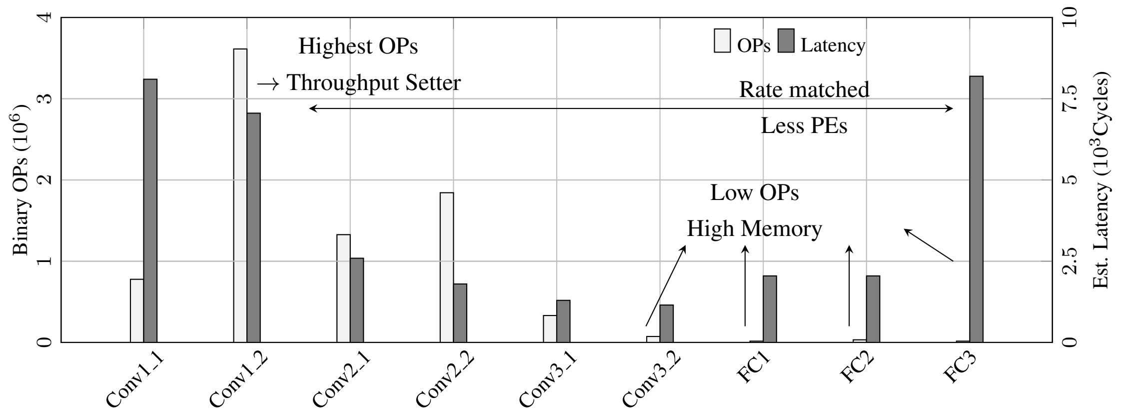


Figure 5.8: Binary operations and layer-wise latency estimates based on PE/SIMD choices for BinaryCoP-n-CNV.

PE count and SIMD lanes for the n-CNV prototype allow it to reach a maximum throughput of $\sim$6400 classifications per second when its pipeline is full. This high-performance can be used to classify images from multiple cameras in multi-gate settings. The inference power values reported in table 5.2 show a total power requirement of around 2 W for all prototypes. For single entrance/gate classifications, all prototypes have an idle power of around 1.65 W. In this setting, a classification needs to be triggered only when a subject is attempting to pass through the entrance where BinaryCoP is deployed. The idle power is required mostly by the processor (ARM-Cortex A9) on the SoC and the board (PYNQ-Z1). This can potentially be reduced further by choosing a smaller processor to pair with the proposed hardware accelerator. Although the PYNQ-Z1 board has no power measurement bus (PMBus) to isolate the power measurements of the FPGA from the rest of the components, we can infer that the hardware accelerator requires roughly 0.4 W for the inference task from the two measured power values in table 5.2. The current design is still dependent on the processor for pre- and post-processing, therefore the joint power is reported for fairness.

5.2.4.3 Grad-CAM and Confusion Matrix Analysis

The confusion matrix in figure 5.9 shows the generalization of BinaryCoP-CNV on all classes after balancing the dataset. As expected, it is extremely rare to mistake nose+mouth exposed with a correctly worn mask. Less critically, nose and nose+mouth have a slight misclassification overlap, still at only 2% of the total samples given for each class. Finally, the chin exposed and the correct class have some sample misclassifications ($\leq$1%), which could be attributed to the chin area being small in some images and hard to detect at low-resolution.

The output heat maps generated by Grad-CAM are analyzed to interpret the predictions of our BNNs with respect to the diverse attributes of the MaskedFace-Net dataset. In figure 5.10 till figure 5.13, columns 1 and 2 indicate the label and input image respectively. Columns 3, 4 and 5 highlight the heat maps obtained from the Grad-CAM output of BinaryCoP-CNV, BinaryCoP-n-CNV and a full-precision version of CNV with FP32 parameters. The heat maps are overlaid

True Class \ Predicted Class	Correct	Nose	N+M	Chin
Correct	7125 98%	41 1%	1 0%	90 1%
Nose	26 0%	7042 98%	94 2%	26 0%
N+M	4 0%	79 1%	5651 98%	9 0%
Chin	107 1%	41 1%	7 0%	7363 98%

Figure 5.9: Confusion matrix of BinaryCoP-CNV on the test set.

on the raw input images for better visualization. All raw images chosen have been classified correctly by all the networks, for fair interpretation of feature-to-prediction correlation.

In figure 5.10-a, the region of interest (RoI) for the *correctly masked* class is shown. BinaryCoP's learning capacity allows it to focus on key facial lineaments of the human wearing the mask, rather than the mask itself. This potentially helps in generalizing on other mask types. For the child example shown in the first row, the focus of BinaryCoP lies on the nose area, asserting that it is fully covered to result in a correctly masked prediction. Similarly, for the adult in row 2, BinaryCoP-CNV focuses on the upper edge of the mask to predict its coverage of the face. This also holds for our small version of BinaryCoP, with significantly reduced learning capacity. The RoI curves finely above the mask, tracing the exposed region of the face. In the third-row example, BinaryCoP-CNV falls back to focusing on the mask, whereas BinaryCoP-n-CNV continues to focus on the exposed features. Both models achieve the same prediction by focusing on different parts of the raw image. In contrast to the BinaryCoP variants, the FP32 model seems to focus on a combination of several different features on all three examples. This can be attributed to its larger learning capacity and possible overfitting.

In figure 5.10-b, we analyze the Grad-CAM output of the *uncovered nose* class. BinaryCoP-CNV and BinaryCoP-n-CNV focus specifically on two regions, namely the nose and the straight upper edge of the mask. These clear characteristics cannot be observed with the oversized FP32 CNN. In figure 5.10-c, the results show the RoI for predicting the *exposed mouth and nose* class. All models seem to distribute their attention onto several exposed features of the face. figure 5.10-d shows Grad-CAM results for *chin exposed* predictions. Although the top region of the mask points upwards, similar to the correctly worn mask, the BNNs pay less attention to this region and instead focus on the neck and chin. With the full-precision FP32 model, it is difficult

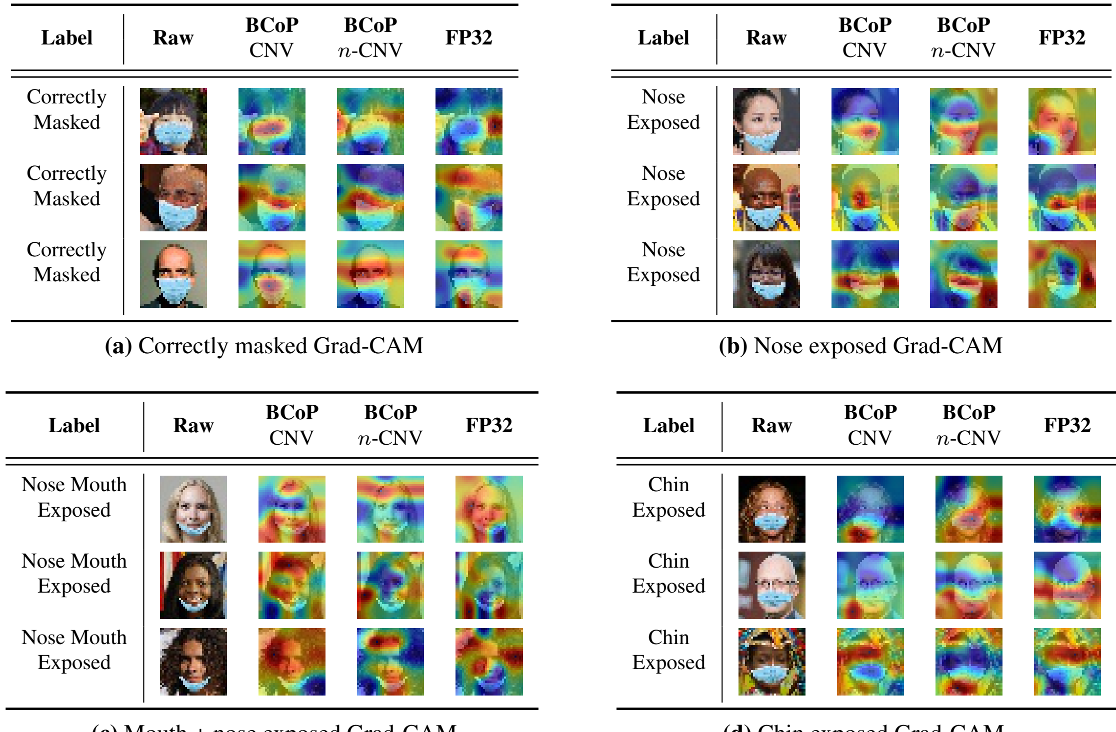


Figure 5.10: Grad-CAM output of two BinaryCoP variants and a FP32 CNN. Results are collected for all four wearing positions on a diverse set of individuals. Binarized models show distinct regions of interest which are focused on the exposed part of the face rather than the mask. The FP32 model is difficult to interpret in some cases. **It is recommended to view this figure in color.**

to interpret the reason for the correct classification, as little to no focus is given to the chin region, again hinting at possible overfitting.

Beyond studying the BNNs' behavior on different class predictions, the attention heat maps can be used to understand the generalization behavior of the classifier. In figure 5.11 to figure 5.13, BinaryCoP's generalization over ages, hair colors and head gear is tested, as well as complete face manipulation with double-masks, face paint and sunglasses. In figure 5.11, the smaller eyes of infants and elderly do not hinder BinaryCoP's ability to focus on the top region of the correctly worn masks. In figure 5.12, BinaryCoP-CNV shows resilience to differently colored hair and head-gear, even when having a similar light-blue color as the face-masks (row 2 and 3). In contrast, the FP32 model's attention seems to shift towards the hair and head-gear for these cases. Finally, in figure 5.13, both BinaryCoP variants focus on relevant features of the corresponding label, irrespective of the obscured or manipulated faces. This qualitatively shows that the complex training of BNN, along with their lower information capacity, constrains them to focus on a smaller set of relevant features, thereby generalizing well for unprecedented cases.

Label	Raw	BCoP CNV	BCoP n-CNV	FP32
Correctly Masked				
Correctly Masked				
Correctly Masked				

Figure 5.11: Grad-CAM results for age generalization. **It is recommended to view this figure in color.**

Label	Raw	BCoP CNV	BCoP n-CNV	FP32
Correctly Masked				
Correctly Masked				
Nose Exposed				

Figure 5.12: Grad-CAM results for hair/headgear generalization. **It is recommended to view this figure in color.**

5.2.4.4 Comparison with Other Works

As mentioned in Sec 5.2.2, detection of masks has piqued the interest of many researchers in the computer vision domain due to its relevance in the context of the ongoing COVID-19 pandemic. NVIDIA proposed mask recognition using object detection models [159]. These models require INT8 or Float-16 numerical precision, with ResNet18 as a backbone for input images of 960$\times$544 pixel resolution. The complexity is orders of magnitude higher than the models proposed in this work. A head-to-head comparison is difficult to make due to differences in the training approach, the CNN model, the datasets used, and the application requirements. The networks are trained to predict only two classes (mask, no mask), which is a simpler problem compared to the exact positioning supported by BinaryCoP. However, the localization and higher resolution makes it a more computationally complex task overall. With the NVIDIA Jetson Nano hardware, which typically requires $\sim$10 W of power on intensive workloads, a frame rate of 21 FPS is achieved. The more powerful 25 W Jetson AGX Xavier can achieve up to 508 FPS. Compared to the NVIDIA approach [159], BinaryCoP is targeted at low-power, embedded applications with peak inference power of $\sim$2 W and high classification rates of up to $\sim$6400 FPS on smaller resolution input images. It is worth noting that BinaryCoP can also classify high resolution images containing multiple individuals, by slicing the input into many 32$\times$32 frames and batch

Label	Raw	BCoP CNV	BCoP n-CNV	FP32
Correctly Masked				
Correctly Masked				
Chin Exposed				
Nose Mouth Exposed				
Nose Mouth Exposed				

Figure 5.13: Grad-CAM results for face manipulation with double-masks, face paint and sunglasses. **It is recommended to view this figure in color.**

processing them. This application makes use of the high-throughput results presented in table 5.2. Another approach proposed by Agarwal et al. [160] achieves the task of detecting a range of personal protective equipment (PPE). Processing takes place on cloud servers, which could raise privacy and data safety concerns in public settings. Wang et al. [161] propose an in-browser serverless edge computing method, with object detection models. The browser-enabled device must support the WebAssembly instruction format. The authors benchmarked their approach on an iPad Pro (A9X), an iPhone 11 (A13) and a MacBook pro (Intel i7-9750H), achieving 5, 10 and 20 FPS respectively. Needless to say, these devices (or similar) are expensive and cannot be placed in abundance in public areas. Similarly, [162] offers an Android application solution, which is suitable for users self-checking their masks. In this case, low-power, edge-hardware, and continuous surveillance are not emphasized.

BinaryCoP offers a unique, low-power, high-throughput solution, which is applicable to cheap, embedded FPGAs. Moreover, the BinaryCoP solution is not constrained to FPGA platforms. Software-based inference of BinaryCoP is also possible on other low-power microcontrollers, with binary instructions. Training on synthetic data enables generating more samples with different mask colors, shapes, and sizes [163], further improving the generalizability of the BNNs, while keeping real-world data available for fine-tuning stages.

5.2.5 Discussion

Applying BNNs to face-mask wear and positioning prediction solves several challenges such as maintaining data privacy of the public by processing data on the edge-device, deploying the classifier on an efficient XNOR-based accelerator to achieve low-power computation, and minimizing the neural network's memory footprint by representing all parameters in the binary domain, enabling deployment on low-cost, embedded hardware. The accelerator requires only $\sim$1.65 W of power when idling on single gates/entrances. Alternatively, high-performance

is possible, providing fast batch classification on multiple gates and entrances with multiple cameras, at $\sim$6400 FPS and 2 W of power. An accuracy of up to 98% for four wearing positions of the MaskedFace-Net dataset was achieved. The Grad-CAM approach was used to study the features learned by the classifier. The results showed the classifier's high generalization ability, allowing it to perform well on different face structures, skin-tones, hair types, and age groups. BinaryCoP reused many of the semi-automated HW-DNN co-design concepts first introduced in Binary-LoRAX. Additionally, semi-automated design was incorporated in the training loop, where synthetic data was generated in an automated manner, and interpretability tools, such as Grad-CAM, allowed the human to verify the features being learned. Optimizations from compilers coupled with human-engineered DNN architectures resulted in high FPS performance and task-accuracy, while maintaining low power in a privacy-preserving, edge setting. Both BinaryCoP and Binary-LoRAX have shown how semi-automated HW-DNN co-design can combine human-expertise and compiler optimizations to achieve highly efficient AI that can impact human lives in a positive way.

6 Fully-Automated Co-Design

THE HARDWARE component libraries are fixed, the supported layers are known, the schedule of computations can be modeled accurately, and the execution metrics (power/latency) can be estimated with high fidelity. Now, all what is left to do is find the right configuration in a search space with over $\sim 10^{34}$ solutions [9]. Calling this solution a "needle in a haystack" is an understatement, unless the haystack in question is as big as the observable universe and has as much hay as we have stars. Fortunately, problems of this size emerge often in engineering and mathematical optimization, giving purpose to the well-established field of metaheuristics. In fully-automated co-design, search agents involving methods like genetic algorithms (GAs), Bayesian optimizers, and reinforcement learning (RL) are exploited to traverse such multi-dimensional, noisy search spaces and return sufficiently good solutions for the target application, in a reasonable amount of time. For these search agents to function properly, the prerequisites of a well-defined search space must be fulfilled, namely a finite set of design hyper-parameters and optimization criteria, as well as an evaluation function, model, or implementation. The evaluation function is necessary to assess the fitness of the design decisions in each step of the search algorithm. An evaluation model plays a crucial role here, as it must be accurate, high in fidelity, and fast enough to quickly provide the reward/fitness value of the design decisions, such that the search agent can quickly and correctly traverse to the next step. In this chapter, two works are discussed where this type of HW-DNN co-design is used. In HW-FlowQ [10], a multi-abstraction level HW-DNN co-design methodology is presented, where a tripartite search space is traversed and three levels of abstraction are proposed to converge to the final solution in a controlled manner. In AnaCoNGA [9], two nested GAs jointly search the hardware and neural network design spaces, while *reducing* the overall search time compared to a single GA searching the neural network design space. This novel, fully-automated co-design methodology was nominated for the best paper award at the Design, Automation and Test in Europe (DATE) conference in 2022.

6.1 HW-FlowQ: A Multi-Abstraction Level HW-CNN Co-Design Quantization Methodology

Model compression through quantization is commonly applied to CNNs deployed on compute and memory-constrained embedded platforms. As discussed in section 2.2.1, different layers of a CNN can have varying degrees of numerical precision for both weights and activations, resulting in a large search space. Together with the hardware design space, the challenge of finding the globally optimal HW-CNN combination for a given application becomes daunting. HW-FlowQ is a systematic approach that enables the co-design of the target hardware platform

and the compressed CNN model through quantization. The search space is viewed at three levels of abstraction, allowing for an iterative approach for narrowing down the solution space before reaching a high-fidelity CNN hardware modeling tool, capable of capturing the effects of mixed-precision quantization strategies on different hardware architectures (processing unit counts, memory levels, cost models, dataflows) and two types of computation engines (bit-parallel vectorized, bit-serial). To combine both worlds, a multi-objective non-dominated sorting genetic algorithm (NSGA-II) is leveraged to establish a Pareto-optimal set of quantization strategies for the target HW-metrics at each abstraction level. HW-FlowQ detects optima in a discrete search space and maximizes the task-related accuracy of the underlying CNN while minimizing hardware-related costs. The Pareto-front approach keeps the design space open to a range of non-dominated solutions before refining the design to a more detailed level of abstraction. With equivalent prediction accuracy, energy and latency are improved by 20% and 45% respectively for ResNet56, compared to existing mixed-precision search methods.

6.1.1 The Tripartite Search Space

Quantization simplifies the hardware components required to execute the multitude of arithmetic operations in modern CNNs, bringing benefits in terms of area, power and/or latency. Another advantage comes from lower bitwidth operands efficiently exploiting the data movement bandwidth available on the hardware. However, reduced bitwidth representations have a lower information capacity, losing the fine details captured by gradient propagation at training time. Finding a suitable quantization scheme that achieves the target benefits without degrading task-related accuracy becomes more challenging at extremely low bitwidths, e.g. $\leq$ 4-bits. Moreover, not all layers require equal numerical precision [7], rendering the search space even larger.

Evaluating a quantization strategy based on metrics that are loosely correlated to hardware benefits can lead to sub-optimal deployment setups. This has influenced recent works to optimize neural networks with HIL approaches [31, 7]. HIL-based optimization techniques necessitate the fully-functional and fabricated hardware to be readily available when optimizing the CNN, leaving little to no room for adjusting the target execution platform. Look-up table approaches have the same impediment, as filling up the table would require synthesized hardware measurements [84].

HW-FlowQ bridges the gap between hardware design and CNN quantization. To comprehensively address the design challenge, three large search spaces for the hardware design, the CNN structure, and the layer-wise quantization strategy need to be considered. Traversing all three in an unstructured manner would likely lead to sub-optimal solutions. In a top-down approach, HW-FlowQ narrows down the tripartite design space iteratively, with the help of an extensive HW-modeling tool and a GA. HW-FlowQ provides the flexibility of studying the impact of mixed-precision quantization and hardware-specific design parameters, without having to finalize the hardware platform in the early phases of development.

The contributions of this work can be summarized as follows:

- Developing a hardware *model-in-the-loop* quantization methodology which allows design space exploration of CNN quantization strategies and hardware platforms at different design phases.

- Exploring single and multi-objective genetic algorithms (SOGA, MOGA) for finding Pareto-optimal quantization strategies with respect to the underlying hardware platform.
- Modeling vectorized and bit-serial accelerators, with varying resources and dataflows for mixed-precision quantization, enabling HW-design exploration *during* CNN optimization.

6.1.2 Related Work

6.1.2.1 Quantization Methods

Many works have focused on improving the training scheme for quantized neural networks [59, 35, 164, 36]. DoReFa-Net [35] adapts the STE [59] and bounds the magnitude of latent weights and activations between 0 and 1. These latent datatypes are deterministically quantized. QNN [164] additionally quantizes the weights in the first and last layers of the CNN. The work in PACT [36] improves the training procedure of quantized neural networks by learning the optimal clipping level for the activations of each layer at training time. The dynamic clipping function allows for a larger representational capability than DoReFa-Net, thereby increasing the prediction accuracy. The aforementioned works explore the effect of quantization on the accuracy of the model and the achieved compression rate. Claims on the degree of improvement in energy efficiency or latency are difficult to make, as these complex metrics highly depend on the underlying hardware details (memory hierarchy, interconnect, technology, etc.).

6.1.2.2 Quantization & Search Schemes

Dong et al. [165] determine the bitwidths of activations and weights based on the Hessian spectrum obtained for individual layers. The quantization criterion is based on model size, which loosely correlates to final energy or latency for an inference execution. Wu et al. [166] propose a framework that learns quantization levels for each layer during the training period. To encourage lower-precision weights and activations, a loss term is associated with the quantization objective, capturing the benefits in OPs and model size, but not necessarily the effectiveness on hardware metrics, such as energy and latency. The authors of HAQ [7] propose an RL-based exploration scheme to determine HW-aware layer-wise quantization levels for weights and activations in a CNN model. The reward function is evaluated after executing the inference of the quantized CNN on an FPGA design. In APQ [84] a joint model architecture-pruning-quantization search is proposed. Pre-trained and pruned sub-networks are extracted from a once-for-all network. Then, mixed-precision quantization is applied and a predictor estimates the final accuracy. An energy/latency look-up table is used to provide the hardware feedback for a target accelerator during the search. Therefore, a set of pre-sampled data points is required from a readily available target hardware.

6.1.2.3 Hardware Modeling

Timeloop [4] is a HW-modeling tool that exploits CNN execution determinism to offer accurate estimates for a given hardware description. The tool provides the flexibility of changing the cost of hardware operations (e.g. read, write, MAC, etc.) and the memory hierarchy, among other

design parameters, as described in section 2.3.2.1 and 3.3.2. Interstellar [100] proposes formal dataflow definitions. Different to Timeloop, Interstellar uses the Halide programming language to represent the memory hierarchy and data movement constraints. Tetris [108] and Tangram [109] make use of a dataflow scheduler for DNN workloads on spatial accelerators to test the potential of other manipulations possible for their memory hierarchies. The mentioned works have proven the effectiveness of HW-modeling of DNN workloads. Nevertheless, other aspects have not been explored as thoroughly, such as adding the effects of layer-wise mixed-precision quantization on the resulting dataflow or supporting mixed-precision computation units. There is a need to extensively integrate hardware models with CNN optimization algorithms to aid the exploration of mixed-precision CNNs with respect to the hardware model under consideration, particularly when multiple hardware architectures are being considered as potential fabrication candidates.

6.1.2.4 Hardware-Software Co-Design

Jiang et al. [167] propose a HW-SW co-exploration framework, which includes the hardware's performance in the reward function of an RL agent and iteratively tunes both the CNN and hardware architectures. Fine HW-level details, such as scheduling schemes and quantized execution, are not explored, as the optimization loop targets optimally partitioning the CNN workload over a pool of FPGAs. The authors of [168] propose a HW-CNN co-design framework based on NAS. Fine-level details such as dataflow mapping are not studied, nor the effect of layer-wise quantization on bit-serial or vectorized hardware. The authors of ALOHA [169] propose a multi-phase GA-based framework for HW-aware CNN design, which takes into account the trade-off between model compression and adversarial robustness. Compression with mixed-precision or layer-wise quantization is not explored, nor the direct interactions between layer-wise quantization and different hardware designs and processing units. Although ALOHA considers the target hardware in the early phases of the CNN's design, the hardware itself is not actively co-designed in the flow. FINN-R [104] is an accelerator generator for quantized CNN inference on Xilinx FPGAs. Two types of architectures are supported: a pipelined dataflow architecture with appropriately sized MAC units for each layer to exploit layer-wise quantization, and a single multi-layer array that is reused across all layers. Candidate architectures are evaluated using an analytical model for resource usage and throughput. The framework is hardware focused, and does not explore the quantization and training search space of the CNN. NHAS [88] aims to find an optimal quantized CNN using an evolutionary algorithm. An efficient hardware architecture dimensioning for the compute array and on-chip memory is searched to accelerate a given set of benchmark workloads. The hardware evaluation follows a look-up table approach. Although NHAS promotes the idea of joint HW-CNN co-design, the approach can be enhanced by understanding the influence of each search decision against various aspects, such as the quantization method, mixed-precision accelerator choice, and scheduling schemes, with the help of hardware models during the optimization.

6.1.3 HW-FlowQ

HW-FlowQ is a HW-CNN co-design methodology which facilitates a top-down design approach, iteratively going through different levels of abstraction and performing some iterations of ex-

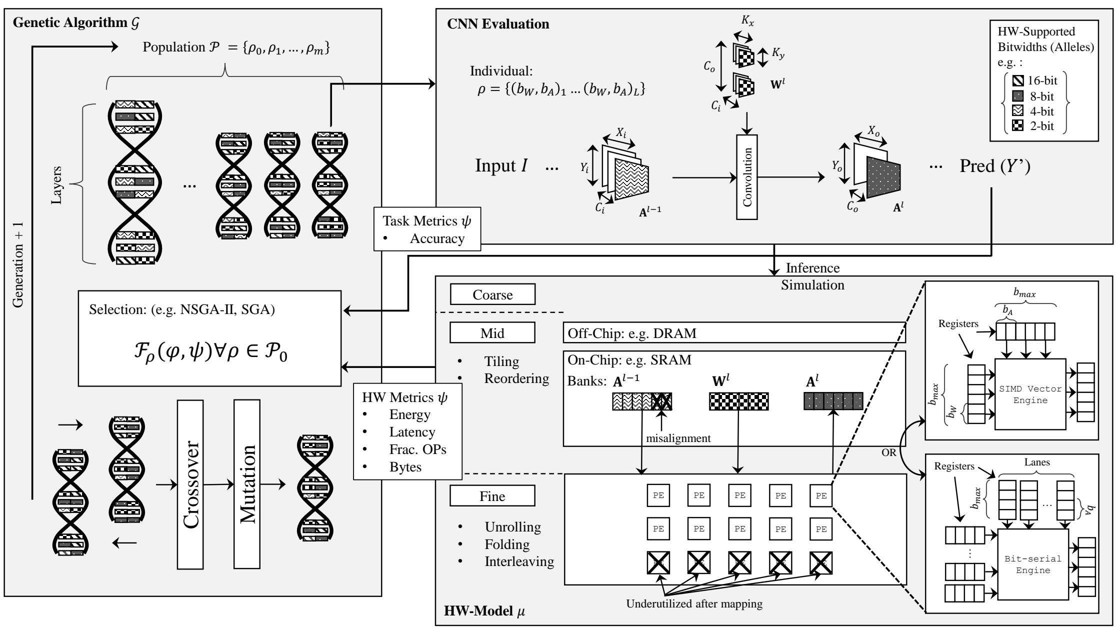


Figure 6.1: Overview of the HW-FlowQ methodology. Population $\mathcal{P}$ is evaluated on task-related accuracy ψ and hardware metrics φ. The three proposed HW-modeling abstraction levels: *Coarse*, *Mid* and *Fine*, enable the genetic algorithm $\mathcal{G}$ to consider the hardware metrics of the CNN relative to the current design phase.

ploration, before fixing some parameters and refining the design to one abstraction level lower. Moving away from hardware-in-the-loop and look-up table approaches in existing works, here, a *model-in-the-loop* design flow is implemented.

HW-FlowQ is based on an interaction between the individuals ρ of population $\mathcal{P}$, with the hardware model μ in the context of a genetic search algorithm $\mathcal{G}$ (figure 6.1). In detail, the genotype of an individual ρ encodes the layer-wise quantization levels for weights and activations (b_W, b_A) of the CNN. The individual's fitness F is measured through the HW-model estimations φ and CNN accuracy term ψ, computed w.r.t. the images and labels of a validation set. When $\mathcal{G}$ is a SOGA, the objectives φ and ψ are combined into a single cost function for fitness evaluation (section 6.1.3.2). $\mathcal{G}$ can also take the form of a MOGA, such as NSGA-II.

6.1.3.1 HW-Model Abstraction Levels

To enable the design steps of HW-SW co-design, different representations of the target HW platform need to be accessible based on the design phase. Starting from an abstracted, high-level representation makes it possible to coarsely search for HW-CNN combinations that may suit the application at hand. After some high-level parameters are fixed, a step of refinement can take place. This brings the exploration to a finer level of detail, but within the scope of the fixed parameters of the previous abstraction level (figure 6.2). More implementation-specific aspects can be considered after each refinement iteration. At any stage, if the exploration fails to find any suitable solutions, an abstraction step can take the design back to a coarser level and re-evaluate

the higher-level parameters which were set. This type of design flow is commonly used in VLSI design, where complex, large design spaces must be explored at different levels of abstraction, from system-level down to transistor logic [38, 2].

Three levels of abstraction are offered in HW-FlowQ, namely *Coarse*, *Mid* and *Fine*. Starting with *Coarse*-level optimization, the framework can be used to test the effect of quantization on differently shaped/sized CNNs, given as an input. The total computations required and the task-related accuracy can be evaluated. The CNN parameters at this level heavily influence the start of the co-design process, as they set the upper-bound of task-related accuracies possible, as well as the range of fractional operations and on-chip memory the hardware must accommodate. After quantization, if the target compression and/or task-related accuracy cannot be met, support for lower quantization levels needs to be considered and/or new CNN architectures need to be provided. The quantization training scheme can also be decided at this stage (e.g. DoReFa, PACT, QNN). It is important to note that HW-FlowQ does not constitute a NAS methodology, but is rather complementary to such techniques. As an example, a NAS framework can provide HW-FlowQ with high-accuracy CNN architectures as inputs at the *Coarse*-level. Then, HW-FlowQ can quantize them optimally for target hardware designs, as well as facilitate designing customized hardware for the proposed CNNs. Once the CNN(s) meets the high-level requirements, the *Mid*-level evaluates the feasibility of different memory hierarchies to buffer and move data before reaching the on-chip computation units. Parameters such as data transfer volumes, computation-to-communication (CTC) ratio and off-chip memory accesses can be searched. This information can help in deciding which off-chip to on-chip communication infrastructure and bandwidth is suitable to meet the application constraints. The CNN can further be quantized with this HW-model-in-the-loop setup, in order to close the gap between the hardware constraints and the computation/communication requirements, while maintaining the task-accuracy goals. Performing one more iteration of refinement takes the design to the *Fine*-level. At this stage, the hardware computation architecture can be defined. Precise information on the supported quantization levels, number of computation units available, register file sizes, supported data movements, and more, can be provided. HW-FlowQ provides high-fidelity estimates of the benefits that can be achieved on the prospective hardware design, for a particular quantization strategy (figure 6.5). Details on the *Fine*-level modeling are provided in the next sections.

Considering all design parameters holistically would imply searching *all* possible quantization strategies for *all* candidate CNNs (*Coarse*), on *all* possible on-chip/off-chip communication and on-chip memory sizes (*Mid*) for *all* possible dataflows, compute array sizes, multiplier

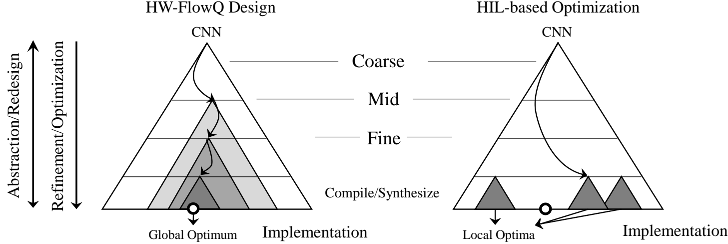


Figure 6.2: Iterative refinement increases the likelihood of finding the global optimum. Flow inspired by [2].

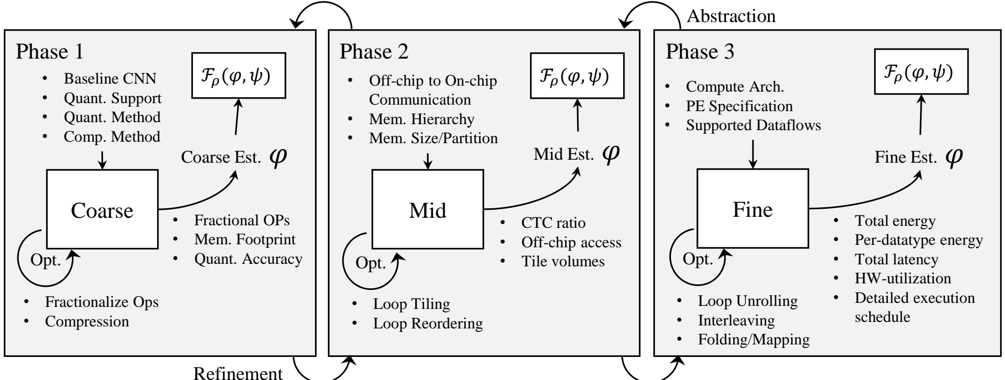


Figure 6.3: Input, output and optimization details of the HW-model μ abstraction levels used at each phase. After refinement, the inputs of the preceding phase are inherited to the next.

types and register dimensions (*Fine*). This would ultimately waste an immense amount of GPU hours, searching for solutions which could have been eliminated at the *Coarse*-level already. Additionally, with so many search parameters, the convergence of the search algorithm becomes more difficult to guarantee, potentially leading to sub-optimal results. To address this challenge, the step-wise optimization in HW-FlowQ's *Coarse*, *Mid* and *Fine* levels along with the Pareto-front-based quantization approach (NSGA-II) promotes a design-flow which leads to improved synergies in the final HW-CNN implementation and a more practical approach to searching the three large search spaces of CNN architecture, quantization strategy, and hardware design.

Figure 6.3 summarizes the inputs required at each level, the optimization that the HW-model μ can perform internally at each phase and the output estimates which can be used to evaluate the hardware-related fitness F_ρ of different individuals in population $\mathcal{P}$. Traversal between the levels is indicated by refinement and abstraction arrows. The decision on whether the search takes a step of refinement can be inferred from a list of application constraints. For example, a maximum number of fractional operations needs to be met, before a transition between *Coarse* to *Mid* can take place. Similarly, a desired off-chip communication constraint can be set, before the design transitions between *Mid* to *Fine*. If a certain constraint cannot be met, the framework must reconsider the inputs of the current level (e.g. at *Coarse* reconsider baseline CNN architecture, at *Mid* reconsider memory hierarchy, etc.). If changing the inputs of the level does not meet the targets, the inputs of the level above are reconsidered (abstraction). Through this progressive filtering of design decisions at each level, the output of the overall framework meets the desired application targets at the end of the flow.

6.1.3.2 Genetic Algorithm

Finding the correct layer-wise quantization strategy for both weights and activations with respect to a target HW-model is a complex problem which would benefit from gradient-free optimization due to the discrete nature of the solution space. The search space for the quantization strategy alone consists of Q^{2L} solutions, where Q is the set of possible quantization levels and L is the number of layers. Quantizing some layers leads to larger drops in accuracy than others,

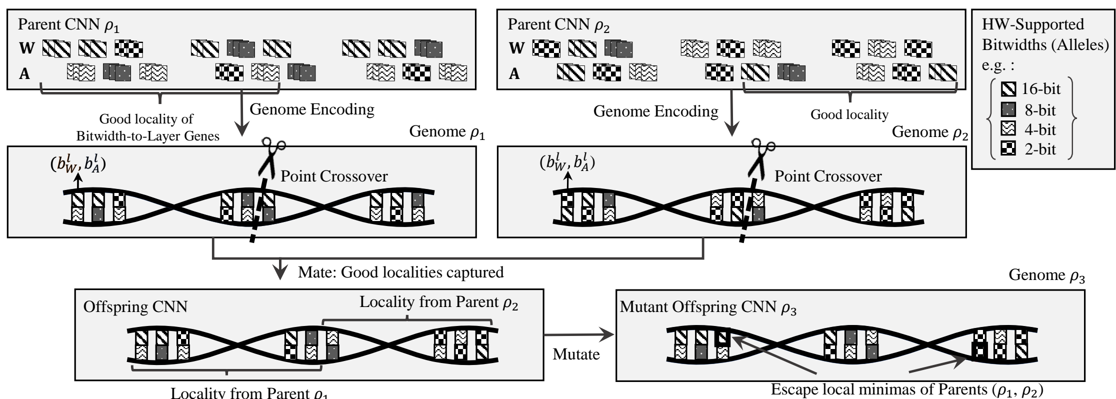


Figure 6.4: Layer-wise genome encoding allows for intuitive use of genetic operators (crossover, mutation) to capture and maintain good localities of bitwidth-to-layer encodings from two fit parents into their offspring.

and different accuracy drops can take place at different quantization levels for the same layer. Moreover, quantization strategies change the mapping and scheduling space of the CNN on the hardware, as explained in section 3.2. For example, a quantization strategy might make new schedules possible, which lead to sudden drops in latency and energy, as soon as a particular computation tile fits the on-chip memory after quantization. In this work, GAs are leveraged to tackle the quantization strategy search problem, as they are known to be resilient to noisy search spaces, quick to prototype, and do not need smooth, continuous search spaces to perform well.

Explicit, bijective encoding is used to create the genomes of potential solutions as shown in figure 6.4. A single genome represents a potential CNN quantization strategy and has as many genetic loci as there are layers in the CNN. Each genetic locus encapsulates a tuple of integer bitwidth values for weights and activations (b_W , b_A) at the corresponding layer. The set of possible alleles at each genetic locus is defined by the bitwidths supported by the HW-model, i.e. Q. Bitwidth-to-layer encoding can be captured intuitively in sequential genomes, which leads to a sensible use of GA operators, such as single-point crossover (example in figure 6.4). Neighboring CNN layers have higher feature correlation than distant layers. Therefore, quantized layer relationships encoded in neighboring genetic loci can survive in a population and be reused through single-point crossover to create more efficient offspring. The more fit the parents become throughout the generations, the better genetic localities they will have to create better individuals. Mutation further allows offspring to escape local minima of their parents.

Referring back to figure 6.1, on the top-left an initial population $\mathcal{P}_0$ is randomly generated at the start of the genetic algorithm $\mathcal{G}$, with each individual encoding the quantization levels of each layer of the CNN in its genes. The individuals of $\mathcal{P}_0$ are briefly fine-tuned and evaluated based on their task-accuracy ψ on a validation set (figure 6.1 top-right), as well as hardware estimates φ of the HW-model through inference simulation (figure 6.1 bottom-right). Based on the GA configuration, ψ and φ define the fitness of each individual $\rho^0 \in \mathcal{P}_0$. As depicted in figure 6.1, ψ and φ are fed back to a *selection* phase in $\mathcal{G}$, to constrain the cardinality of the population $|\mathcal{P}|$. Individuals survive this phase based on their fitness. Survivors are allowed to *mate* and produce offspring in $\mathcal{P}_1$, which inherit alleles from two survivor parents through *crossover*. A round of

mutation takes place, altering alleles of the offspring in $\mathcal{P}_1$. The population goes through the same phases of fitness evaluation, selection and crossover for the subsequent generations.

6.1.3.3 Fitness Evaluation

Single and multi-objective genetic algorithms are explored in this work. Both GAs share the same evolutionary flow described earlier, but are different in their observation of fitness. By definition, SOGA maximizes a *single* reward. Since our problem inherently involves multiple objectives (ψ and φ), a balanced reward function must be defined to combine them into a single fitness value F to apply SOGA.

$$F_\rho = \begin{cases} \left(1 - \frac{\psi^* - \psi}{t}\right) \cdot \log\left(\frac{\varphi^*}{\varphi}\right), & \text{if SOGA} \\ \psi, \varphi & \text{otherwise NSGA-II} \end{cases} \tag{6.1}$$

The fitness definition of SOGA in equation 6.1 is inspired by the cost function proposed in [37]. ψ^* and φ^* are the task-related accuracy and the hardware estimates of the uncompressed CNN, respectively. The function balances the improvements in hardware efficiency $\log(\varphi^*/\varphi)$ while trying to maintain task-related accuracy through the term $(1 - (\psi^* - \psi)/t)$. t sets a threshold on accuracy degradation, where a difference between ψ^* and ψ equal to or greater than t turns the accuracy term negative and renders the fitness F_ρ of individual ρ unacceptable.

In the case of NSGA-II optimization, the algorithm evaluates the *Pareto optimality* of each individual with respect to the population $\mathcal{P}$. This relieves the burden of crafting a single fitness function, which may not always guarantee a fair balance between multiple objectives. Additionally, having an array of potential solutions in a Pareto-front is a better approach for design space exploration, compared to having a single solution suggested by the search algorithm. Design space exploration is a fundamental part of HW-SW co-design making NSGA-II an attractive alternative to SOGA.

Considering the accuracy-related fitness term ψ, the quantization strategies of a population $\mathcal{P}$ need to be evaluated in a reasonable amount of time to avoid a bottleneck in the search process. To address this problem, quantized networks in $\mathcal{P}$ are not fully trained during the search. Instead, the CNN model is instantiated and loaded with pre-trained floating-point weights, then quantized according to the genome of ρ and briefly fine-tuned to recover from the accuracy loss introduced by the direct quantization process. This process can also be parallelized, as the individuals within a population can be fine-tuned at the same time, on a single or multiple GPUs. The learning behavior of 2-bit, 4-bit and 6-bit networks was analyzed, to see how early the training curves can be differentiated. This gives an indication of how well the accuracy will be at the end of a full-training cycle. The accuracy fitness evaluation epochs in section 6.1.6 were chosen accordingly. The GA essentially evaluates the learning capacity of the individual, not its final fully-trained accuracy. At the end of the search, when a solution is chosen, it is trained from scratch, without loading any pre-trained weights. It is worth mentioning that fast accuracy predictors, such as the ones proposed in [84, 170], could also be used for the purpose of fast accuracy-related fitness evaluation in HW-FlowQ's GA.

6.1.3.4 Genetic Operators

The mutation, crossover and selection operations are pivotal to the GA's efficacy. Single-point crossover is applied, which intuitively has a high probability of capturing attractive bitwidth-to-layer encodings of two fit individuals and maintains inter-layer dependencies across segments of the CNN, as shown in figure 6.4. With mutation probability p_{mut} a single allele at a randomly selected genetic locus is replaced by another from the set of possible alleles, Q. All individuals conform with the CNN and the quantization levels supported on the hardware. Tournament selection is used for SOGA, where tournaments take place to decide all the survivors. NSGA-II selection is based on the crowded-comparison-operator [171].

6.1.3.5 Modeling Mixed-Precision Inference

This work focuses on modeling spatial architectures similar to [172, 4, 100, 3], with an on-chip buffer and a compute core with an array of PEs, as depicted in figure 6.1. The energy cost of data accesses depends on the technology and the size of the memory. HW-FlowQ supports independent read-write costs for off-chip communication, memory blocks, as well as the register files (RFs) of the PEs on the compute blocks. The cost models are inspired by the approach proposed in [3, 173, 4, 100]. A normalized energy cost is set for each operation that can take place on the architecture. The HW-model attempts to map the computations of a particular CNN workload efficiently onto the HW-model. For each schedule, the HW-model is able to extract the number of *actions* (reads, writes, MACs) required at each level of the accelerator as explained in section 3.3.2. The number of actions is multiplied by the cost of each action on each type of memory/compute unit. The exact normalized energy costs chosen in this work align with the Timeloop [4] framework and the Eyeriss model in [3], as shown in table 6.1. HW-FlowQ also supports manually setting costs for each action, based on different fabrication technologies.

Scheduler and Mapper. Modern compute architectures allocate a considerable amount of their power budget for memory accesses and data movement [173]. Moreover, redundant data movement can have a significant impact on latency. This has made efficient scheduling of CNNs on spatial hardware an active field of research [3, 109, 108, 172, 100, 4].

The three main techniques commonly used to optimize a nested loop's execution on hardware were introduced in section 2.3.2, namely *loop tiling, reordering and unrolling*. HW-FlowQ's scheduler and mapper components handle loop optimization techniques largely in a similar manner to the popular frameworks Interstellar [100] and Timeloop [4]. Here, the additional considerations to capture the effect of mixed-precision quantization are discussed.

Quantization shrinks the bitwidth of datatypes allowing larger computation tiles to fit in a given lower-level memory. This increases the number of possible loop tiling and reordering schedules. Loop optimization through *unrolling* is handled by HW-FlowQ's mapper component and is dependent on both the dataflow supported by the accelerator and the mixed-precision computation technique. It is important to note that when unrolling fractionalized (quantized) computations on a vectorized or multi-lane bit-serial PE-array, a single PE may handle more spatially distributed computations, as long as its register files fit the operands/partial sums needed/generated by said computations. This can be exploited by HW-FlowQ's mapper to find more efficient schedules which fit on a smaller physical computation array and require less PE-to-PE data movement.

Depending on the defined HW-model, the scalar or SIMD vector-engines in the PE can be word-aligned, making some quantization degrees less attractive than others. An example of sub-optimal SIMD-register usage is marked with a red-cross in figure 6.1 (middle-right). HW-FlowQ can also model bit-serial compute units such as the ones in [66], in which case, a relative improvement for any quantization level for weights and/or activations can be achieved on the compute block. The word alignment on the compute block can be set differently to that of the outer memory blocks and the off-chip memory interconnect.

In the convolution operation, PSUMs can grow after each accumulation to a maximum of $2b_{W/A}+C_i$, where $b_{W/A}$ is the bitwidth of the operands. The HW-model considers instances of the largest possible PSUM, according to the maximum bitwidth b_{max} supported by the accelerator. The increase in vector throughput due to quantization of $\mathbf{W}^l$ and $\mathbf{A}^{l-1}$ is constrained to the maximum amount of PSUM RF memory available on the PE. After complete accumulation, a speed-up can be achieved in writing back the output pixels at the quantization level of the input of the following layer b^l_A of $\mathbf{A}^l$.

Vectorized and Bit-Serial Computation. To estimate the benefits in the computation of low-bitwidth and mixed-precision CNNs, vectorized and bit-serial compute units are modeled [8, 174, 66]. The choice of the computation unit has a direct influence on the schedule, as it affects how many computation cycles are required for a particular operation and how many unique computations can be assigned to the same hardware at different bitwidths.

For vectorized accelerators, an aligned SIMD-MAC unit is modeled, which has a maximum bitwidth of b_{max} for both weights and activations. A speed-up through data-level parallelism at the PE-level can happen at V_{speedup} integer steps, as shown in equation 6.2.

$$V_{\text{speedup}} = \left\lfloor \frac{b_{max}}{\texttt{max}(b_W, b_A)} \right\rfloor \tag{6.2}$$

V_{speedup} is the vectorization degree aligned with the wider operand between b_W and b_A. This not only allows for more parallel computations in the same cycle, but also reduces the memory access cost at the register file level, which would now access V_{speedup} data that fit into the SIMD-register with bitwidth b_{max} in a single read operation. The limitation of vectorized computation units is that they can only perform complete operations, and therefore cannot always fully exploit any arbitrary bitwidth. For example, a 16-bit vector unit can perform 2 complete 8-bit computations, however, if the operands were 6-bits each, a non-integer speed-up of $\sim$2.67 would not be possible. Another limitation is that variable bitwidths of $b_W \neq b_A$ cannot be exploited for higher parallelism even if both are aligned, due to the $\texttt{max}(b_A, b_W)$ term in equation 6.2. The wider of the two operands dictates the number of parallel computations which fit in the vector engine.

Bit-serial computation units can fully exploit any level of quantization for both operands. Their performance is enhanced with respect to a b_{max} computation according to

$$BS_{\text{speedup}} = \frac{b^2_{max}}{b_W \times b_A}. \tag{6.3}$$

It is important to note that BS_{speedup} cannot be directly compared to V_{speedup} due to the inherent differences between how both architectures perform a single b_{max} computation. Bit-serial units require $b_W \times b_A$ cycles to complete a single computation, whereas bit-parallel (irrespective of the

vectorization degree) produce a complete result(s) in every computation cycle. To compensate for this, most bit-serial accelerators employ more computation *lanes* to make up for their slower method of computation, and try to match the *throughput* of bit-parallel architectures, while benefiting from the flexibility of supporting and getting a speedup at any reduced bitwidth for either operand b_W or b_A. We model each PE to have a fixed number of independent lanes to perform parallel computations. The bits trickle in from the register files over $b_W \times b_A$ cycles.

6.1.4 Mixed-Precision Model Validation

To validate HW-FlowQ's *Fine*-level modeling and mapping components, its estimates are compared to the Eyeriss architecture [3] for AlexNet [52] inference. AlexNet is not used for quantization experiments (section 6.1.6), as it is severely outdated. Its diverse CNN layer shapes (large/small kernels, strides, group convolutions) are used here to prove the fidelity/precision of HW-FlowQ's hardware modeling framework on varying computation workloads. Similarly, Eyeriss provides sufficiently complex on-chip data movement, which is also challenging to model (e.g. vertical, horizontal, and diagonal data movement on spatial array). This diversity in workloads and data movement helps validate the HW-modeling framework. The model is further validated with Timeloop [4] across workloads from the DeepBench benchmark suite [5], on Eyeriss-256-DB (configuration in table 6.1).

Figure 6.5 (top-left) shows the breakdown of normalized energy contributions at each memory level for the AlexNet convolutional layers. HW-FlowQ's 16-bit HW-model demonstrates high fidelity and accuracy with respect to [3] and [4]. The effects of quantization on latency and energy, for vectorized and bit-serial HW-models is shown as well. Extending the comparison with Timeloop, the DeepBench workloads demonstrate the consistency between the modeling techniques (figure 6.5 (top-right)). Deviating points occur when both frameworks resolve to different mapping solutions (i.e. not related to the correctness of the model, but rather the schedule search). For both bit-serial and vectorized accelerators, a non-trivial variation can be observed for the energy benefits of different quantization schemes. Quantization allows for larger computation tiles to fit on the on-chip memory, making more loop-tilings legal for a given workload. This changes the schedule search space and introduces new solutions for the model's mapper to consider. A slightly more direct relationship can be observed for latency, particularly for bit-serial accelerators, which have a high parallelism potential (due to computation lanes) that is not easily saturated. It is important to note that the y-axis follows a logarithmic scale for bit-serial latency, due to the quadratic speed-up gains (equation 6.3). Timeloop [4] does not support bit-serial PEs and as such is not compared to HW-FlowQ in the corresponding plots.

6.1.5 Cross-Abstraction Level Interactions

HW-FlowQ groups hardware-related design parameters into abstraction levels to facilitate the interpretation of HW-CNN interactions (recall figure 6.3). The neighboring abstraction levels must be cohesive to create a sensible flow between them, which can guide the designer and the genetic algorithm towards a more efficient co-design strategy. For example, CNN workloads (*Coarse*) directly affect the on-chip memory (*Mid*). However, the effect of a CNN workload (*Coarse*) on the dataflow (*Fine*) is hard to understand without knowing the on-chip/off-chip

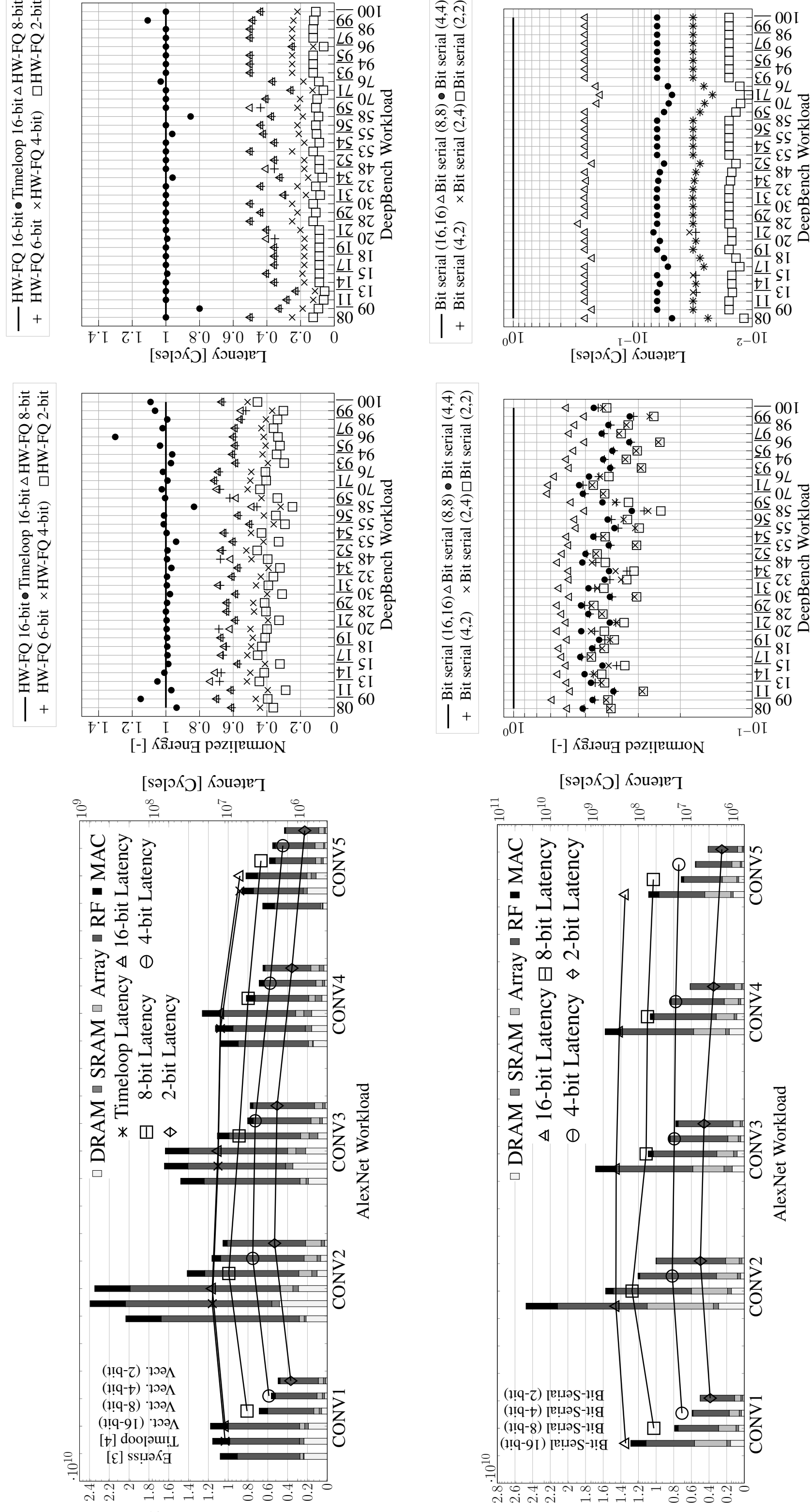


Figure 6.5: 16-bit AlexNet validation of HW-FlowQ with Eyeriss [3] and Timeloop [4] (top-left), as well as 8, 4 and 2-bit vectorized execution. Validation with Timeloop on DeepBench workloads [5] (top-right). Bit-serial execution of AlexNet (bottom-left) and DeepBench (bottom-right).

interconnect (*Mid*) or the on-chip memory size (*Mid*) in between. The criteria are separated to divide the complexity of CNN structure search (*Coarse*), interconnect/memory hierarchy search (*Mid*), and compute architecture search (*Fine*).

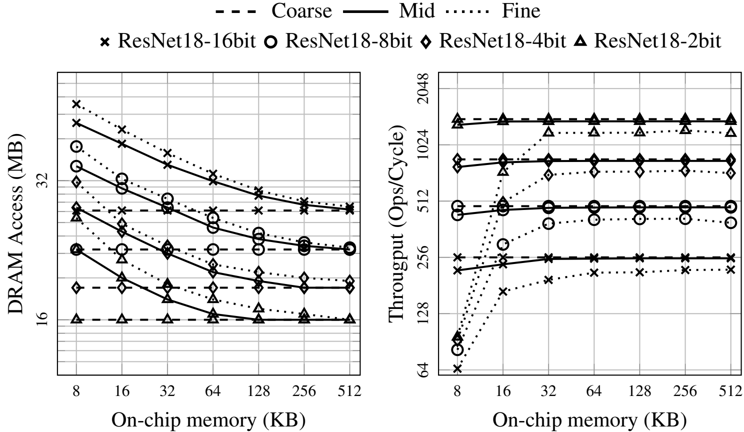


Figure 6.6: Analysis of DRAM access and throughput on on-chip buffer size at different levels of hardware abstraction and quantization.

In figure 6.6, the effect of changing the numerical precision of ResNet18 for ImageNet on off-chip (DRAM) accesses and computational throughput is investigated. These metrics can be evaluated at all three abstraction levels, which makes them useful in highlighting the *flow* between the levels. At the *Coarse*-level, the DRAM accesses are estimated as the CNN's total necessary reads and writes for all datatypes (inputs, weights and outputs). Since the *Coarse*-level abstraction is agnostic to the on-chip memory details, all its corresponding dashed curves are constant. However, among the *Coarse*-level curves, the difference in read/write volumes at each quantization level (left plot), as well as the speed-up possible through vectorization (right plot) is still captured.

Moving on to the *Mid*-level, the model can capture more details of the hardware. In this case, the limitations of an under-dimensioned on-chip memory or an insufficient off-chip to on-chip communication bandwidth can be detected. The *Mid*-level estimates are sensitive to on-chip memory and communication, but semi-agnostic to the compute architecture. For this example, the bandwidth of off-chip to on-chip communication is set to 8 bytes/cycle. On the DRAM accesses plot, the solid lines approach the dashed (ideal) curves, as the on-chip memory grows. More importantly, at lower numerical precisions, the *Mid*-level estimates meet the corresponding *Coarse*-level estimates at smaller on-chip memory sizes. Moreover, the *Mid*-level's limited information on the computation architecture can still be used to detect bottlenecks in communication and/or on-chip memory size. The *Mid*-level abstracts the details of the compute architecture through the assumption that all PEs are fully utilized and can always perform computations, if sufficient data is available. In the throughput plot, communication bottlenecks for small on-chip memory sizes can be observed, which are not able to provide the ideal computation architecture with enough data to fully utilize it. These communication bottlenecks are more evident for CNN models consisting of multiple fully-connected layers

(AlexNet and VGG-16). This behavior does not change with numerical precision, since smaller bitwidths also increase the ideal computation throughput of vectorized PEs (i.e. *Coarse*-level estimates get better).

At the *Fine* abstraction level, the model considers the CNN, the memory hierarchy, and the compute architecture details (register files, dataflow, mapping, etc.). For this example, the validated Eyeriss-256-DB from table 6.1 is used, with varying SRAM sizes. The *Fine*-level dotted curves approach the *Mid* and *Coarse* curves at a slower rate, as the on-chip memory increases. This is due to the other limitations of the computation architecture, which the *Fine*-level takes into account (e.g. sub-optimal unrolling, limited register file sizes, etc.). The *Fine*-level provides much more information (as shown in figure 6.2), but for the purpose of highlighting the cross-abstraction level interactions, these are not discussed in this section.

The different bitwidth ResNet18s have different *Coarse* lines (dashed), which limit the theoretical optimum DRAM accesses and throughput. The *Mid* and *Fine* lines (solid and dotted), which capture more hardware details, never surpass their respective dashed lines which are defined at the *Coarse* stage. The search at the *Coarse* level provides these theoretical optimal performance levels for a range of mixed-precision CNN quantization strategies, while subsequent levels try to reach that optimum, by parameterizing the hardware. For example, if the target was to achieve the theoretical best performance with respect to 4-bit *Coarse*, either the hardware can be over-dimensioned, (dotted-diamond line at 512KB of on-chip memory) or the CNN can be quantized down to 2-bit with an on-chip memory of 32KB (*Mid* and *Fine* triangle lines of 2-bit touch/surpass the 4-bit *Coarse* line). Both options allow us to reach the theoretical optimum set by *Coarse* for 4-bit, but each option would have a different effect on accuracy, where the over-dimensioned hardware would achieve higher accuracy due to larger bitwidths, while the 2-bit CNN would have lower accuracy but a smaller on-chip memory design.

From figure 6.6, a multi-abstraction *flow* can be deduced, which can help the designer eliminate hardware and CNN candidates at early stages of the co-design, without having to spend costly GPU hours on training or synthesis and HIL-based testing.

6.1.6 Evaluation

HW-FlowQ is evaluated based on CIFAR-10 [27] and ImageNet [26] datasets for the classification task and Cityscapes [147] for the semantic segmentation task. The 50K train images of CIFAR-10 are used for training and accuracy fitness ψ evaluation, while the 10K test images are used for final accuracy evaluation at the end of the search. The images have a resolution of 32×32 pixels. ImageNet consists of ∼1.28M train and 50K validation images with a resolution of 256×256 pixels. The Cityscapes dataset consists of 2975 training images and 500 test images. The images of size 2048×1024 show German street scenes along with their pixel-level semantic labels of 19 classes.

Section 6.1.6.1 and section 6.1.6.3 highlight the flexibility of the HW-modeling tool and search approach. To isolate and identify the effects of changing the HW-model on the resultant quantization strategy, all other variables of the experiment are fixed, including the CNN workload (ResNet20). In section 6.1.6.2, the hyper-parameters of NSGA-II and its convergence are studied. Here, the task is made more complex by enlarging the quantization search space, and employing a deeper 56-layer CNN. In section 6.1.6.4, HW-FlowQ is applied to a different task domain, namely

semantic segmentation. The DeepLabv3 [175] model is used to study the effects of layer-wise quantization on the encoder, bottleneck layers (including the atrous spatial pyramid pooling (ASPP) block), and the decoder layers of the segmentation network. Finally, in section 6.1.6.5, HW-FlowQ is compared with state-of-the-art methods of uniform and variable quantization, further testing on wide and high resolution CNNs (ResNet18 for ImageNet). If not otherwise mentioned, all hyper-parameters specifying the task-related training were adopted from the CNN's base model and its corresponding quantization method. The first and last layers are kept at 16-bits, following the heuristic of other quantization works [35, 34, 36]. The hardware metrics are generated based on the hardware configurations described in table 6.1. The vectorized Spatial-256 HW-model with row-stationary dataflow is used in section 6.1.6.5 with additional support for 1-bit (XNOR-Net). As a *Coarse*-level metric, fractional operations (Frac. OPs) is used as a measure of CNN computation compression, with respect to the hardware it is executed on. For example, Frac. OPs of a vectorized accelerator are computed as the layer-wise sum of operations over the speed-up due to the respective layer's quantization:

$$Frac.OPs = \sum_{l=0}^{L} \frac{OPs_l}{V^l_{speedup}}. \tag{6.4}$$

Table 6.1: Hardware configurations and normalized access energy costs used for experiments and validation.

HW-Model \ Specs	PE	DRAM	SRAM		Array	Registers	
	Array	Cost	Size	Cost	Cost	Size (filter, ifmap, psums)	Cost
Spatial-168*	12×14	200	128KB	6	2	224, 12, 16 Words	1
Spatial-256*	16×16	200	256KB	13.84	2	224, 12, 16 Words	1
Spatial-1024*	32×32	200	3072KB	155.35	2	224, 12, 16 Words	1
Eyeriss-1024	32×32	200	3072KB	155.35	2	224, 37, 16 Words	1
Eyeriss-256 - DB	16×16	200	128KB	7.41	0	192, 12, 16 Words	0.99

*: Same dimensioning for bit-serial (BS) and other dataflows (RS, OS, WS)

For experiments on CIFAR-10, the population size $|\mathcal{P}|$ is set to 25 and 50 for exploration and comparison with state-of-the-art experiments respectively. The number of generations is fixed to 50 for all CIFAR-10 experiments. Probabilities for mutation and crossover are set to 0.4 and 1.0 respectively. For ImageNet experiments, $|\mathcal{P}|$ and the number of generations are scaled down to 10, while Cityscapes experiments have $|\mathcal{P}|$=25 and for 10 generations. The CNNs trained on CIFAR-10 are fine-tuned for 2 epochs and their accuracy fitness is evaluated on 10K random samples from the train-set during the search. For ImageNet, 0.4 epochs of fine-tuning are performed before evaluating on the valid-set. For Cityscapes, 10 epochs are necessary to evaluate the candidate population. As explained in section 6.1.3.2, the accuracy fitness (ψ) is the GA's measure of the learning capacity of an individual. To avoid artificially biasing the search algorithm towards individuals that perform well on the test set, the accuracy fitness is restricted to train or validation set. This way, the framework does not indirectly "see" the test set during the

search. After the search concludes, we fully train the chosen individual from scratch and report its test set accuracy as "Accuracy Top-1" in the result tables.

6.1.6.1 A HW-CNN Co-Design Example

This experiment serves as a simple example of how the design levels can be used to iteratively narrow down the range of solutions that could suit a potential application. Additionally, SOGA and MOGA (NSGA-II) variants of the search algorithm, presented in section 6.1.3.2, are compared. In a real use-case, a set of different CNNs can be considered at the start of the exploration, for example, proposed by a NAS algorithm. For simplicity, ResNet20 is chosen as the baseline, with task-accuracy of 92.47% on the CIFAR-10 dataset. At the *Coarse*-level, compression starts with Frac. OPs being the target optimization criterion. Considering a vectorized accelerator, the GA is allowed to maintain individuals that have bitwidths which are supported by the target HW-accelerator. In this example, the bitwidths are restricted to $b_A = b_W \in \{16,8,6,4,2\}$. In table 6.2, the reduction in Frac. OPs with respect to the bitwidth restrictions given is around 75%, at a task-related accuracy of around 89-90%. Assuming a higher accuracy/fractionalization or lower CNN memory footprint was necessary, the condition $b_A = b_W$ can be relaxed, allowing the GA to find solutions with varying b_A and b_W values. Additionally, more quantization levels can be supported and expand the range $\{16,8,6,4,2\}$ with intermediate quantization levels and/or binary OPs. This would result in a more fine-grained search that results in new solutions that achieve our desired task-accuracy and CNN memory footprint, at the cost of potentially more complex hardware (supporting more b_W and b_A options for example). It is important to note that NSGA-II offers a *range* of Pareto-optimal Frac. OPs vs. accuracy solutions (*Accuracy Pareto-leader*: Top-1 accuracy of 90.70% at 63% φ_{OPs} reduction; *Compression Pareto-leader*: Top-1 accuracy of 89.34% at 77.09% φ_{OPs} reduction).

Table 6.2: ResNet20 for CIFAR-10 quantized at different abstraction levels of the Spatial-256 hardware with SOGA and NSGA-II.

Configuration (< φ >;< `level` >;< $\mathcal{G}$ >)	**Accuracy** Top-1 [%]	**Accuracy** ψ Fitness [%]	**HW-**φ Fitness [%]*	**N. Energy** [$\times10^7$]	**Latency** [$\times10^3$cyc.]
Baseline (16 bit)	92.47	-	-	32.84	191
Frac. OPs; Coarse; SOGA	89.28	88.44	79.64	-	-
Frac. OPs; Coarse; NSGA-II	**90.09**	**92.80**	73.09	-	-
DRAM acc.; Mid; SOGA	89.18	91.93	67.79	-	-
DRAM acc.; Mid; NSGA-II	**90.00**	**95.33**	65.56	-	-
N. Energy; Fine; SOGA	89.45	91.18	51.05	16.07	52
N. Energy; Fine; NSGA-II	**90.09**	**94.75**	48.12	17.04	61
Latency; Fine; SOGA	88.44	86.21	78.71	14.94	41
Latency; Fine; NSGA-II	**89.99**	**94.78**	68.77	17.05	59

*: Measured as (1-(Compressed Metric/Baseline Metric))*100

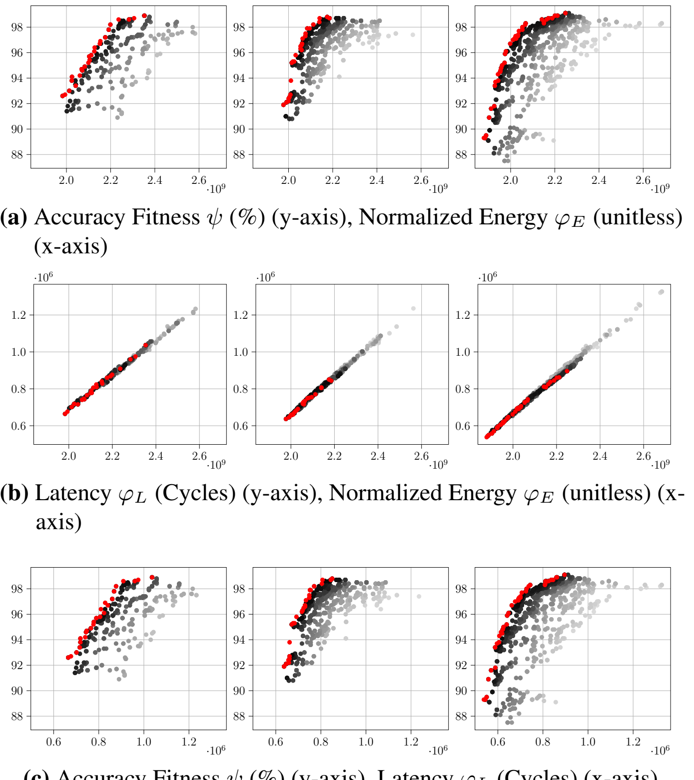
Once satisfied with the CNN's compression potential, the search can be refined to take the off-chip to on-chip memory movement into consideration. The number of processing passes (rounds of communication between off-chip to on-chip) necessary to complete all the compu-

tations of a CNN can be estimated. The layer tiling and loop ordering can be searched for different quantization strategies. For this example, the *Mid*-level estimates are based on DRAM accesses when the on-chip buffer is dimensioned to 256KB. The CNN's DRAM accesses can be reduced by around 65% with respect to the 16-bit baseline CNN, while maintaining the same accuracy that was targeted at the *Coarse*-level. Based on the bandwidth of the off-chip to on-chip communication infrastructure considered, this can confirm that our dimensioning of the on-chip buffer is in a good range to achieve a significant reduction of DRAM accesses, without having to *over-quantize* our CNN and lose the task-related accuracy goal. Finally, the *Fine*-level estimates give us a better understanding of how our CNN can be scheduled on a particular HW-model. For this example, we proceed with the Spatial-256 configuration presented in table 6.1, with row-stationary dataflow. Normalized energy can be reduced by around 50%, while maintaining the target Top-1 accuracy from the higher abstraction levels. When considering latency, we observe the drawback of the SOGA approach, not being able to decently balance accuracy and the hardware-reward. Although the emerging solution maximizes the latency reward significantly (78.71%), it leads to a considerable accuracy degradation (88.44%). The reward function (equation 6.1) was designed to balance both accuracy and hardware-rewards, however, due to the high potential of improving latency through quantization, we find that the SOGA algorithm was willing to sacrifice the train reward ψ (down to 86.21%) to get a much larger overall reward through latency φ_L. This highlights the weakness of handcrafting reward functions to achieve multi-criteria optimization. On the other hand, NSGA-II offers a range of solutions, from which a well-balanced solution is shown in table 6.2, reducing latency by (68.77%) and maintaining the task-related accuracy since the *Coarse*-level. The Pareto-optimal solutions for latency vs. accuracy range from Top-1: 90.63% at 60.00% φ_L reduction, to Top-1: 89.37% at 72.74% φ_L reduction. The set of all Pareto-front solutions is not shown in the table.

6.1.6.2 Multi-Objective Genetic Optimization using NSGA-II

To relieve the burden of designing a fair cost function, facilitate design space exploration, and maintain a diverse set of Pareto-optimal solutions, NSGA-II is leveraged as the search technique for the next experiments. The multi-objective capabilities are exploited to simultaneously construct a Pareto-front which optimizes for task-related accuracy, energy, and latency, concurrently in a single search experiment. To collect more information on the hyper-parameters of the GA, the characteristics of the search space, and the relationship between quantization and the optimization targets, the following experiments are performed on the Spatial-256 HW-model with row-stationary dataflow and the deeper ResNet56 CNN, trained on the CIFAR-10 dataset. As mentioned in section 6.1.3, the quantization search space has a size of Q^{2L}, where Q is the set of possible quantization levels and L the number of layers. The larger search space of ResNet56 (5^{54} solutions for $b_A = b_W \in \{16,8,6,4,2\}$) helps in verifying the scalability of the GA search approach.

Figure 6.7 shows 2-D projections of the 3-D Pareto-fronts produced by three experiments, each with a different population size $|\mathcal{P}|$ and/or number of generations. An increase in generation count (left vs. middle) allows the Pareto-front to take a more convex form. On the other hand, increasing the population size $|\mathcal{P}|$ results in an extended Pareto-front, finely covering a wider surface and a larger hypervolume, however with similar form as the middle configuration. The solutions which



(a) Accuracy Fitness ψ (%) (y-axis), Normalized Energy φ_E (unitless) (x-axis)

(b) Latency φ_L (Cycles) (y-axis), Normalized Energy φ_E (unitless) (x-axis)

(c) Accuracy Fitness ψ (%) (y-axis), Latency φ_L (Cycles) (x-axis)

Figure 6.7: 2-D projections of three 3-D Pareto-fronts for ResNet56 quantization: left to right ($|\mathcal{P}|$, generations) = (25, 25), (25, 50), (50, 50). Grey to black shades represent Pareto-fronts of older to newer generations, red points belong to the final Pareto-front. **It is recommended to view this figure in color.**

are most attractive, are those which offer a trade-off among the optimization criteria. For ($|\mathcal{P}|$, generations) = (25, 50) and (50, 50), the points which contribute the most to the total Pareto-front hypervolume (lie at the apex of the convex Pareto-front) are comparable in hardware metrics and accuracy fitness. The ($|\mathcal{P}|$, generations) = (25, 25) configuration has solutions of equivalent accuracy fitness, however, their hardware metrics are worse. With these insights, the number of generations is fixed to 50 for all CIFAR-10 experiments to get better convergence. $|\mathcal{P}|$ is set to 25 for exploration experiments and 50 for comparison with state-of-the-art experiments.

The hypervolume occupied by the 3-D frontier can be measured at each generation to derive the search convergence. As a decision-making technique, a reference point is extrapolated from the polar solutions (worst in each dimension) of the final Pareto-front, and the farthest solution from it in the frontier is found, based on Euclidean distance. In this work, this solution is referred to as the hypervolume-leader (HV-leader), which offers a balanced trade-off among the Pareto-points of the frontier.

6.1.6.3 HW Modeling and Exploration

Quantization on Different HW Dimensions. In this experiment, three candidate hardware accelerators are modeled to observe the effect of hardware dimensioning (computing units, buffer sizes and memory access costs) on quantizing ResNet20 for CIFAR-10. Figure 6.8-a shows three 2-D projections of four 3-D Pareto-fronts optimizing task-related train reward ψ, normalized energy φ_E and processing cycle latency φ_L. φ_E and φ_L are measured for processing 4 frames, to compare with a batch-processing HW-model.

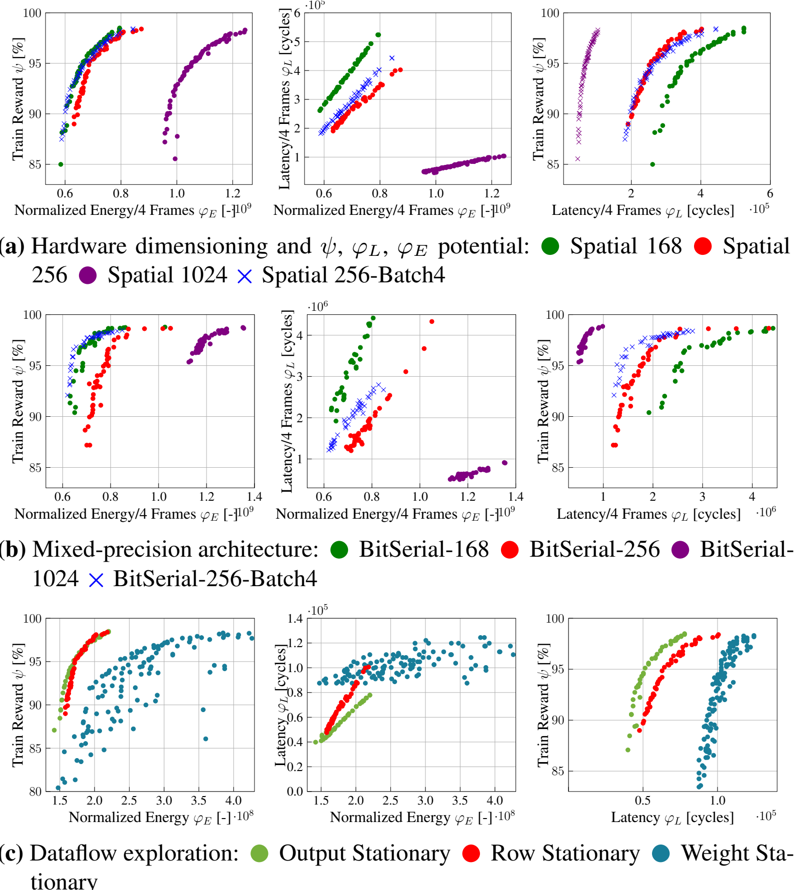
The three differently dimensioned spatial compute arrays (details in table 6.1) show similar characteristics in the shape and form of their Pareto-fronts. A slight difference can be observed for Spatial-1024, where its Pareto-front has a narrower latency range with respect to different quantization strategies. This indicates that the loop unrolling capacity is already exploited at a high degree due to the large PE-array, and cannot be improved much further through quantization. This hints to Spatial-1024 being slightly over-dimensioned for the task. On the other hand, Spatial-168 and 256 HW-models show a wider range of solutions for φ_E and φ_L, leaving more room for exploiting quantization to meet a given constraint for the CNN under consideration (ResNet20). In all three plots, a gap can be noticed between the 1024 model and the others, indicating a good potential for testing a model dimensioned in between (e.g. 512 PE array).

Increasing the batch size to 4 on the Spatial-256 configuration shows an improvement in terms of energy, bringing the Spatial-256 configuration closer to the energy efficiency of Spatial-168, while maintaining the latency of the single-batch Spatial-256 configuration. This can be seen in figure 6.8-a.

In table 6.3, the results of the baseline 16-bit ResNet20 executing on the 3 hardware configurations are shown, as well as a larger batch-size. Maintaining the desired accuracy threshold of 90%, solutions are chosen from the Pareto-front of each hardware configuration. With this accuracy requirement, the Spatial-1024 configuration can achieve a very low latency with respect to other configurations, without over-quantizing the CNN. This comes at the cost of more energy required by the execution, due to larger memories and more expensive access costs (recall costs in table 6.1). For the small Spatial-168 hardware configuration, the latency is the highest, as it needs to maintain an accuracy of 90% (i.e. cannot over-quantize), while processing on fewer PEs. Nevertheless, the smaller design reduces the energy requirements of the execution. Finally, taking a look at Spatial-256 with batch-size 4, an improvement in latency and energy can be observed due to better reuse of the weights with respect to the batched inputs, bringing the energy of the execution close to the small Spatial-168 accelerator.

Quantization on Bit-Serial Mixed-Precision Accelerators. So far, vectorized accelerators have been considered, which support $b_A = b_W \in \{16,8,6,4,2\}$. In this experiment, the underlying computation unit is modified to observe the benefits that can be achieved for a bit-serial accelerator, which supports any $b_A, b_W \leq 16$ for any layer. All PEs have 16 computation lanes to allow for higher throughput and partially compensate for the slower, serialized operations. The results of this experiment are shown in figure 6.8-b.

An interesting difference can be observed when changing the dimensioning of the accelerator. A larger bit-serial accelerator produces an even more compact Pareto-front, due to its ability to maximize loop unrolling over the large PE-array, extended further with the computation lanes. For the energy/latency graph (middle) more solutions appear for a particular energy and/or latency,



(a) Hardware dimensioning and ψ, φ_L, φ_E potential: ● Spatial 168 ● Spatial 256 ● Spatial 1024 × Spatial 256-Batch4

(b) Mixed-precision architecture: ● BitSerial-168 ● BitSerial-256 ● BitSerial-1024 × BitSerial-256-Batch4

(c) Dataflow exploration: ● Output Stationary ● Row Stationary ● Weight Stationary

Figure 6.8: 2-D projections of 3-D Pareto-fronts of 3 exploration experiments on ResNet20 for CIFAR-10 for hardware dimensioning, bit-serial processing and dataflow variants. **It is recommended to view this figure in color.**

breaking the almost linear relationship between optimal energy and latency mapping observed for vectorized accelerators (figure 6.8-a). This can be attributed to both the change in compute architecture and the variations possible for both b_A and b_W. In table 6.4, similar latency and energy trends can be observed for bit-serial computation, as with vectorized computation for batch-size 1. A Pareto-optimal solution is chosen for each hardware and trained to achieve an accuracy above 90%. The smallest BS-168 is the slowest, yet the most energy efficient, while BS-1024 significantly reduces the latency at the cost of more energy for data movement. The improved effect of batch processing is more prominent for bit-serial accelerators. The 256-PE bit-serial accelerator with batch processing offers a significant improvement in terms of energy, bringing the 256-PE configuration to better energy efficiency than the smaller 168-PE counterpart executing batch-size 1 inputs. Additionally, latency also gets a decent improvement of 13.5%. This shows the advantages of relaxing the $b_A = b_W$ constraint on the hardware and the GA search.

Table 6.3: Quantization of ResNet20 for CIFAR-10 on different hardware dimensions.

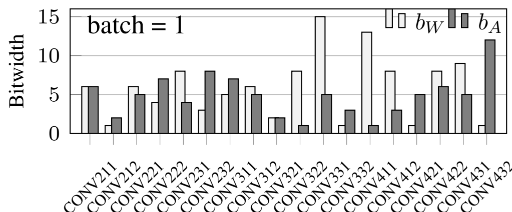
Configuration (< `choice` >;< μ >;< `batch` >)	**Accuracy** Top-1 [%]	**Accuracy** ψ Fitness [%]	**N. Energy** φ_E / 4 Frames [$\times 10^7$]	**Latency** φ_L / 4 Frames [$\times 10^3$cyc.]
Baseline (16-bit); Spatial-168; 1	92.47	-	130.28	1128
Baseline (16-bit); Spatial-256; 1	92.47	-	131.36	764
Baseline (16-bit); Spatial-1024; 1	92.47	-	190.40	204
Baseline (16-bit); Spatial-256; 4	92.47	-	119.94	764
Pareto-Choice; Spatial-168; 1	90.31	94.22	64.80	343
Pareto-Choice; Spatial-256; 1	90.26	96.31	72.68	282
Pareto-Choice; Spatial-1024; 1	90.25	95.08	105.41	68
Pareto-Choice; Spatial-256; 4	90.03	95.19	66.09	259

Table 6.4: Quantization of ResNet20 for CIFAR-10 on bit-serial accelerators.

Configuration (< `choice` >;< μ >;< `batch` >)	**Accuracy** Top-1[%]	**Accuracy** ψ Fitness [%]	**N. Energy** φ_E / 4 Frames [$\times 10^7$]	**Latency** φ_L / 4 Frames [$\times 10^3$cyc.]
Baseline (16-bit); BS-168; 1	92.47	-	141.07	20365
Baseline (16-bit); BS-256; 1	92.47	-	147.65	12296
Baseline (16-bit); BS-1024; 1	92.47	-	208.26	3326
Baseline (16-bit); BS-256; 4	92.47	-	137.57	12296
Pareto-Choice; BS-168; 1	90.17	94.90	68.36	2468
Pareto-Choice; BS-256; 1	90.37	92.91	75.58	1406
Pareto-Choice; BS-1024; 1	90.19	96.95	116.63	563
Pareto-Choice; BS-256; 4	90.33	92.10	62.01	1216

To further analyze this aspect, the layer-wise quantization strategy chosen by the GA for batch sizes 1 and 4 on bit-serial accelerators are presented in figure 6.9. Layers with large activation volumes can have lower bitwidth activations (low b_A), while the weights can be kept at a slightly higher bitwidth (higher b_W). The opposite can be done for layers with large filter volumes. This extends the improvements to be gained on mixed-precision accelerators and larger batch sizes (i.e. larger activation volumes). In figure 6.9, the quantization strategy chosen for batch size of 4 reflects the GA's attempt to compress the large activations more aggressively than for batch size of 1, particularly for the first half of the CNN. To compensate for the potential accuracy loss, the GA maintains larger weight bitwidths b_W for batch = 4. The resulting quantized CNNs of both batch 1 and batch 4 have an equivalent accuracy ($\sim$90.3%), but with a noticeable improvement in hardware metrics for batch size of 4, due to the GA taking the capabilities of the hardware into account.

Quantization on Different Dataflows. To demonstrate the effect of quantization on dataflows, a WS dataflow and an OS dataflow are presented. WS unrolls computations in dimensions C_o and C_i over the processing element array, while OS unrolls X_o and Y_o, and replicates the unrolling over C_o. Both WS and OS support channel interleaving in order to maximize their register utilization, similar to the RS dataflow.



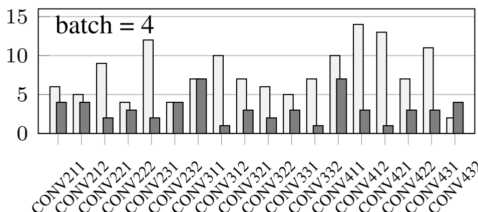


Figure 6.9: Layer-wise bitwidth strategy for BS-256 hardware. Batch size 1 (left) and 4 (right). NSGA-II compensates for larger activations (batch=4) by lowering b_A and maintains accuracy by increasing b_W, when compared to batch=1 inference.

Table 6.5: Quantization of ResNet20 for CIFAR-10 on different dataflows.

Configuration ($<$ `choice` $>$;$<$ μ $>$;$<$ `batch` $>$)	**Accuracy** Top-1[%]	**Accuracy** ψ Fitness [%]	**N. Energy** φ_E [$\times 10^7$]	**Latency** φ_L [$\times 10^3$cyc.]
Baseline (16-bit); Fine; OS-256;1	92.47	-	46.33	159
Baseline (16-bit); Fine; WS-256;1	92.47	-	69.40	166
Baseline (16-bit); Fine; RS-256;1	92.47	-	32.84	191
HV-Leader; Fine; OS-256;1	89.99	93.98	16.36	48
HV-Leader; Fine; WS-256;1	89.11	90.87	20.54	97
HV-Leader; Fine; RS-256;1	89.65	95.19	17.20	64

The baselines in table 6.5 show RS is the most energy-efficient, while OS offers the best latency. WS is placed in the middle in terms of latency but has worse energy efficiency when compared to the other considered dataflows. The Pareto-fronts of quantization strategies in figure 6.8-c demonstrate the effect of dataflows on three accelerators, which are otherwise identical in dimensioning. WS proves to be highly sensitive to quantization, having many unique non-dominated combinations of φ_E, φ_L and ψ. Generally, WS is the least efficient in terms of latency and energy, for a particular train accuracy ψ. OS dataflow enjoys its lead in latency, due to a higher potential of unrolling as a result of quantization over vectorized PEs (each vectorized PE acts as V_{speedup} virtual PEs). Consequently, its energy rivals that of RS. The higher parallelism degree on a single SIMD-vector engine reduces the total cost of MAC operations. Furthermore, since the loop unrolling is taking place across the array as well as within the vectorized PEs, fewer PE-to-PE hops are required to achieve the unrolling of the mapper, resulting in less array data movement energy.

6.1.6.4 Mixed-Precision Quantization for Semantic Segmentation

The semantic segmentation task is critical to applications in robotics and autonomous driving. High-quality segmentation can be more computationally complex by several orders of magnitude, when compared to classification tasks (e.g. table 6.6). This is related to both, the typically larger

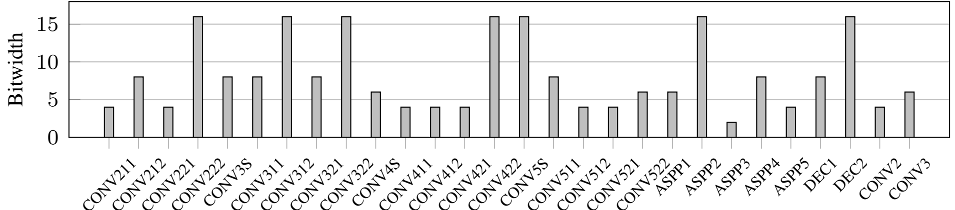


Figure 6.10: Layer-wise bitwidths (b_W=b_A) of a DeepLabv3 Pareto-choice strategy with 67.3% mIoU on Cityscapes. Short and parallel layers have b_A equal to their respective bottom layer.

input image resolution and the additional layers needed for semantic segmentation (bottleneck, ASPP block and decoder layers).

For the DeepLabv3 network executing on Eyeriss-1024 (details in table 6.1), HW-FlowQ must adapt to the task's training challenges, particularly on low-bitwidth ($\leq$4-bit) configurations for PACT quantization, which often lead to exploding gradients. Despite this difficultly of PACT, HW-FlowQ produced the Pareto-choice candidate shown in figure 6.10, which achieved 67.3% mean intersection over union (mIoU) with a 21.6% reduction in fractional operations over uniform 8-bit PACT quantization of DeepLabv3 (shown in table 6.6). In figure 6.11, qualitative semantic segmentation results are shown for uniform 8-bit PACT and the HW-FlowQ Pareto-choice, for three example scenes in the Cityscapes dataset. These results show an impressive potential of mixed-precision low-bitwidth quantization on complex semantic segmentation tasks, potentially with more advanced quantization techniques under HW-FlowQ in future work.

6.1.6.5 Comparison with State-of-the-Art

In this section, HW-FlowQ is compared against state-of-the-art quantization approaches on shallow, deep, and wide CNNs for classification and semantic segmentation tasks. DoReFa-Net [35] and PACT [36] indicate uniform quantization with the respective method. The PACT method for weights and activations propagates the gradients better during training for deeper and wider networks; therefore, ResNet56, ResNet18, and DeepLabv3 were compared against it. The work in HAQ [7] was reimplemented and the reward in equation 6.1 was adapted to the RL-agent. The RL-agent was integrated with the HW-model μ and thoroughly tested with different agent hyper-parameters to achieve the convergence behavior depicted in [7]. The agent achieved better results when optimizing for latency, as such, the respective results are quoted. In all cases, HW-FlowQ and HAQ use the same quantization method as the works they are compared against in the table. Finally, XNOR-Net is presented as a BNN variant to compare with a highly efficient implementation.

For ResNet20, the HV-leader provides energy and latency reductions of 48% and 69% while maintaining a Top-1 accuracy of 90.15%. In contrast, HAQ achieved a reduction of 42% and 57% for energy and latency. For ResNet56, our HV-leader achieved equivalent energy and latency to PACT (4-bit), while maintaining better accuracy. The HV-Leader achieved an improvement of 20% and 45% over HAQ for energy and latency, at an equivalent Top-1 accuracy of 92%. The improvements over HAQ go beyond these results. The RL-based approach in HAQ has the same limitations as SOGA, namely the aggregation of multiple criteria into one cost function. Our

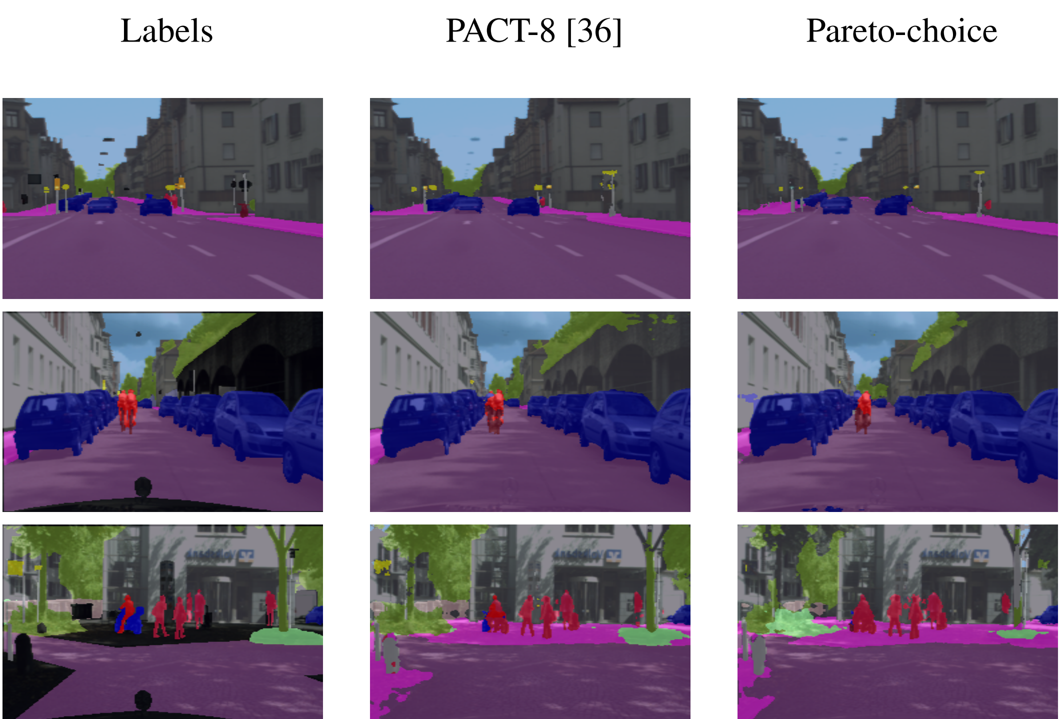


Figure 6.11: Qualitative results of DeepLabv3 quantization on Cityscapes scenarios. Black regions have no ground-truth labels. Pareto-choice has 21.6% Frac. OPs compression compared to uniform 8-bit PACT. **It is recommended to view this figure in color.**

Table 6.6: Comparison of HW-FlowQ with state-of-the-art quantization methods on Eyeriss-256 Vectorized.

Model/ Dataset	Method	Accuracy Top-1 [%]	F.Ops [$\times 10^6$]	N. Energy [$\times 10^7$]	Latency [$\times 10^3$cycles]
ResNet20 CIFAR-10	Baseline (16-bit)	92.47	41	33	191
	DoReFa-Net (4-bit) [35]	89.75	10	16	51
	DoReFa-Net (2-bit) [35]	87.16	5	14	43
	XNOR-Net (1-bit) [34]	83.98	3	15	17
	HAQ [7]	89.75	17	19	83
	HV-Leader [This Work]	**90.15**	**12**	**17**	**60**
ResNet56 CIFAR-10	Baseline (16-bit)	93.89	125	101	588
	PACT (4-bit) [36]	90.96	32	48	155
	PACT (2-bit) [36]	90.43	16	42	80
	XNOR-Net (1-bit) [34]	85.61	8	47	48
	HAQ [7]	92.07	56	61	279
	HV-Leader [This Work]	**92.00**	**30**	**49**	**154**
ResNet18 ImageNet	Baseline (16-bit)	69.01	1814	1489	9498
	PACT (4-bit) [36]	66.59	542	822	3070
	PACT (2-bit) [36]	63.59	330	703	1897
	XNOR-Net (1-bit) [34]	52.51	224	676	1230
	HV-Leader [This Work]	**67.02**	551	821	3191
DeepLabv3 CityScapes	Baseline (16-bit)*	69.68	147367	140057	273588
	PACT (8-bit) [36]*	69.95	76028	92035	134395
	XNOR-Net (1-bit) [34]*	58.51	13606	39148	71559
	Pareto-Choice [This Work]*	67.30	59616	87475	121422

*: Executed on Eyeriss-1024

MOGA approach through NSGA-II inherently supports multi-criteria optimization. The designer does not need to handcraft a reward function which fairly captures all the optimization targets in one reward value. For the ImageNet experiment, a HV-leader with an accuracy of 67.02% and hardware estimates comparable to PACT (4-bit) was achieved in only 10 generations and $|P|$=10.

6.1.7 Discussion

HW-FlowQ optimizes CNNs by finding quantization strategies based on high-fidelity HW-model-in-the-loop setups. Abstraction levels and design phases inspired by VLSI design flows help in systematically narrowing down hyper-parameters for both the CNN and hardware design, exposing HW-CNN co-design synergies. Exploring vectorized and bit-serial compute engines, the performance trade-offs for different mixed-precision workloads can be exploited by the GA. The effectiveness of NSGA-II was demonstrated, offering a Pareto-optimal set of quantization strategies for different HW-models during the optimization process. The HW-models introduced in this work provide the metaheuristic method, i.e. the genetic algorithm, with all the necessary details to autonomously make decisions on CNN design and compression. Transitions between

the design abstraction levels can also take place in an automated manner, whenever target application constrains are met. The effectiveness of models, abstraction levels, and metaheuristics is demonstrated in this work, providing a comprehensive example of fully-automated HW-CNN co-design.

6.2 AnaCoNGA: Analytical HW-CNN Co-Design using Nested Genetic Algorithms

After demonstrating the effectiveness of analytical models for automated HW-DNN co-design in HW-FlowQ, a subsequent framework was proposed to push the limits of automation further. AnaCoNGA is an analytical co-design methodology which enables *two* genetic algorithms to evaluate the fitness of design decisions on layer-wise quantization of a neural network and hardware resource allocation. Quantization strategy search (QSS) quantizes weights and activations of each layer to maximize deployment efficiency, while maintaining task-related accuracy. Hardware architecture search (HAS) finds optimal hardware designs which minimize resource utilization and maximize performance metrics of the deployment. AnaCoNGA embeds the HAS algorithm *into* the QSS algorithm to evaluate the hardware design Pareto-front of each considered quantization strategy. The co-design framework harnesses the speed and flexibility of analytical HW-modeling to enable automated, parallel HW-CNN co-design. With this approach, QSS is focused on seeking high-accuracy quantization strategies which are guaranteed to have efficient hardware designs at the end of the search. AnaCoNGA improved task-accuracy by 2.88 p.p. with respect to a uniform 2-bit ResNet20 on CIFAR-10, and achieved a 35% and 37% improvement in latency and DRAM accesses, while reducing LUT and BRAM resources by 9% and 59% respectively, when compared to a standard edge variant of the accelerator.

6.2.1 Introduction

An example of a HW-DNN co-design scenario is the numerical quantization of CNN and the hardware design of a bit-serial accelerator. As mentioned in previous chapters, CNNs benefit from layer-wise and datatype variable numerical precision [7]. To extract the mentioned benefits of variable numerical precision, a hardware accelerator can employ bit-serial computation units [6]. Such an accelerator can have an array of spatially distributed computation units and a distributed on-chip buffer to efficiently provide the computation array with data.

For an automated co-design framework to effectively arrive at a solution that meets an application's constraints, a large and complex solution space must be explored. This motivates the development of a lightweight, easily reconfigurable HW-models, which can be used to evaluate design choices in this complex space [101, 4]. Harnessing the speed and flexibility of such HW-models can enable the *parallel* co-design of HW and CNN, without prohibitive synthesis or cycle-accurate simulation bottlenecks.

AnaCoNGA embeds HAS ***into*** QSS, in a *nested* GA formulation. The main contributions of this work can be summarized as follows:

- Formulating an analytical HW-model for the execution of CNN workloads on a state-of-the-art bit-serial accelerator [6], allowing fast exploration and evaluation of hardware performance and resource utilization, without the need for costly synthesis or cycle-accurate simulation.
- Automating the design of a bit-serial accelerator through MOGA-based HAS, circumventing the need for handcrafted reward functions.

- We insert the HAS loop into the QSS loop. For each potential QSS, the HAS loop efficiently evaluates a 4-D HW design Pareto-front. After synthesis, the HW-CNN co-designed pair achieve a 35% and 37% reduction in latency and DRAM accesses, while achieving 2.88 p.p. higher accuracy compared to a 2-bit ResNet20-CIFAR-10 executing on a standard edge variant of the accelerator.

6.2.2 Related Work

6.2.2.1 CNN-Aware Hardware Design

AutoDNNchip [33] proposes a framework for automated ASIC/FPGA design of a hardware accelerator for a given performance target and a specific CNN model. The design space is explored with a performance model to select candidate hardware architectures, followed by a run-time simulation to optimize their pipelines. MAGNet [32] is an accelerator generator for CNNs based on a reconfigurable, tile-based spatial array. A baseline accelerator is iteratively mapped and evaluated for a target CNN and then tuned using Bayesian search. Both [33] and [32] support mixed-precision computation, but do not explore the layer-wise quantization search space. The works which fall under this category resemble the HAS loop presented in section 6.2.3.3.

6.2.2.2 Joint HW-CNN Co-Design

NHAS [88] aims to find an optimal quantized CNN architecture using an evolutionary algorithm. An efficient hardware dimensioning for the compute array and on-chip memory is searched to accelerate a pool of CNN workloads used as a benchmark. After the hardware is configured, the CNN search space is explored. The hardware evaluation follows a look-up table approach due to the smaller quantization search space considered. This *sequential* approach of co-design can be enhanced by including the hardware design search *within* the CNN search loop. Other works which target joint HW-CNN co-design are [167] and [176], both of which include the hardware's performance in the reward function of an RL-agent and *iteratively* tune both the CNN and hardware architectures. Fine HW-level details, such as scheduling schemes and quantized execution, are not explored in [167], as the optimization loop targets optimally partitioning the CNN workload over a pool of FPGAs. In [176], layer-wise quantization is not supported.

AnaCoNGA is a nested co-design approach to perform hardware design *parallel* to the quantization search, leading to a tight coupling between the HW and CNN, without iteratively or sequentially switching between the two domains. The classification of the mentioned works is shown in Tab. 6.7.

6.2.3 Methodology

In this section, the three main components of AnaCoNGA are presented, namely the analytical accelerator model, the QSS algorithm, and the HAS algorithm, followed by integration of the components into the framework.

Table 6.7: Classification of HW-CNN optimization methods.

Classification	HW Metrics	CNN Compression	HW Design	Parallel Co-design
QSS [7, 84]	✓	✓	✗	✗
HAS [33, 32]	✓	✗	✓	✗
QSS+HAS [176, 167, 88, 10]	✓	✓	✓	✗
AnaCoNGA	✓ (Nested)	✓	✓ (Nested)	✓

6.2.3.1 Bit-Serial Accelerator Modeling for BISMO

Convolutional and fully-connected layers can be lowered into a GEMM by representing the weight tensor $\mathbf{W}^l$ and activation tensor $\mathbf{A}^{l-1}$ of layer l as 2-D matrices Mat_W and Mat_A (equation 6.5). The dimensions m and n represent the rows and columns of each matrix.

$$\begin{gathered}\text{Mat}_W \in \mathbb{R}^{m_W \times n_W},\ \text{Mat}_A \in \mathbb{R}^{m_A \times n_A} \\ \mathbf{A}^l = \text{Conv}(\mathbf{W}^l, \mathbf{A}^{l-1}) = \text{Mat}_W \times \text{Mat}_A\end{gathered} \tag{6.5}$$

Note that transposing both matrices and switching their order would also produce the convolution result. Therefore, we will refer to the matrix positions instead of the datatype for the remainder of the text. LHS is the left-hand side matrix, while RHS is the right-hand side. For readability, m rows and n columns will appear as subscripts of the corresponding matrix when referring to its dimensions, e.g., LHS_n is the number of columns of the left-hand side matrix.

The BISMO accelerator [6], abstracted in figure 6.12, is composed of a $\text{D}_m \times \text{D}_n$ array of PEs. Each PE is responsible for the dot-product of one row of the LHS against one column of the RHS. Due to the bit-serial decomposition of the GEMM operation, the same row and column must be computed as many times as the bitwidths of its operands necessitate. This decomposition is elaborated in [6]. Typically, $\text{D}_m \times \text{D}_n$ is much smaller than the layer's $\text{LHS}_m \times \text{RHS}_n$. The computation must be broken down into smaller $\text{D}_m \times \text{D}_n$ sized tiles. Furthermore, each PE can perform D_k binary dot-products in parallel, whereby the row-column dot-product is computed in tiles of D_k, if the inner-product of LHS and RHS is greater than D_k. To maintain structured parallelism across the computation array, the LHS and RHS matrices are padded to obtain matrices that are divisible by the dimensions of the array. Equation 6.6 shows the computation to obtain the padded dimensions of both matrices.

$$\begin{gathered}\text{Padded_LHS}_m = \text{LHS}_m + \text{D}_m - (\text{LHS}_m \mod \text{D}_m) \\ \text{Padded_LHS}_n = \text{LHS}_n + \text{D}_k - (\text{LHS}_n \mod \text{D}_k) \\ \text{Padded_RHS}_m = \text{Padded_LHS}_n \\ \text{Padded_RHS}_n = \text{RHS}_n + \text{D}_n - (\text{RHS}_n \mod \text{D}_n)\end{gathered} \tag{6.6}$$

With the padded matrices, the number of tiles necessary to complete the computation can be expressed in equation 6.7.

$$\mathrm{T}_m = \mathrm{Padded_LHS}_m/\mathrm{D}_m, \quad \mathrm{T}_n = \mathrm{Padded_RHS}_n/\mathrm{D}_n, \\ \mathrm{T}_k = \mathrm{Padded_LHS}_n/\mathrm{D}_k \tag{6.7}$$

Knowing the size of the padded matrices and the number of tiles necessary to complete the GEMM operation, the size of each tile can be computed in bytes according to figure 6.8, where $\mathrm{LHS}_{\mathrm{bits}}$ is the numerical precision of the LHS elements.

$$\mathrm{LHS_T}_{\mathrm{bytes}} = \frac{\mathrm{Padded_LHS}_m \cdot \mathrm{Padded_LHS}_n \cdot \mathrm{LHS}_{\mathrm{bits}}}{\mathrm{T}_m \cdot 8} \tag{6.8}$$

The total number of binary operations of the workload can be computed simply by observing the dimensions of the GEMM matrices and the bitwidths of each matrix's elements (equation 6.9). However, through the padding action in equation 6.6, superfluous computations were introduced in order to maintain structured parallelism on the PE array. The degree of operation efficiency η_{OPs} can be computed as the ratio of workload-related operations to padded operations, as shown in equation 6.9.

$$\mathrm{Bin_OPs} = \mathrm{LHS}_m \cdot \mathrm{LHS}_n \cdot \mathrm{RHS}_n \cdot \mathrm{LHS}_{\mathrm{bits}} \cdot \mathrm{RHS}_{\mathrm{bits}} \cdot 2 \\ \eta_{\mathrm{OPs}} = \frac{\mathrm{Bin_OPs}}{\mathrm{Padded_Bin_OPs}} \tag{6.9}$$

DRAM requests relating to the LHS depend on both T_m and T_n, whereas RHS elements are only requested T_n times (see equation 6.10). This is a function of BISMO's standard scheduler maintaining reuse of the RHS matrix. Since each T_m of the LHS must be computed against all tiles T_n of the RHS, the scheduler keeps the RHS tiles on chip until they have been used exhaustively. When a new tile of RHS is loaded, all T_m tiles of the LHS must be called again to be computed with it. This has direct implications on execution, which our modeling approach can exploit. Recall in equation 6.5, transposing Mat_W and Mat_A and reversing their order results in the convolution as well. Therefore, depending on the convolutional layer being computed (weights dominated or activation dominated, deep layer vs. shallow layer), choosing the right matrix position (RHS vs. LHS) for weights and activations could improve the inference performance.

$$\mathrm{DRAM}_{\mathrm{LHS}} = \mathrm{T}_m \cdot \mathrm{T}_n \cdot \mathrm{LHS_T}_{\mathrm{bytes}}, \;\; \mathrm{DRAM}_{\mathrm{RHS}} = \mathrm{T}_n \cdot \mathrm{RHS_T}_{\mathrm{bytes}} \\ \mathrm{DRAM}_{\mathrm{Result}} = \mathrm{LHS}_m \cdot \mathrm{RHS}_n \cdot 4, \; \text{32-bit write-back} \\ \mathrm{DRAM}_{\mathrm{Total}} = \mathrm{DRAM}_{\mathrm{LHS}} + \mathrm{DRAM}_{\mathrm{RHS}} + \mathrm{DRAM}_{\mathrm{Result}} \tag{6.10}$$

Finally, looking at the cycles spent for computation, each bit of each tile of each matrix must be computed against the bits from the other matrix. Additional cycles are spent as part of the pipeline for each bit combination on each T_m computed against T_n. The BISMO accelerator overlaps data transfers with computation, resulting in equation 6.11 being sufficiently accurate for design space exploration.

$$\mathrm{Compute_Cycles} = (\mathrm{T}_m \cdot \mathrm{T}_n \cdot \mathrm{T}_k \cdot \mathrm{LHS}_{\mathrm{bits}} \cdot \mathrm{RHS}_{\mathrm{bits}}) + \\ \mathrm{T}_m \cdot \mathrm{T}_n \cdot (8 \cdot (\mathrm{LHS}_{\mathrm{bits}} \cdot \mathrm{RHS}_{\mathrm{bits}} + 1) + 3) + 2 \cdot \mathrm{T}_n \tag{6.11}$$

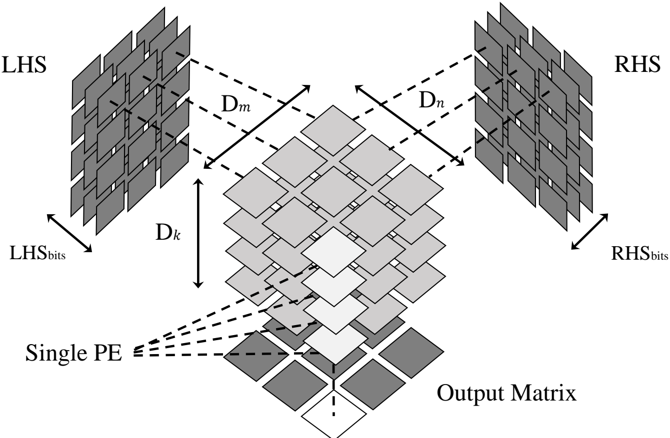


Figure 6.12: High-level abstraction of a bit-serial accelerator [6]: The dimensions $\mathrm{D}_m, \mathrm{D}_n, \mathrm{D}_k$ determine the tiling degree of matrices RHS and LHS.

With this analysis, workload execution metrics can be evaluated with respect to hardware parameters such as compute array dimensions $\mathrm{D}_m, \mathrm{D}_n, \mathrm{D}_k$, as well as on-chip buffer sizes for LHS and RHS without having to synthesize the HW each time. The analytical model introduced in this section is used to perform fast exploration and design of the hardware as an example. It is important to note that AnaCoNGA is not limited to this hardware analytical model; the optimization loops introduced in the next sections can potentially be used to harness the speed and flexibility of more advanced fast analytical hardware models in literature, such as CoSA [101], GAMMA [102] or ZigZag [177].

6.2.3.2 Genetic Quantization Strategy Search (QSS)

The quantization strategy search (QSS) is essentially the search space introduced in HW-FlowQ (section 6.1.3.2). To recap, for an L-layer CNN, there are Q^{2L} solutions, where Q is the set of possible quantization levels for weights and activations. It is important to note that QSS can be applied to any quantization technique (DoReFa [35], PACT [36], or others), as it only tries to find the best bitwidths for each layer and datatype. A MOGA is used to tackle the multi-criteria optimization problem of maximizing accuracy and minimizing hardware execution complexity. No hardware design takes place in this search.

An initial population $\mathcal{P}_0$ is randomly generated, with each genome encoding a quantization strategy, i.e. a quantization tuple ($\mathbf{W}^l_{\text{bits}}$, $\mathbf{A}^{l-1}_{\text{bits}}$) for each layer of the CNN (explicit, bijective encoding). The genomes of $\mathcal{P}$ are briefly fine-tuned and evaluated based on their task-accuracy on a validation set. When using *standalone* QSS, the GA must additionally consider the fitness of the quantized CNNs on hardware estimates (DRAM accesses and computation cycles). Based on the three fitness metrics, the *Pareto optimality* of each individual is identified with respect to the population $\mathcal{P}$. The population goes through phases of selection, crossover and mutation for the subsequent generations, producing Pareto-optimal CNN quantization strategies.

Table 6.8: Hardware configurations used for model validation.

Config	$D_m \times D_n$	D_k	LHS_Buffer	RHS_Buffer
HW1	4×4	128	128KB	128KB
HW2	4×8	256	128KB	256KB
HW3	8×8	256	256KB	256KB

6.2.3.3 Genetic Hardware Architecture Search (HAS)

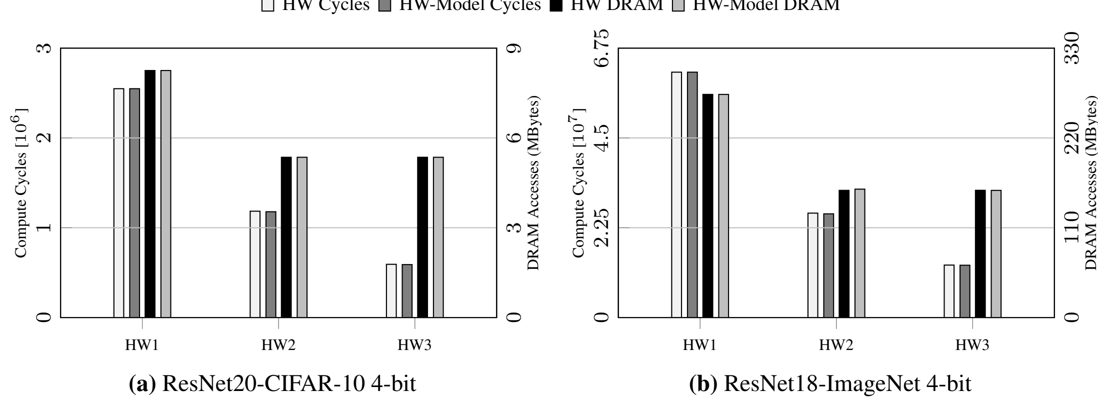
A second GA is formulated to allocate and optimally dimension the hardware. Each individual's genome captures hardware design decisions, namely D_m, D_n, D_k, LHS_Buffer, and RHS_Buffer at each genetic locus. Here, uniform crossover is used instead of single-point, as maintaining neighboring genetic loci does not necessarily lead to better solutions, since each hardware dimension affects a different dimension of the GEMM operation. The fitness criteria of this GA are the hardware design's execution performance (compute cycles and DRAM accesses) of a *predetermined* quantized CNN, as well as the amount of FPGA resources (BRAMs and LUTs) it requires for its allocation. To estimate the FPGA resource utilization of a genome, we use the model proposed by Umuroglu et al. [65]. For performance criteria, the model presented in section 6.2.3.1 is used. NSGA-II is applied to this 4-dimensional solution space as it can return a multitude of Pareto-optimal solutions for the designer to choose from.

6.2.3.4 Model Validation and Real Hardware Measurements

To validate the proposed analytical hardware model, we synthesize three differently dimensioned BISMO accelerators, detailed in table 6.8. HW1 and HW3 represent small and large accelerators, while HW2 has an asymmetric processing element array. Small and large GEMM operations are tested on all three accelerators by executing all the convolutional and fully-connected layers of ResNet20 for CIFAR-10 (small) and ResNet18 for ImageNet (large), with 4-bits for weights and activations. The hardware measurements are collected from profiling registers built into the accelerator and read out at the end of the execution. In figure 6.13, the results of the HW-model's accuracy and fidelity are presented, compared to the synthesized hardware. Compute cycles and DRAM accesses are the considered execution metrics. The high accuracy and correlation of the measured and estimated values make the model well-suited for design space exploration. This HW-model is used for exploration, however all the results reported in the final table 6.10 are on real, synthesized hardware. The HW-model circumvents the need for repeated synthesis, HIL setups, cycle-accurate models, or construction of a look-up table. This drastically speeds up experiments, allowing us to run the GA for more generations and larger populations, resulting in better search outcomes.

6.2.3.5 AnaCoNGA: Nested HW-CNN Co-Design

The QSS and HAS loops present a causality dilemma: what comes first, the optimized hardware or the CNN quantization strategy? In section 6.2.2, we mention existing methods which tackle



(a) ResNet20-CIFAR-10 4-bit

(b) ResNet18-ImageNet 4-bit

Figure 6.13: Validation of the HW-model vs. real HW measurements for compute cycles and DRAM accesses on three BISMO configurations (HW1-3). Small and large workloads are verified from ResNet20-CIFAR-10 (left) and ResNet18-ImageNet (right).

this challenge sequentially or iteratively. To perform true co-design, both hardware and CNN need to be jointly and **concurrently** considered.

One approach is to combine HAS genomes with QSS genomes into one GA. However, this would result in a prohibitively complex, large search space (4.77×10^{37} for ResNet20 on the considered bit-serial accelerator), with many direct and indirect relationships between the hardware and quantization parameters. The complex, joint search space would also necessitate larger populations and generations for the GA, leading to excessive GPU hours. Another approach could be to iterate between the search spaces [88, 167, 176]. An iterative approach brings us to the same dilemma, since the hardware was initially biased for a different quantization strategy, and a newly found HW-CNN combination is sub-optimal with respect to another combination, which had a different quantization strategy prior.

To tackle this challenge, the two genetic algorithms are *nested* in AnaCoNGA, as shown in figure 6.14. On the one hand, the HAS GA requires roughly $\sim$1.5 minutes to execute for 200 generations and 200 hardware genomes and can be *parallelized*. This is due to the fast analytical HW-model in section 6.2.3.1, and the LUT/BRAM utilization models proposed in [65]. On the other hand, the QSS genetic algorithm requires some epochs of fine-tuning to evaluate the accuracy of a potential quantization genome. This can be a costly fitness evaluation process for larger networks and datasets. When nesting the HAS GA into the QSS GA, we can exploit the **speed** of the HAS loop to evaluate the hardware design Pareto-front for **each** considered quantization genome (parallel HAS blocks in figure 6.14). In each HAS experiment, a 4-D Pareto-front of hardware designs is generated for the respective quantization genome. The 4-D hardware Pareto-front is checked for solutions that meet our target hardware constraints. If no solution in the HAS Pareto-front satisfies our hardware requirements, then the QSS receives a signal to *remove* the genome's fine-tuning step and assign it a null accuracy, without wasting any GPU training time (feedback line from HAS to QSS in figure 6.14). With this approach, the QSS is relieved from optimizing hardware metrics and can now be reformulated into a *single-objective* genetic algorithm (SOGA), which is solely *focused* on improving the accuracy of the quantized

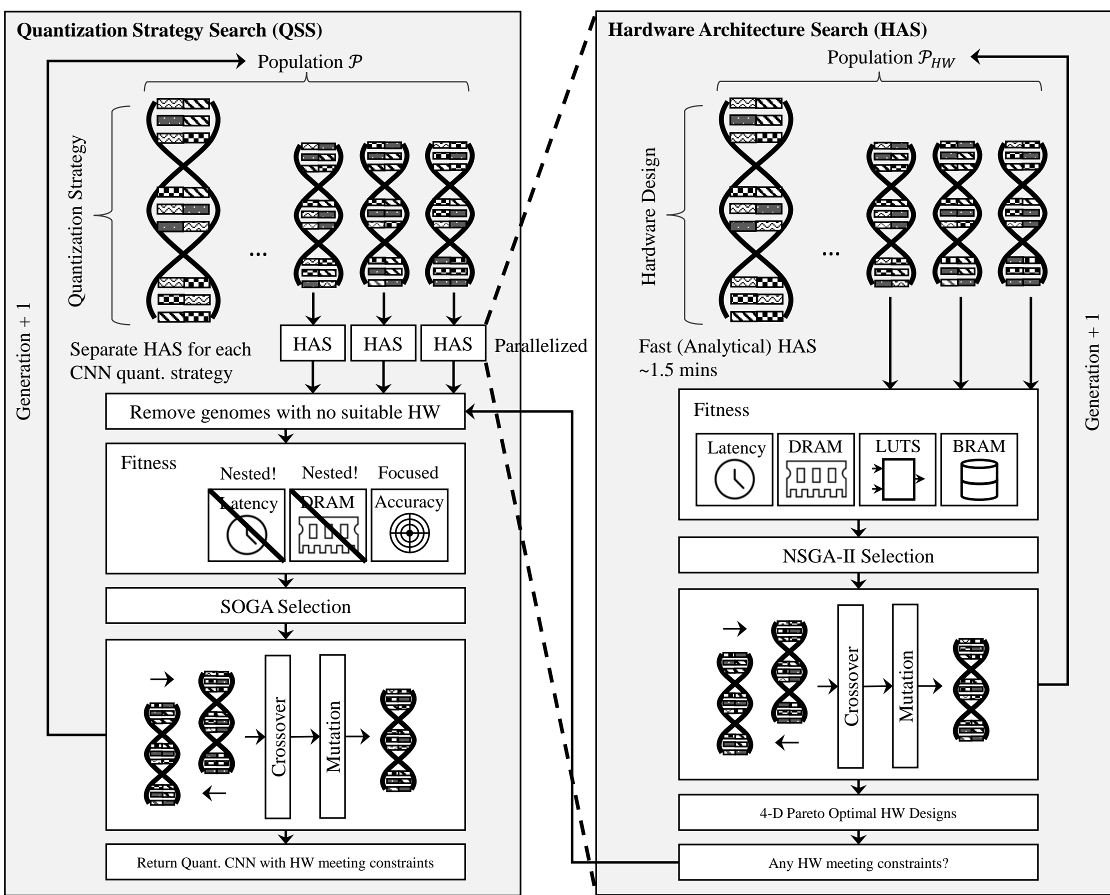


Figure 6.14: AnaCoNGA: Each individual from QSS executes its *own* HAS MOGA. Any QSS individual can prove itself efficient on its *own* hardware design to get a chance for its accuracy to be evaluated. QSS is relieved from optimizing hardware and is transformed to a SOGA (i.e. accuracy focused).

CNNs. QSS essentially allows each quantization genome to evaluate its own hardware design space before accepting them into the population. Therefore, two radically different QSS genomes could meet the target hardware constraints (DRAM, computation cycles, BRAM and LUTs) by finding themselves *specialized* hardware designs in their respective HAS explorations. This way, the hardware design remains ***flexible*** (undefined) on the scale of the overall experiment, but is guaranteed to exist for any genome which is eventually chosen by the QSS at the end of the search. AnaCoNGA's design loops enable the use of analytical HW-models such as [101, 102, 177], harnessing their speed and flexibility to achieve parallel, fully-automated, HW-CNN co-design.

The described nested search algorithm is elaborated explicitly in algorithm 6.1.

Algorithm 6.1 Nested Genetic Algorithm: AnaCoNGA Co-Design

```
P ← GenerateInitPopulation(PopSize)
HW ← SetTargets(LUT, BRAM, Cyc, DRAM)
P ← EvaluateFitness(P, HW)                                  ▷ Init Population Eval.
while P.Generation < FinalGeneration do
    O ← Crossover(P)                                        ▷ Produce Offspring
    O ← Mutate(O, p_m)                                      ▷ Mutate with probability p_m
    P ← EvaluateFitness(O, HW)                              ▷ Evaluate O and append to P
    P ← SOGASelection(P)                                    ▷ Select only w.r.t. Accuracy
return P.HallOfFame                                         ▷ Return best solutions
function EVALUATEFITNESS(P, HW)
    for ρ in P do                                           ▷ Each individual ρ in population P
        ρ.HWParetoFront ← HAS_NSGA2(ρ)                      ▷ MOGA
        if anyHW in i.HWParetoFront satisfies HW then
            ρ.Acc ← Finetune(ρ)                             ▷ GPU time for individual
        else                                                ▷ No guarantee from HAS on efficient HW
            ρ.Acc ← null                                    ▷ Discarded at SOGA selection
```

6.2.4 Evaluation

6.2.4.1 Experimental Setup

AnaCoNGA is evaluated on CIFAR-10, CIFAR-100, and ImageNet datasets. The 50K train and 10K test images of CIFAR-10 and CIFAR-100 are used to train and evaluate the quantization strategies. ImageNet consists of ∼1.28M train and 50K validation images. After an ablation study, we set the population size and number of generations to 50 for QSS GAs on ResNet20. Probabilities for mutation and crossover are set to 0.5 and 1.0, respectively. For ResNet56, we reduce the running population size $|\mathcal{P}|$ to 25. For ResNet18-ImageNet experiments, $|\mathcal{P}|$ is set to 25 and the number of generations is reduced to 25. The CNNs trained on CIFAR-10 are fine-tuned for 3 epochs and evaluated on 10K random samples during the search. For ImageNet, we fine-tune for 1.5 epochs before evaluating on the valid-set. The quantization method for ResNet20 experiments is DoReFa [35], while deeper (ResNet56) and higher resolution (ResNet18) experiments use the PACT method [36]. Results denoted with (2, 4-bit) indicate 2-bit weights and 4-bit activations. For comparison, binarized variants are trained using the XNOR-Net method [34].

The Xilinx Z7020 SoC on the PYNQ-Z1 board is used as the target platform for all hardware experiments in table 6.10 and figure 6.16, with all designs synthesized at a 200MHz target clock frequency. For HAS experiments, both the population size and generations are set to 200, since no significant improvement was observed for larger experiments. Mutation and crossover probabilities are set to 0.4 and 1, respectively. In table 6.10, AnaCoNGA's nested HAS GA uses the respective (2, 4-bit) configuration's hardware performance as its hardware constraint/target.

Table 6.9 shows the valid alleles which can be used in genomes. The $\not\equiv$ symbol indicates the parameters that can take different values within a genome. Valid alleles are chosen within ranges

of dimensions synthesizable on the Z7020. Larger designs (larger search space) are feasible on larger FPGAs [65].

Table 6.9: Hardware and quantization search space.

Parameter	Valid Alleles
D_m , D_n	2, 4, 6, 8, 10, 12, 14, 16, 32, 48, 64 $D_m \not\equiv D_n$
D_k	64, ..., 512, steps of 32
LHS_Depth RHS_Depth	32, 64, 128, 256, 512, 1024, 2048, 3072, 4096 Bits per D_k, per D_m or D_n, LHS_Depth $\not\equiv$ RHS_Depth
$\mathbf{W}^l_{bits}, \mathbf{A}^{l-1}_{bits}$	1, 2, 3, ..., 8 $\mathbf{W}^l_{bits} \not\equiv \mathbf{A}^{l-1}_{bits}$, independent for each $l \in L$

6.2.4.2 Quantization Strategy Search (QSS) Loop Evaluation

To evaluate the *standalone* QSS space, the accelerator dimensions are fixed to measure the hardware fitness metrics of each potential solution considered by the GA. The HW3 configuration is chosen from table 6.8, as it is one of the BISMO configurations proposed in [6] and used as the standard CNN edge accelerator in [7]. Figure A.1 shows 2-D projections of the 3-D search space of the QSS loop compressing ResNet20 for the CIFAR-10 dataset. To visualize the progress of the algorithm, old-generation Pareto-fronts are plotted in grayscale (darker points indicate newer generation Pareto-fronts), while red crosses belong to the final Pareto-front. The projections reveal a loose correlation between DRAM accesses and compute cycles, as well as a convex Pareto-front between prediction accuracy and hardware efficiency. From this Pareto-front of quantization strategies, a solution can be chosen to fit the needs of the application. After fine-tuning uniformly sampled individuals from the Pareto-front, solutions ranging from 89.44% down to 86.45% in Top-1 prediction accuracy are found, with increasing degrees of hardware efficiency. Picking the solution which gives us an accuracy of 89.44% provides a reduction of 55.4% in DRAM accesses and 74% in computation latency with respect to a uniform 8-bit execution. In table 6.10, the results of standalone QSS solutions chosen for ResNet20 are detailed, as well as the more difficult search problem of ResNet56 quantization, which has a larger quantization search space (recall QSS space = Q^{2L}), for both CIFAR-10 and CIFAR-100. The results show standalone QSS producing non-dominated strategies with respect to uniform quantization, on the HW3 BISMO design.

6.2.4.3 Hardware Architecture Search (HAS) Loop Evaluation

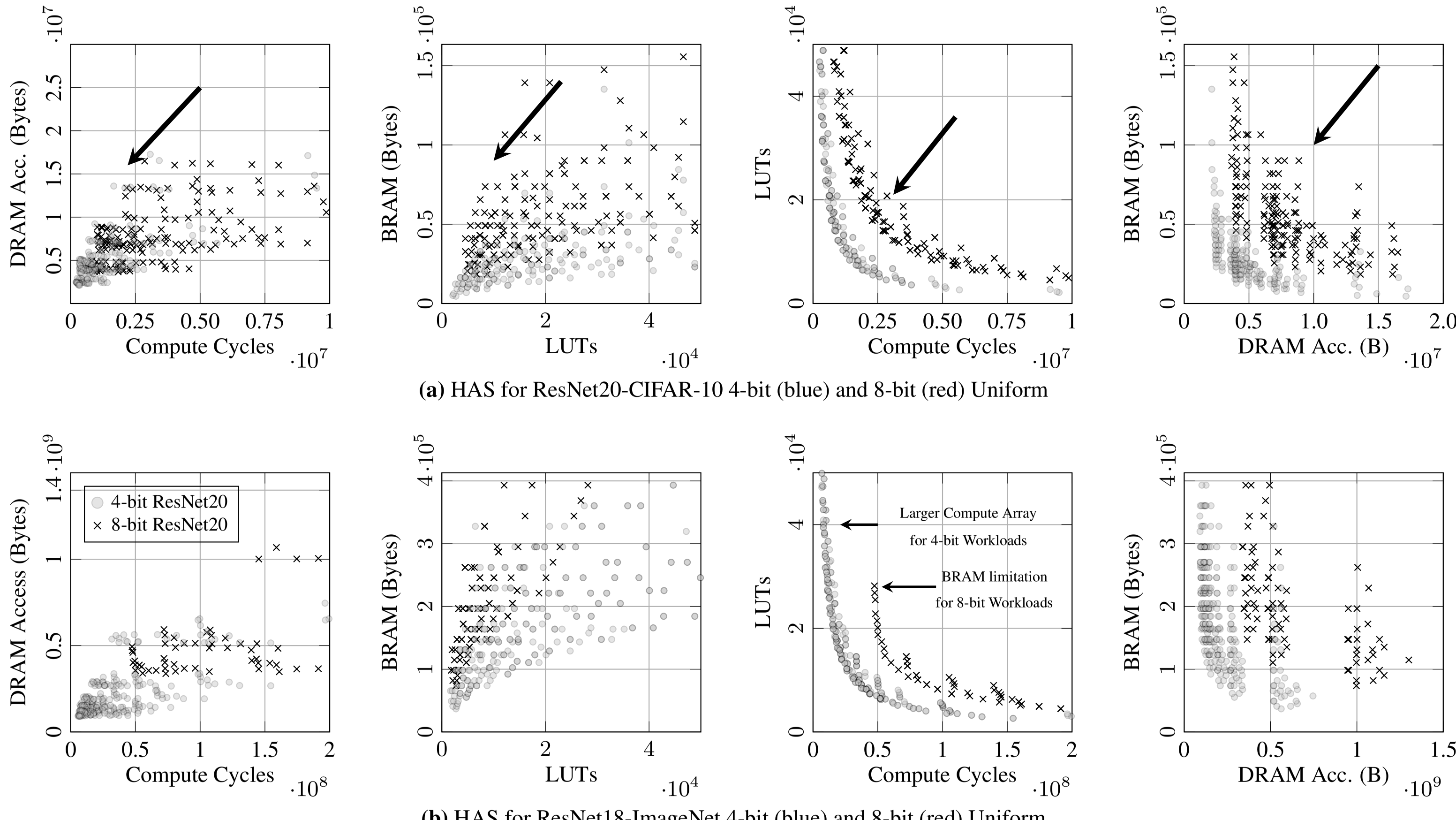
To evaluate the standalone HAS loop, the hardware search is executed for uniform 8-bit and 4-bit variants of ResNet20 for CIFAR-10 and the larger ResNet18 for ImageNet. Figure 6.15 shows 2-D projections of the final 4-D Pareto-fronts achieved by HAS, optimizing for computation latency, DRAM accesses, and BRAM and LUT utilization. A clear shift can be noticed in the hardware design space when the quantization strategy changes, even for the tested uniform

strategies. Another observation is that the shift is not only due to the more efficient execution metrics of 4-bit vs. 8-bit, but also due to new legal scheduling options on differently dimensioned accelerators. This can be seen in the non-overlapping circle and cross markers of the BRAM vs. LUTs 2-D projection plot, indicating different hardware dimensions being optimal for the 4-bit and 8-bit CNNs, while respecting the resource limitations of the Z7020 FPGA. The 2-D projection of computation cycles against LUTs for ResNet18 shows the effect of legality checks in the analytical model on the shape of the Pareto-front. The cross marks (8-bit) do not extend beyond 28K LUTs of logic, indicating that larger computation arrays cannot allocate the associated BRAM requirements to fit a single minimum-sized tile of computation for one or more of the ResNet18 layers (i.e., design not synthesizable, or computation not possible). Therefore, the size of synthesizable computation arrays with sufficient BRAM to feed the array with minimum tile sizes is restricted with the 8-bit CNN. On the other hand, the circle markers (4-bit) of the same plot extend to larger LUT utilization, indicating the existence of large *synthesizable* compute arrays, with sufficient BRAM to load smaller tiles of the smaller 4-bit ResNet18.

Further HAS results are presented in table 6.10, applied to the strategies found in the QSS experiments of the previous section (labeled QSS+HAS). From the resulting Pareto-fronts, candidates with the lowest execution and DRAM access cycles are chosen for synthesis, without exceeding the resource utilization of the HW3 BISMO choice from table 6.8. The GA finds non-trivial asymmetric hardware configurations ($\mathrm{D}_m \neq \mathrm{D}_n$), which exploit the position of the tensors $\mathbf{W}^l$ and $\mathbf{A}^{l-1}$ into either LHS or RHS matrices. The asymmetric hardware allows the scheduler to swap the position of weights and activations in the middle of the CNN execution, to maintain high computation efficiency and low DRAM accesses, by reusing the datatype placed in the RHS matrix of the computation. This naturally reduces the LUT and BRAM requirements of the design. Figure A.2 shows the layer-wise execution details of a 4-bit ResNet18-ImageNet on an asymmetric $\mathrm{D}_m \times \mathrm{D}_n \times \mathrm{D}_k$ = 8×14×96 hardware configuration found through HAS. For comparison, the same workload is executed on the symmetric HW3, which has higher theoretical peak binary trillion operations per second (TOPS) (6.55 binary TOPS vs. 4.30 binary TOPS). The HAS solution heavily reduces the amount of DRAM accesses for all the layers. This indicates better tiling dimensions and compute efficiency η_{OPs} with respect to workloads, which naturally brings down the computation cycles and reduces the chances of stalls. For the HAS asymmetric solution, layers 1-5 and 12-17 are executed with weights on the LHS, while other portions of the CNN are executed with weights on the RHS. The layers are not schedulable otherwise, indicating that the HAS solution is tightly-coupled with the schedule and the legality checks of the analytical model, allocating sufficient resources to guarantee at least a single efficient and legal scheduling for each workload of the CNN exists, thereby reducing the FPGA's resource utilization. In table 6.10, the real hardware measurements of sequential co-design (QSS+HAS) show a clear advantage to all standalone QSS CNNs, dramatically lowering their DRAM accesses and latency below or equivalent to a 1-bit strategy executing on standard BISMO dimensions from [7], with less LUT and BRAM required for the design.

6.2.4.4 Analysis of AnaCoNGA Co-Designed Solutions

In table 6.10, the results of all the considered networks, datasets, and search combinations are shown executed on **synthesized** hardware. The uniform bitwidth CNNs are paired with the

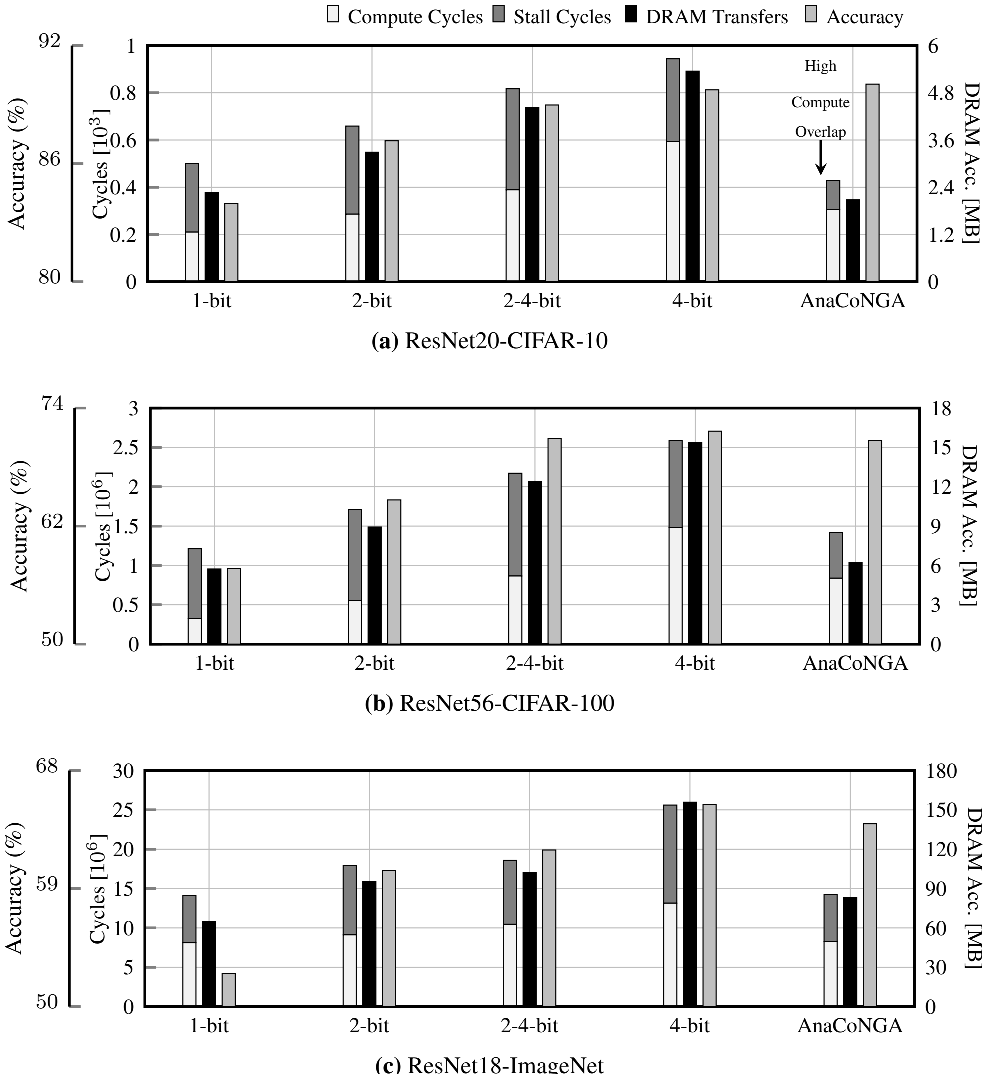


(a) HAS for ResNet20-CIFAR-10 4-bit (blue) and 8-bit (red) Uniform

(b) HAS for ResNet18-ImageNet 4-bit (blue) and 8-bit (red) Uniform

Figure 6.15: HAS: 2-D projections of a 4-D Pareto-front in a multi-objective search space. The GA optimizes for hardware resources (LUTs, BRAM) and performance metrics (DRAM accesses, execution cycles) for ResNet20 (top) and ResNet18 (bottom). The proposed analytical model allows for fast exploration and evaluation of solutions.

edge BISMO variant used in [7]. An improvement is observed in task-related accuracy for *all* AnaCoNGA solutions over sequential co-design (QSS+HAS). This can be attributed to the *accuracy-focused* SOGA implemented in the QSS of AnaCoNGA, which leaves the HAS to be handled by the nested MOGA (recall figure 6.14). Furthermore, the nested HAS allows more diverse, high-accuracy quantization individuals to survive through QSS, as each QSS individual can find their *own* hardware design to meet the application constraints.



(a) ResNet20-CIFAR-10

(b) ResNet56-CIFAR-100

(c) ResNet18-ImageNet

Figure 6.16: Breakdown of execution on synthesized hardware. Higher DRAM accesses are correlated with lower compute efficiency and stalls. AnaCoNGA reduces latency and DRAM accesses while maintaining high accuracy.

The latency and DRAM accesses of AnaCoNGA and QSS+HAS variants are comparable to or better than a *single-bit* network executing on the handcrafted accelerator. As mentioned in section 6.2.4.3, the HAS MOGA finds asymmetric hardware designs and relies on the scheduler

Table 6.10: Quantization and hardware design experiments. Uniform and standalone QSS are executed on a standard edge variant (HW3) used in [7]. Latency and DRAM are measured on hardware.

Model	Work	Acc [%]	LUT Util	BRAM Blocks	Latency K. Cycles	DRAM Acc. MB	HW Config. $D_m \times D_n \times D_k$, LHS, RHS_Buf	Peak Bin. TOPS
ResNet20 CIFAR-10	XNOR (1-bit) [34]	83.98	32639	135	501	2.26	8×8×256, 256KB, 256KB	6.55
	DoReFa (2-bit) [35]	87.16			659	3.29		
	DoReFa (2,4-bit) [35]	88.98			817	4.43		
	DoReFa (4-bit) [35]	89.75			944	5.35		
	QSS (standalone)	89.44			798	4.17		
	QSS+HAS	89.44	29687	55	422	1.99	8×16×64, 32KB, 32KB	3.28
	AnaCoNGA	**90.04**	**29671**	**55**	**428**	**2.08**	8×16×64, 16KB, 32KB	
ResNet56 CIFAR-10	XNOR (1-bit) [34]	85.61	32639	135	1212	5.73	8×8×256, 256KB, 256KB	6.55
	PACT (2-bit) [36]	90.28			1710	8.93		
	PACT (2,4-bit) [36]	92.97			2172	12.41		
	PACT (4-bit) [36]	93.27			2585	15.37		
	QSS (standalone)	91.89			2120	11.98		
	QSS+HAS	91.89	29643	79	1242	5.44	4×32×64, 16KB, 64KB	3.28
	AnaCoNGA	**92.31**	**29638**	**79**	**1315**	**5.83**	4×32×64, 8KB, 64KB	
ResNet56 CIFAR-100	XNOR (1-bit) [34]	57.70	32639	135	1212	5.73	8×8×256, 256KB, 256KB	6.55
	PACT (2-bit) [36]	64.66			1710	8.93		
	PACT (2,4-bit) [36]	70.91			2172	12.41		
	PACT (4-bit) [36]	71.65			2585	15.37		
	QSS (standalone)	69.52			2054	11.60		
	QSS+HAS	69.52	29638	79	1240	5.45	4×32×64, 8KB, 32KB	3.28
	AnaCoNGA	**70.68**	**29643**	**79**	**1420**	**6.23**	4×32×64, 16KB, 64KB	
ResNet18 ImageNet	XNOR (1-bit) [34]	52.51	32639	135	14090	64.93	8×8×256, 256KB, 256KB	6.55
	PACT (2-bit) [36]	60.36			17932	95.23		
	PACT (2,4-bit) [36]	61.94			18596	102.07		
	PACT (4-bit) [36]	65.40			25609	155.85		
	AnaCoNGA	**63.94**	**28035**	**123**	**14250**	**83.07**	8×14×96, 192KB, 168KB	4.30

to switch the order of weights and activations in LHS or RHS, depending on the layer being executed. This leads to lower LUT and BRAM requirements for all the HAS-based designs, while executing more efficiently than the oversized HW3. All AnaCoNGA-based hardware designs are *smaller* (fewer peak binary TOPS) than HW3, but achieve *better* performance due to their tightly-coupled dimensioning, which improves their compute efficiency. To better understand AnaCoNGA's hardware performance, the total execution time is split and the cycles are measured with respect to compute, as well as the non-overlapping cycles spent on other parts of the pipeline (stall cycles). This data is presented in figure 6.16. Although the HAS genetic algorithm is not aware of pipeline stalls, it optimizes for minimal compute cycles and lower DRAM accesses, where, particularly the latter, is correlated with *lower pipeline stalls*. Hardware designs with these traits naturally bring down stall cycles, leading to higher compute and memory access overlap. For figure 6.16-a, the AnaCoNGA solution indeed has *higher compute cycles* than 1-bit due to its higher bitwidths, which results in a higher accuracy CNN. However, the DRAM accesses are well-optimized, due to HAS designing an accelerator which achieves efficient compute tiles and high compute efficiency η_{OPs}, resulting in *fewer stall cycles*, ultimately bringing the total latency of the execution *below* 1-bit on the HW3 edge BISMO design, while maintaining task-related accuracy *higher* than a uniform 4-bit solution. Similar trends can be observed in figure 6.16

for ResNet56 and ResNet18 as well, achieving lower execution metrics than 2-bit CNNs and maintaining high task-related accuracies.

AnaCoNGA also brings benefits in terms of reduced GPU hours. For ResNet20 and ResNet56 on CIFAR-10, QSS and AnaCoNGA were run on a single NVIDIA Titan RTX GPU. The search took 14 hours for ResNet20 with AnaCoNGA, which is a 51% reduction with respect to *standalone* QSS. For ResNet56, a 24% reduction in GPU hours was achieved, leading to 34 hours of search time. Overall, the nested HAS constraint analyzing the 4-D Pareto-fronts of all QSS genomes, allowed the SOGA to skip the evaluation of accuracy for genomes which had no promising hardware designs.

6.2.5 Discussion

AnaCoNGA is a HW-CNN co-design framework using two GAs, QSS and HAS, combined in a novel nested scheme to eliminate handcrafted reward functions, iterative switching between the two domains, and fine-tuning CNN genomes with sub-optimal HW-design spaces. The speed and flexibility of analytical HW-models were harnessed to achieve true parallel co-design, while reducing the overall search time when compared to iterative or sequential approaches. This fully-automated co-design approach requires no intervention of human experts during the optimization process and achieves non-trivial synergies which would be very difficult for an expert to predict. Counterintuitively, by searching both the hardware and neural network design spaces, the optimization was *faster* than only searching one design space. This lowers the effort on both the ML and HW engineer and achieves better results in both domains. The accuracy of ResNet20-CIFAR-10 was improved by 2.88 p.p. compared to a uniform 2-bit CNN, and achieved a 35% and 37% improvement in latency and DRAM accesses, while reducing LUT and BRAM resources by 9% and 59% respectively, when compared to an edge variant of the accelerator. AnaCoNGA is a prime example of how metaheuristic techniques and well-defined, parameterizable analytical models can provide fully-automated co-design in the final development stages of a DNN deployment.

7 Conclusion & Outlook

COMPLEXITIES of abstract artificial algorithms are only truly understood when their implementation in the real world is needed. This is exacerbated when the algorithm and the execution medium are designed in a segregated manner. Co-design brings algorithm and medium under the same scope, manifesting a real-world implementation with synergies that bring them closer to algorithms observed in nature.

This dissertation presented several challenges in implementing DNNs on hardware. These challenges were hard to resolve without compromises in DNN and/or hardware design targets. The pitfalls of incoherent co-design were highlighted with examples of how compromises in DNN targets do not result in benefits on the target hardware and vice versa. This problem statement was addressed by applying classical concepts from the field of VLSI design and HW-SW co-design. These included different *methodologies*, *executable models*, and *design abstraction levels*.

Handcrafted methodologies were presented, where the designer's conceptual understanding of the design challenge is itself the problem formulation [8, 16]. Semi-automated methods were used when parts of the design challenge were solvable with computation models, but guided by human designers [24, 25]. Fully-automated methods tackled challenges with prohibitively large search spaces, but well-defined models and evaluation criteria [9, 10, 11]. The designer essentially takes their hands off the wheel and allows metaheuristic methods to search for the optimal parameters for both hardware and DNN design. Executable models were introduced in several forms, facilitating the injection of hardware-awareness into DNN optimization loops. Look-up tables and regression-based models were developed for off-the-shelf hardware platforms [12, 15, 17, 19, 20], classical SDF-style models were used to parameterize dataflow hardware architectures [21, 22, 23, 24, 25], and analytical models were used to explore mapping and scheduling schemes on differently dimensioned spatial accelerators [9, 10, 11, 12, 13]. Going a step further, differentiable hardware models were developed, proving that hardware optimization does not need to break the smooth, gradient-based training operation of DNNs, but can even be directly injected as part of the learning and backpropagation algorithm of the DNN [15]. This allows the DNN to learn the task at hand as well as learn how to run efficiently on hardware. Finally, the large search spaces and costly evaluation and training times motivated the use of divide-and-conquer approaches to tackle complex design challenges. The introduction of abstraction levels into the design flow allowed the human and/or the metaheuristic agent to focus on solving sub-parts of the problem instead of tackling it once as a whole and landing in incoherent co-design neighborhoods [8, 24, 25, 10, 11]. Focusing on a limited set of design details reduces the development effort, similar to how VLSI engineers' work is integrated at different levels of abstraction, allowing them to consider less complex problems at each stage. The works published under the scope of this dissertation

repeatedly showed the effectiveness of applying these concepts to HW-DNN development processes and brought applications with societal impact to edge devices.

Moving forward, as more AI algorithms are introduced to new applications, a single SoC might need to be optimized for a multitude of DNN architectures. Such an SoC, containing several accelerators and many DNNs, faces new design challenges that are still to be tackled. For example, in the context of an autonomous vehicle, some AI-based tasks can be safety-critical, while others can be executed with best effort. Tasks may have hard deadlines, soft deadlines, or no deadlines at all. They may appear stochastically or deterministically, in bursts or individual events. Some tasks may control the vehicle, while others can run as "shadow-mode" features. Considering the limited area and power on a battery-powered vehicle, this large set of AI-based tasks must be executed on the system with limited, shared resources. The edge SoC must also run other general tasks which are not necessarily AI-based. These tasks add to the communication traffic on the SoC's interconnect, resulting in further complexity in deterministically estimating the execution time of DNNs by the on-chip hardware accelerators. Considering these challenges, new opportunities for co-design emerge at the system level, such as mixed-critical scheduling schemes, DNN sub-graph execution and preemption concepts, sensor data reuse, shared backbones among different DNN tasks, and more.

In addition to being used in more applications, the tasks performed by AI algorithms are becoming harder to quantitatively evaluate. For example, the performance of creative AI art, text, or music generators cannot always be captured by a single metric such as "accuracy" for classification or mIoU for semantic segmentation. The evaluation of the output becomes more subjective. When the AI's task-related performance metrics become harder to define, DNN compression schemes lose their guiding fitness criteria, which makes it more difficult to definitively judge which architecture is better than the other. This can lead to rethinking the methods by which execution efficiency on hardware and task-related performance are weighed against each other.

The field of AI is continuously evolving. The same goes for the field of hardware design. As new algorithms are developed, engineers must have the foresight to evaluate how existing and new hardware architectures handle the associated computational effort. As an example, spiking neural networks (SNNs) are a class of machine learning algorithms, where the neurons are not only sensitive to the magnitude of the incoming input activations, but also their time of arrival [178]. This allows the network to learn temporal relationships among activations, which are referred to as *spikes* in this context. This aspect challenges many of the notions that are taken for granted when designing hardware for state-of-the-art DNNs. However, the concepts presented in this dissertation could be reapplied to ease the development effort of hardware for more exotic forms of neural networks, while concurrently optimizing their algorithmic complexity in software. As artificial neural networks become more accurate models of biological brains, the hardware will undoubtedly also morph into biologically-inspired architectures, moving away from the biases that emerged after decades of classical computing. Engineers in the field of neuromorphic computing work on non-classical compute paradigms, which are designed to capture neuron characteristics more accurately than standard arithmetic digital hardware blocks, instruction-based data processing, or classical memory hierarchies [179]. While reimagining how future neuromorphic computers should be designed, the algorithms which are meant to run on them must always be considered at each stage of research and development. Here, handcrafted

and automated methodologies will be followed, accurate executable models will be critical, and abstraction levels will be introduced to understand the design problem with reasonable detail at each stage of development.

Human-engineered algorithms will behave like biological ones at some point. They will very likely surpass the intelligence observed in biological beings in nature. To take the first steps towards that goal, the fundamental way in which algorithms in nature and their execution medium interact with the real world must be understood. The hardware and the algorithm must become one and the same.

Bibliography

[1] Xilinx, Inc. *XILINX 7 Series DSP48E1 Slice*, 2018. v1.10.

[2] F. Vahid and T. D. Givargis. *Embedded System Design: A Unified Hardware/Software Introduction*. Wiley, 2001.

[3] Y.-H. Chen, J. Emer, and V. Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *International Symposium on Computer Architecture*. IEEE, 2016. doi:10.1109/ISCA.2016.40.

[4] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer. Timeloop: A systematic approach to dnn accelerator evaluation. In *International Symposium on Performance Analysis of Systems and Software*. IEEE, 2019. doi:10.1109/ISPASS.2019.00042.

[5] B. Research. Deepbench. URL: https://github.com/baidu-research/DeepBench.

[6] Y. Umuroglu, L. Rasnayake, and M. Själander. BISMO: A scalable bit-serial matrix multiplication overlay for reconfigurable computing. In *International Conference on Field Programmable Logic and Applications*. IEEE, 2018. doi:10.1109/FPL.2018.00059.

[7] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han. HAQ: Hardware-aware automated quantization with mixed precision. In *Conference on Computer Vision and Pattern Recognition*. IEEE, 2019. doi:10.1109/CVPR.2019.00881.

[8] N. Fasfous, M.-R. Vemparala, A. Frickenstein, and W. Stechele. Orthruspe: Runtime reconfigurable processing elements for binary neural networks. In *Conference on Design, Automation and Test in Europe*. IEEE, 2020. doi:10.23919/DATE48585.2020.9116308.

[9] N. Fasfous, M.-R. Vemparala, A. Frickenstein, E. Valpreda, D. Salihu, J. Höfer, A. Singh, N.-S. Nagaraja, H.-J. Voegel, N. A. V. Doan, M. Martina, J. Becker, and W. Stechele. Anaconga: Analytical hw-cnn co-design using nested genetic algorithms. In *Conference on Design, Automation and Test in Europe*, 2022.

[10] N. Fasfous, M.-R. Vemparala, A. Frickenstein, E. Valpreda, D. Salihu, N. A. V. Doan, C. Unger, N. S. Nagaraja, M. Martina, and W. Stechele. HW-FlowQ: A multi-abstraction level hw-cnn co-design quantization methodology. *Transactions on Embedded Computing Systems*, 2021. doi:10.1145/3476997.

[11] M.-R. Vemparala, N. Fasfous, A. Frickenstein, E. Valpreda, M. Camalleri, Q. Zhao, C. Unger, N. S. Nagaraja, M. Martina, and W. Stechele. Hw-flow: A multi-abstraction level hw-cnn codesign pruning methodology. *Leibniz Transactions on Embedded Systems*, 2022.

[12] P. Mori, M.-R. Vemparala, N. Fasfous, S. Mitra, S. Sarkar, A. Frickenstein, L. Frickenstein, D. Helms, N. S. Nagaraja, W. Stechele, and C. Passerone. Accelerating and pruning cnns for semantic segmentation on fpga. In *Design Automation Conference*, 2022.

[13] A. Frickenstein, M.-R. Vemparala, N. Fasfous, L. Hauenschild, N.-S. Nagaraja, C. Unger, and W. Stechele. Alf: Autoencoder-based low-rank filter-sharing for efficient convolutional neural networks. In *Design Automation Conference*. IEEE, 2020. `doi:10.1109/DAC18072.2020.9218501`.

[14] M.-R. Vemparala, N. Fasfous, A. Frickenstein, M. A. Moraly, A. Jamal, L. Frickenstein, C. Unger, N.-S. Nagaraja, and W. Stechele. L2pf - learning to prune faster. In *Computer Vision and Image Processing*. Springer, 2021. `doi:10.1007/978-981-16-1103-2_22`.

[15] M.-R. Vemparala, N. Fasfous, L. Frickenstein, A. Frickenstein, A. Singh, D. Salihu, C. Unger, N.-S. Nagaraja, and W. Stechele. Hardware-aware mixed-precision neural networks using in-train quantization. In *British Machine Vision Conference*, 2021. URL: `https://www.bmvc2021.com/`.

[16] N. Fasfous, L. Frickenstein, M. Neumeier, M.-R. Vemparala, A. Frickenstein, E. Valpreda, M. Martina, and W. Stechele. Mind the scaling factors: Resilience analysis of quantized adversarially robust cnns. In *Conference on Design, Automation and Test in Europe*, 2022.

[17] M.-R. Vemparala, N. Fasfous, A. Frickenstein, S. Sarkar, Q. Zhao, S. Kuhn, L. Frickenstein, A. Singh, C. Unger, N.-S. Nagaraja, C. Wressnegger, and W. Stechele. Adversarial robust model compression using in-train pruning. In *Conference on Computer Vision and Pattern Recognition Workshops*. IEEE, 2021. `doi:10.1109/CVPRW53098.2021.00016`.

[18] M.-R. Vemparala, A. Frickenstein, N. Fasfous, L. Frickenstein, Q. Zhao, S. Kuhn, D. Ehrhardt, Y. Wu, C. Unger, N.-S. Nagaraja, and W. Stechele. Breakingbed: Breaking binary and efficient deep neural networks by adversarial attacks. In K. Arai, editor, *Intelligent Systems and Applications*. Springer, 2022. `doi:10.1007/978-3-030-82193-7_10`.

[19] M.-R. Vemparala, A. Singh, A. Mzid, N. Fasfous, A. Frickenstein, F. Mirus, H.-J. Voegel, N. S. Nagaraja, and W. Stechele. Pruning cnns for lidar-based perception in resource constrained environments. In *Intelligent Vehicles Symposium Workshops*. IEEE, 2021. `doi:10.1109/IVWorkshops54471.2021.9669256`.

[20] E. H. Chen, M.-R. Vemparala, N. Fasfous, A. Frickenstein, A. Mzid, N. S. Nagaraja, J. Zeisler, and W. Stechele. Investigating binary neural networks for traffic sign detection and recognition. In *Intelligent Vehicles Symposium*. IEEE, 2021. `doi:10.1109/IV48863.2021.9575557`.

[21] M. Sagi, M. Rapp, H. Khdr, Y. Zhang, N. Fasfous, N. A. Vu Doan, T. Wild, J. Henkel, and A. Herkersdorf. Long short-term memory neural network-based power forecasting of multi-core processors. In *Conference on Design, Automation and Test in Europe*. IEEE, 2021. doi:10.23919/DATE51398.2021.9474028.

[22] M. Sagi, N. A. Vu Doan, N. Fasfous, T. Wild, and A. Herkersdorf. Fine-grained power modeling of multicore processors using ffnns. In *Embedded Computer Systems: Architectures, Modeling, and Simulation: 20th International Conference SAMOS*. Springer, 2020. doi:10.1007/978-3-030-60939-9_13.

[23] M. Sagi, N. A. Vu Doan, N. Fasfous, T. Wild, and A. Herkersdorf. Fine-grained power modeling of multicore processors using ffnns. *International Journal of Parallel Programming*, 2022. doi:10.1007/s10766-022-00730-9.

[24] N. Fasfous, M.-R. Vemparala, A. Frickenstein, L. Frickenstein, M. Badawy, and W. Stechele. Binarycop: Binary neural network-based covid-19 face-mask wear and positioning predictor on edge devices. In *International Parallel and Distributed Processing Symposium Workshops*. IEEE, 2021. doi:10.1109/IPDPSW52791.2021.00024.

[25] N. Fasfous, M.-R. Vemparala, A. Frickenstein, M. Badawy, F. Hundhausen, J. Höfer, N.-S. Nagaraja, C. Unger, H.-J. Vögel, J. Becker, T. Asfour, and W. Stechele. Binarylorax: Low-latency runtime adaptable xnor classifier for semi-autonomous grasping with prosthetic hands. In *International Conference on Robotics and Automation*. IEEE, 2021. doi:10.1109/ICRA48506.2021.9561045.

[26] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet large scale visual recognition challenge. *International Journal of Computer Vision*, 2015. doi:10.1007/s11263-015-0816-y.

[27] A. Krizhevsky. Learning multiple layers of features from tiny images. *University of Toronto*, 2009.

[28] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Conference on Computer Vision and Pattern Recognition*. IEEE, 2016. doi:10.1109/CVPR.2016.90.

[29] L. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *Transactions on Pattern Analysis and Machine Intelligence*, 2018. doi:10.1109/TPAMI.2017.2699184.

[30] J. Redmon and A. Farhadi. YOLO9000: Better, faster, stronger. In *Conference on Computer Vision and Pattern Recognition*. IEEE, 2017. doi:10.1109/CVPR.2017.690.

[31] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han. AMC: Automl for model compression and acceleration on mobile devices. In *European Conference on Computer Vision*. Springer, 2018. doi:10.1007/978-3-030-01234-2_48.

[32] R. Venkatesan, Y. S. Shao, M. Wang, J. Clemons, S. Dai, M. Fojtik, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, Y. Zhang, B. Zimmer, W. J. Dally, J. Emer, S. W. Keckler, and B. Khailany. MAGNet: A modular accelerator generator for neural networks. In *International Conference on Computer-Aided Design*. IEEE, 2019. doi:10.1109/ICCAD45719.2019.8942127.

[33] P. Xu, X. Zhang, C. Hao, Y. Zhao, Y. Zhang, Y. Wang, C. Li, Z. Guan, D. Chen, and Y. Lin. AutoDNNchip: An automated dnn chip predictor and builder for both fpgas and asics. In *International Symposium on Field-Programmable Gate Arrays*. Association for Computing Machinery, 2020. doi:10.1145/3373087.3375306.

[34] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. XNOR-Net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*. Springer, 2016. doi:10.1007/978-3-319-46493-0_32.

[35] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou. DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients. In *arXiv*, 2016. doi:10.48550/ARXIV.1606.06160.

[36] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan. PACT: Parameterized clipping activation for quantized neural networks. In *arXiv*, 2018. doi:10.48550/ARXIV.1805.06085.

[37] Q. Huang, K. Zhou, S. You, and U. Neumann. Learning to prune filters in convolutional neural networks. In *Winter Conference on Applications of Computer Vision*. IEEE, 2018. doi:10.1109/WACV.2018.00083.

[38] G. D. Micheli, A. Sangiovanni-Vincentelli, and P. Antognetti. *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation*. Springer, 1987.

[39] K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 1980. doi:10.1007/bf00344251.

[40] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *arXiv*, 2014. doi:10.48550/ARXIV.1409.1556.

[41] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *arXiv*, 2020. doi:10.48550/ARXIV.2010.11929.

[42] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998. doi:10.1109/5.726791.

[43] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 1989. doi:10.1016/0893-6080(89)90020-8.

[44] D. Hubel and T. Wiesel. *Brain and Visual Perception: The Story of a 25-year Collaboration*. Oxford University Press, 2012. doi:10.1093/acprof:oso/9780195176186.001.0001.

[45] J. Weng, N. Ahuja, and T. Huang. Learning recognition and segmentation of 3-d objects from 2-d images. In *International Conference on Computer Vision*. IEEE, 1993. doi:10.1109/ICCV.1993.378228.

[46] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*. Association for Computing Machinery, 2015.

[47] A. Brock, S. De, S. L. Smith, and K. Simonyan. High-performance large-scale image recognition without normalization. In *arXiv*, 2021. doi:10.48550/ARXIV.2102.06171.

[48] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In *Conference on Computer Vision and Pattern Recognition*. IEEE, 2015. doi:10.1109/CVPR.2015.7298965.

[49] S. Smith, M. Patwary, B. Norick, P. LeGresley, S. Rajbhandari, J. Casper, Z. Liu, S. Prabhumoye, G. Zerveas, V. Korthikanti, E. Zhang, R. Child, R. Y. Aminabadi, J. Bernauer, X. Song, M. Shoeybi, Y. He, M. Houston, S. Tiwary, and B. Catanzaro. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. In *arXiv*, 2022. doi:10.48550/ARXIV.2201.11990.

[50] Y. Lin, F. Lv, S. Zhu, M. Yang, T. Cour, K. Yu, L. Cao, and T. Huang. Large-scale image classification: Fast feature extraction and svm training. In *Computer Vision and Pattern Recognition*. IEEE, 2011. doi:10.1109/CVPR.2011.5995477.

[51] F. Perronnin, Y. Liu, J. Sánchez, and H. Poirier. Large-scale image retrieval with compressed fisher vectors. In *Computer Vision and Pattern Recognition*. IEEE, 2010. doi:10.1109/CVPR.2010.5540009.

[52] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 2017. doi:10.1145/3065386.

[53] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *European Conference on Computer Vision*. Springer, 2014. doi:10.1007/978-3-319-10590-1_53.

[54] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *arXiv*, 2014. doi:10.48550/ARXIV.1409.4842.

[55] J. Hu, L. Shen, and G. Sun. Squeeze-and-excitation networks. In *Conference on Computer Vision and Pattern Recognition*. IEEE, 2018. doi:10.1109/CVPR.2018.00745.

[56] V. Sze, Y. Chen, T. Yang, and J. S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 2017. doi:10.1109/JPROC.2017.2761740.

[57] R. Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. In *arXiv*, 2018. URL: https://arxiv.org/abs/1806.08342, doi:10.48550/ARXIV.1806.08342.

[58] NVIDIA Corporation. *NVDLA TensorRT Documentation*. URL: https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html.

[59] Y. Bengio, N. Léonard, and A. Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. In *arXiv*, 2013. doi:10.48550/ARXIV.1308.3432.

[60] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized neural networks. In *Neural Information Processing Systems*. Curran Associates, Inc., 2016.

[61] S. Darabi, M. Belbahri, M. Courbariaux, and V. P. Nia. Regularized binary network training. In *arXiv*, 2018. doi:10.48550/ARXIV.1812.11800.

[62] X. Lin, C. Zhao, and W. Pan. Towards accurate binary convolutional neural network. In *Neural Information Processing Systems*. Curran Associates Inc., 2017.

[63] Q. Lou, F. Guo, M. Kim, L. Liu, and L. Jiang. AutoQ: Automated kernel-wise neural network quantization. In *International Conference on Learning Representations*, 2020. URL: https://openreview.net/forum?id=rygfnn4twS.

[64] M.-R. Vemparala, A. Frickenstein, and W. Stechele. An efficient fpga accelerator design for optimized cnns using opencl. In *Architecture for Computing Systems*. Springer, 2019. doi:10.1007/978-3-030-18656-2_18.

[65] Y. Umuroglu, D. Conficconi, L. Rasnayake, T. B. Preusser, and M. Själander. Optimizing bit-serial matrix multiplication for reconfigurable computing. *Transactions on Reconfigurable Technology and Systems*, 2019. doi:10.1145/3337929.

[66] S. Sharify, A. D. Lascorz, K. Siu, P. Judd, and A. Moshovos. Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks. In *Proceedings of the 55th Annual Design Automation Conference*. Association for Computing Machinery, 2018. doi:10.1145/3195970.3196072.

[67] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H.-J. Yoo. UNPU: An energy-efficient deep neural network accelerator with fully variable weight bit precision. *Journal of Solid-State Circuits*, 2019. doi:10.1109/JSSC.2018.2865489.

[68] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos. Stripes: Bit-serial deep neural network computing. In *International Symposium on Microarchitecture*. IEEE, 2016. doi:10.1109/MICRO.2016.7783722.

[69] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural networks. In *International Symposium on Computer Architecture*. IEEE Press, 2018. doi:10.1109/ISCA.2018.00069.

[70] J. Hertz, A. Krogh, and R. G. Palmer. *Introduction to the Theory of Neural Computation*. Addison-Wesley Longman Publishing Co., Inc., 1991.

[71] Y. LeCun, J. S. Denker, and S. A. Solla. Optimal brain damage. In *Neural Information Processing Systems*. Morgan-Kaufmann, 1990.

[72] B. Hassibi, D. Stork, and G. Wolff. Optimal brain surgeon and general network pruning. In *International Conference on Neural Networks*. IEEE, 1993. doi:10.1109/ICNN.1993.298572.

[73] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *arXiv*, 2015. doi:10.48550/ARXIV.1510.00149.

[74] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf. Pruning filters for efficient convnets. In *arXiv*, 2016. URL: https://arxiv.org/abs/1608.08710, doi:10.48550/ARXIV.1608.08710.

[75] Y. He, P. Liu, Z. Wang, et al. Filter pruning via geometric median for deep convolutional neural networks acceleration. In *Conference on Computer Vision and Pattern Recognition*. IEEE, 2019. doi:10.1109/CVPR.2019.00447.

[76] Y. He, X. Zhang, and J. Sun. Channel pruning for accelerating very deep neural networks. In *International Conference on Computer Vision*. IEEE, 2017. doi:10.1109/ICCV.2017.155.

[77] T. Yang, Y. Chen, and V. Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *Conference on Computer Vision and Pattern Recognition*. IEEE, 2017. doi:10.1109/CVPR.2017.643.

[78] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. EIE: Efficient inference engine on compressed deep neural network. In *International Symposium on Computer Architecture*. IEEE Press, 2016. doi:10.1109/ISCA.2016.30.

[79] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *International Symposium on Computer Architecture*. Association for Computing Machinery, 2017. doi:10.1145/3079856.3080254.

[80] NVIDIA Corporation. *NVIDIA Ampere GA102 GPU Architecture*, 2020. URL: https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.1.pdf.

[81] M. Tan and Q. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*. PMLR, 2019. URL: `https://proceedings.mlr.press/v97/tan19a.html`.

[82] 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. In *arXiv*, 2017. `doi:10.48550/ARXIV.1704.04861`.

[83] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Conference on Computer Vision and Pattern Recognition*. IEEE, 2018. `doi:10.1109/CVPR.2018.00474`.

[84] T. Wang, K. Wang, H. Cai, J. Lin, Z. Liu, H. Wang, Y. Lin, and S. Han. APQ: Joint search for network architecture, pruning and quantization policy. In *Conference on Computer Vision and Pattern Recognition*. IEEE, 2020. `doi:10.1109/CVPR42600.2020.00215`.

[85] 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 *Conference on Computer Vision and Pattern Recognition*. IEEE, 2019. `doi:10.1109/CVPR.2019.00293`.

[86] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han. Once for all: Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations*, 2020. URL: `https://openreview.net/forum?id=HylxE1HKwS`.

[87] H. Cai, L. Zhu, and S. Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *International Conference on Learning Representations*, 2019. URL: `https://openreview.net/forum?id=HylVB3AqYm`.

[88] Y. Lin, D. Hafdi, K. Wang, Z. Liu, and S. Han. Neural-hardware architecture search. In *Neural Information Processing Systems Workshops*, 2019.

[89] 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 lt;0.5mb model size. In *arXiv*, 2016. `doi:10.48550/ARXIV.1602.07360`.

[90] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. In *arXiv*, 2020. `doi:10.48550/ARXIV.2005.14165`.

[91] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. In *arXiv*, 2014. `doi:10.48550/ARXIV.1412.6572`.

[92] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations*, 2018. URL: `https://openreview.net/forum?id=rJzIBfZAb`.

[93] E. Wong, L. Rice, and J. Z. Kolter. Fast is better than free: Revisiting adversarial training. In *International Conference on Learning Representations*, 2020. URL: https://openreview.net/forum?id=BJx040EFvH.

[94] M. Alwani, H. Chen, M. Ferdman, and P. Milder. Fused-layer cnn accelerators. In *International Symposium on Microarchitecture*. IEEE, 2016. doi:10.1109/MICRO.2016.7783725.

[95] NVIDIA Corporation. *NVIDIA Tesla V100 GPU Architecture*, 2017. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[96] NVIDIA Corporation. *NVDLA Open Source Project - Primer*. URL: http://nvdla.org/primer.html.

[97] NVIDIA Corporation. *NVIDIA Turing GPU Architecture*, 2017. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[98] TensorFlow. *TensorFlow Lite*. URL: https://www.tensorflow.org/lite/guide.

[99] Intel Corporation. *Deep learning with Intel AVX-512 and Intel DL BOOST*. URL: https://www.intel.com/content/www/us/en/developer/articles/guide/deep-learning-with-avx512-and-dl-boost.html.

[100] 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 *International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 2020. doi:10.1145/3373376.3378514.

[101] Q. Huang, M. Kang, G. Dinh, T. Norell, A. Kalaiah, J. Demmel, J. Wawrzynek, and Y. S. Shao. CoSA: Scheduling by constrained optimization for spatial accelerators. In *arXiv*, 2021. doi:10.48550/ARXIV.2105.01898.

[102] S.-C. Kao and T. Krishna. GAMMA: Automating the hw mapping of dnn models on accelerators via genetic algorithm. In *International Conference on Computer-Aided Design*. Association for Computing Machinery, 2020. doi:10.1145/3400302.3415639.

[103] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers. FINN: A framework for fast, scalable binarized neural network inference. In *International Symposium on Field-Programmable Gate Arrays*. Association for Computing Machinery, 2017. doi:10.1145/3020078.3021744.

[104] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'brien, Y. Umuroglu, M. Leeser, and K. Vissers. Finn-r: An end-to-end deep-learning framework for fast exploration of

quantized neural networks. *Transactions on Reconfigurable Technology and Systems*, 2018. doi:10.1145/3242897.

[105] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G.-J. Nam, B. Taba, M. Beakes, B. Brezzo, J. B. Kuang, R. Manohar, W. P. Risk, B. Jackson, and D. S. Modha. Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2015. doi:10.1109/TCAD.2015.2474396.

[106] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C.-K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y.-H. Weng, A. Wild, Y. Yang, and H. Wang. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 2018. doi:10.1109/MM.2018.112130359.

[107] H. Kwon, A. Samajdar, and T. Krishna. MAERI: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. *ACM SIGPLAN Notices*, 2018. doi:10.1145/3296957.3173176.

[108] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis. TETRIS: Scalable and efficient neural network acceleration with 3d memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 2017. doi:10.1145/3037697.3037702.

[109] M. Gao, X. Yang, J. Pu, M. Horowitz, and C. Kozyrakis. TANGRAM: Optimized coarse-grained dataflow for scalable nn accelerators. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 2019. doi:10.1145/3297858.3304014.

[110] J. Li, G. Yan, W. Lu, S. Jiang, S. Gong, J. Wu, and X. Li. SmartShuttle: Optimizing off-chip memory accesses for deep learning accelerators. In *Conference on Design, Automation and Test in Europe*. IEEE, 2018. doi:10.23919/DATE.2018.8342033.

[111] D. D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner. *Embedded system design modeling, synthesis and verification*. Springer, 2009.

[112] M. Courbariaux, Y. Bengio, and J.-P. David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Neural Information Processing Systems*. Curran Associates, Inc., 2015.

[113] R. Andri, L. Cavigelli, D. Rossi, and L. Benini. YodaNN: An architecture for ultralow power binary-weight cnn acceleration. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018. doi:10.1109/TCAD.2017.2682138.

[114] K. Ando, K. Ueyoshi, K. Orimo, H. Yonekawa, S. Sato, H. Nakahara, S. Takamaeda-Yamazaki, M. Ikebe, T. Asai, T. Kuroda, and M. Motomura. BRein memory: A single-chip binary/ternary reconfigurable in-memory deep neural network accelerator

achieving 1.4 tops at 0.6 w. *Journal of Solid-State Circuits*, 2018. doi:10.1109/JSSC.2017.2778702.

[115] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *International Symposium on Field-Programmable Gate Arrays*. Association for Computing Machinery, 2017. doi:10.1145/3020078.3021741.

[116] L. Yang, Z. He, and D. Fan. A fully onchip binarized convolutional neural network fpga impelmentation with accurate inference. In *International Symposium on Low Power Electronics and Design*. Association for Computing Machinery, 2018. doi:10.1145/3218603.3218615.

[117] S. Liang, S. Yin, L. Liu, W. Luk, and S. Wei. Fp-bnn. *Neurocomputing*, 2018. doi:10.1016/j.neucom.2017.09.046.

[118] D. Nguyen, D. Kim, and J. Lee. Double MAC: Doubling the performance of convolutional neural networks on modern fpgas. In *Conference on Design, Automation and Test in Europe*. IEEE, 2017. doi:10.23919/DATE.2017.7927113.

[119] E. Nurvitadhi, D. Sheffield, Jaewoong Sim, A. Mishra, G. Venkatesh, and D. Marr. Accelerating binarized neural networks: Comparison of fpga, cpu, gpu, and asic. In *International Conference on Field-Programmable Technology*. IEEE, 2016. doi:10.1109/FPT.2016.7929192.

[120] Xilinx, Inc. *Versal: The First Adaptive Compute Acceleration Platform (ACAP)*, 2018. v1.0.

[121] W. Tang, G. Hua, and L. Wang. How to train a compact binary neural network with high accuracy? In *Conference on Artificial Intelligence*. AAAI Press, 2017.

[122] U. Zahid, G. Gambardella, N. J. Fraser, M. Blott, and K. Vissers. FAT: Training neural networks for reliable inference under hardware faults. In *International Test Conference*. IEEE, 2020. doi:10.1109/ITC44778.2020.9325249.

[123] Z. He, A. S. Rakin, J. Li, C. Chakrabarti, and D. Fan. Defending and harnessing the bit-flip based adversarial weight attack. In *Conference on Computer Vision and Pattern Recognition*. IEEE, 2020. doi:10.1109/CVPR42600.2020.01410.

[124] J. Lin, C. Gan, and S. Han. Defensive quantization: When efficiency meets robustness. In *International Conference on Learning Representations*, 2019. URL: https://openreview.net/forum?id=ryetZ20ctX.

[125] Y. He, P. Balaprakash, and Y. Li. FIdelity: Efficient resilience analysis framework for deep learning accelerators. In *International Symposium on Microarchitecture*. IEEE, 2020. doi:10.1109/MICRO50266.2020.00033.

[126] L.-H. Hoang, M. A. Hanif, and M. Shafique. FT-ClipAct: Resilience analysis of deep neural networks and improving their fault tolerance using clipped activation. In *Conference on Design, Automation and Test in Europe*. EDA Consortium, 2020.

[127] A. S. Rakin, Z. He, and D. Fan. Bit-flip attack: Crushing neural network with progressive bit search. In *International Conference on Computer Vision*. IEEE, 2019. doi:10.1109/ICCV.2019.00130.

[128] J. Li, A. S. Rakin, Y. Xiong, L. Chang, Z. He, D. Fan, and C. Chakrabarti. Defending bit-flip attack through dnn weight reconstruction. In *Design Automation Conference*. IEEE, 2020. doi:10.1109/DAC18072.2020.9218665.

[129] F. Liao, M. Liang, Y. Dong, T. Pang, X. Hu, and J. Zhu. Defense against adversarial attacks using high-level representation guided denoiser. In *Conference on Computer Vision and Pattern Recognition*. IEEE, 2018. doi:10.1109/CVPR.2018.00191.

[130] B. Calli, A. Singh, A. Walsman, S. Srinivasa, P. Abbeel, and A. M. Dollar. The YCB object and model set: Towards common benchmarks for manipulation research. In *International Conference on Advanced Robotics*. IEEE, 2015. doi:10.1109/ICAR.2015.7251504.

[131] F. Hundhausen, D. Megerle, and T. Asfour. Resource-aware object classification and segmentation for semi-autonomous grasping with prosthetic hands. In *International Conference on Humanoid Robots*. IEEE, 2019. doi:10.1109/Humanoids43949.2019.9035054.

[132] T. R. Farrell and R. F. Weir. The optimal controller delay for myoelectric prostheses. *Transactions on Neural Systems and Rehabilitation Engineering*, 2007. doi:10.1109/TNSRE.2007.891391.

[133] S. Došen, C. Cipriani, M. Kostić, M. Controzzi, M. C. Carrozza, and D. B. Popović. Cognitive vision system for control of dexterous prosthetic hands: experimental evaluation. *Journal of neuroengineering and rehabilitation*, 2010. doi:10.1186/1743-0003-7-42.

[134] M. Markovic, S. Dosen, C. Cipriani, D. Popovic, and D. Farina. Stereovision and augmented reality for closed-loop control of grasping in hand prostheses. *Journal of neural engineering*, 2014. doi:10.1088/1741-2560/11/4/046001.

[135] G. Ghazaei, A. Alameer, P. Degenaar, G. Morgan, and K. Nazarpour. An exploratory study on the use of convolutional neural networks for object grasp classification. In *International Conference on Intelligent Signal Processing*. IEEE, 2015. doi:10.1049/cp.2015.1760.

[136] J. DeGol, A. Akhtar, B. Manja, and T. Bretl. Automatic grasp selection using a camera in a hand prosthesis. In *International Conference of the IEEE Engineering in Medicine and Biology Society*. IEEE, 2016. doi:10.1109/EMBC.2016.7590732.

[137] P. Weiner, J. Starke, F. Hundhausen, J. Beil, and T. Asfour. The KIT prosthetic hand: design and control. In *International Conference on Intelligent Robots and Systems*. IEEE, 2018. `doi:10.1109/IROS.2018.8593851`.

[138] M. Esponda and T. M. Howard. Adaptive grasp control through multi-modal interactions for assistive prosthetic devices. In *arXiv*, 2018. `doi:10.48550/ARXIV.1810.07899`.

[139] Y. He, R. Shima, O. Fukuda, N. Bu, N. Yamaguchi, and H. Okumura. Development of distributed control system for vision-based myoelectric prosthetic hand. *IEEE Access*, 2019. `doi:10.1109/ACCESS.2019.2911968`.

[140] C. Shi, D. Yang, J. Zhao, and H. Liu. Computer vision-based grasp pattern recognition with application to myoelectric control of dexterous hand prosthesis. *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, 2020. `doi:10.1109/TNSRE.2020.3007625`.

[141] N. Bu, Y. Bandou, O. Fukuda, H. Okumura, and K. Arai. A semi-automatic control method for myoelectric prosthetic hand based on image information of objects. In *International Conference on Intelligent Informatics and Biomedical Sciences*. IEEE, 2017. `doi:10.1109/ICIIBMS.2017.8279702`.

[142] A. Frickenstein, M.-R. Vemparala, J. Mayr, N.-S. Nagaraja, C. Unger, F. Tombari, and W. Stechele. Binary DAD-Net: Binarized driveable area detection network for autonomous driving. In *International Conference on Robotics and Automation*. IEEE, 2020. `doi:10.1109/ICRA40945.2020.9197119`.

[143] B. Zhuang, C. Shen, M. Tan, L. Liu, and I. Reid. Structured binary neural networks for accurate image classification and semantic segmentation. In *Conference on Computer Vision and Pattern Recognition*. IEEE, 2019. `doi:10.1109/CVPR.2019.00050`.

[144] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A large-scale hierarchical image database. In *Conference on Computer Vision and Pattern Recognition*. IEEE, 2009. `doi:10.1109/CVPR.2009.5206848`.

[145] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel. The german traffic sign recognition benchmark: A multi-class classification competition. In *International Joint Conference on Neural Networks*. IEEE, 2011. `doi:10.1109/IJCNN.2011.6033395`.

[146] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng. Reading digits in natural images with unsupervised feature learning. In *Neural Information Processing Systems Workshops*, 2011.

[147] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele. The cityscapes dataset for semantic urban scene understanding. In *Conference on Computer Vision and Pattern Recognition*. IEEE, 2016. `doi:10.1109/CVPR.2016.350`.

[148] F. Hundhausen, J. Starke, and T. Asfour. A soft humanoid hand with in-finger visual perception. In *arXiv*, 2020. doi:10.48550/ARXIV.2006.03537.

[149] L. Wang, Z. Q. Lin, and A. Wong. Covid-net: A tailored deep convolutional neural network design for detection of covid-19 cases from chest x-ray images. *Scientific Reports*, 2020. doi:10.1038/s41598-020-76550-z.

[150] A. I. Khan, J. L. Shah, and M. M. Bhat. Coronet: A deep neural network for detection and diagnosis of covid-19 from chest x-ray images. *Computer Methods and Programs in Biomedicine*, 2020. doi:10.1016/j.cmpb.2020.105581.

[151] When and how to use masks. URL: https://www.who.int/emergencies/diseases/novel-coronavirus-2019/advice-for-public/when-and-how-to-use-masks.

[152] N. O'Mahony, S. Campbell, A. Carvalho, S. Harapanahalli, G. V. Hernandez, L. Krpalkova, D. Riordan, and J. Walsh. Deep learning vs. traditional computer vision. In *Advances in Computer Vision*. Springer, 2020. doi:10.1007/978-3-030-17795-9_10.

[153] A. Cabani, K. Hammoudi, H. Benhabiles, and M. Melkemi. Maskedface-net – a dataset of correctly/incorrectly masked face images in the context of covid-19. *Smart Health*, 2020. doi:10.1016/j.smhl.2020.100144.

[154] T. Mitze, R. Kosfeld, J. Rode, and K. Wälde. Face masks considerably reduce covid-19 cases in germany. *Proceedings of the National Academy of Sciences*, 2020. doi:10.1073/pnas.2015954117.

[155] S. Ge, J. Li, Q. Ye, and Z. Luo. Detecting masked faces in the wild with lle-cnns. In *Conference on Computer Vision and Pattern Recognition*. IEEE, 2017. doi:10.1109/CVPR.2017.53.

[156] Z. Wang, G. Wang, B. Huang, Z. Xiong, Q. Hong, H. Wu, P. Yi, K. Jiang, N. Wang, Y. Pei, H. Chen, Y. Miao, Z. Huang, and J. Liang. Masked face recognition dataset and application. In *arXiv*, 2020. doi:10.48550/ARXIV.2003.09093.

[157] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba. Learning deep features for discriminative localization. In *Conference on Computer Vision and Pattern Recognition*. IEEE, 2016. doi:10.1109/CVPR.2016.319.

[158] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra. Grad-CAM: Visual explanations from deep networks via gradient-based localization. In *International Conference on Computer Vision*. IEEE, 2017. doi:10.1109/ICCV.2017.74.

[159] A. Kulkarni, A. Vishwanath, and C. Shah. Implementing a real-time, ai-based, face mask detector application for covid-19, 2021. URL: https://developer.nvidia.com/blog/implementing-a-real-time-ai-based-face-mask-detector-application-for-covid-19/.

[160] T. Agrawal, K. Imran, M. Figus, and C. Kirkpatrick. Automatically detecting personal protective equipment on persons in images using amazon rekognition, 2020. URL: `https://aws.amazon.com/blogs/machine-learning/automatically-detecting-personal-protective-equipment-on-persons-in-images-using-amazon-rekognition/`.

[161] Z. Wang, P. Wang, P. C. Louis, L. E. Wheless, and Y. Huo. Wearmask: Fast in-browser face mask detection with serverless edge computing for covid-19. In *arXiv*, 2021. `doi:10.48550/ARXIV.2101.00784`.

[162] K. Hammoudi, A. Cabani, H. Benhabiles, and M. Melkemi. Validating the correct wearing of protection mask by taking a selfie: Design of a mobile application "checkyourmask" to limit the spread of covid-19. *Computer Modeling in Engineering & Sciences*, 2020. `doi:10.32604/cmes.2020.011663`.

[163] A. Anwar and A. Raychowdhury. Masked face recognition for secure authentication. In *arXiv*, 2020. `doi:10.48550/ARXIV.2008.11104`.

[164] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *Journal of Machine Learning Research*, 2017. `doi:10.5555/3122009.3242044`.

[165] Z. Dong, Z. Yao, A. Gholami, M. Mahoney, and K. Keutzer. HAWQ: Hessian aware quantization of neural networks with mixed-precision. In *International Conference on Computer Vision*. IEEE, 2019. `doi:10.1109/ICCV.2019.00038`.

[166] B. Wu, Y. Wang, P. Zhang, Y. Tian, P. Vajda, and K. Keutzer. Mixed precision quantization of convnets via differentiable neural architecture search. In *arXiv*, 2018. `doi:10.48550/ARXIV.1812.00090`.

[167] W. Jiang, L. Yang, E. H.-M. Sha, Q. Zhuge, S. Gu, S. Dasgupta, Y. Shi, and J. Hu. Hardware/software co-exploration of neural architectures. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020. `doi:10.1109/TCAD.2020.2986127`.

[168] M. S. Abdelfattah, L. Dudziak, T. Chau, R. Lee, H. Kim, and N. D. Lane. Best of both worlds: Automl codesign of a cnn and its hardware accelerator. In *Design Automation Conference*. Association for Computing Machinery, 2020. `doi:10.5555/3437539.3437731`.

[169] P. Meloni, D. Loi, G. Deriu, A. D. Pimentel, D. Sapra, B. Moser, N. Shepeleva, F. Conti, L. Benini, O. Ripolles, D. Solans, M. Pintor, B. Biggio, T. Stefanov, S. Minakova, N. Fragoulis, I. Theodorakopoulos, M. Masin, and F. Palumbo. ALOHA: An architectural-aware framework for deep learning at the edge. In *Workshop on INTelligent Embedded Systems Architectures and Applications*. Association for Computing Machinery, 2018. `doi:10.1145/3285017.3285019`.

[170] X. Dai, P. Zhang, B. Wu, H. Yin, F. Sun, Y. Wang, M. Dukhan, Y. Hu, Y. Wu, Y. Jia, P. Vajda, M. Uyttendaele, and N. K. Jha. Chamnet: Towards efficient network design through platform-aware model adaptation. In *Conference on Computer Vision and Pattern Recognition*. IEEE, 2019. `doi:10.1109/CVPR.2019.01166`.

[171] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multi-objective genetic algorithm: NSGA-ii. *Transactions on Evolutionary Computation*, 2002. `doi:10.1109/4235.996017`.

[172] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo. Optimizing the convolution operation to accelerate deep neural networks on fpga. *Transactions on Very Large Scale Integration (VLSI) Systems*, 2018. `doi:10.1109/TVLSI.2018.2815603`.

[173] M. Horowitz. 1.1 computing's energy problem (and what we can do about it). In *International Solid-State Circuits Conference Digest of Technical Papers*. IEEE, 2014. `doi:10.1109/ISSCC.2014.6757323`.

[174] A. Frickenstein, M. Rohit Vemparala, C. Unger, F. Ayar, and W. Stechele. DSC: Dense-sparse convolution for vectorized inference of convolutional neural networks. In *Conference on Computer Vision and Pattern Recognition Workshops*. IEEE, 2019. `doi:10.1109/CVPRW.2019.00175`.

[175] L.-C. Chen, G. Papandreou, F. Schroff, and H. Adam. Rethinking atrous convolution for semantic image segmentation. In *arXiv*, 2017. `doi:10.48550/ARXIV.1706.05587`.

[176] W. Jiang, L. Yang, S. Dasgupta, J. Hu, and Y. Shi. Standing on the shoulders of giants: Hardware and neural architecture co-search with hot start. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020. `doi:10.1109/TCAD.2020.3012863`.

[177] L. Mei, P. Houshmand, V. Jain, S. Giraldo, and M. Verhelst. Zigzag: Enlarging joint architecture-mapping design space exploration for dnn accelerators. *Transactions on Computers*, 2021. `doi:10.1109/TC.2021.3059962`.

[178] M. Bouvier, A. Valentian, T. Mesquida, F. Rummens, M. Reyboz, E. Vianello, and E. Beigne. Spiking neural networks hardware implementations and challenges: A survey. *Journal on Emerging Technologies in Computing Systems*, 2019. `doi:10.1145/3304103`.

[179] C. D. Schuman, T. E. Potok, R. M. Patton, J. D. Birdwell, M. E. Dean, G. S. Rose, and J. S. Plank. A survey of neuromorphic computing and neural networks in hardware. In *arXiv*, 2017. `doi:10.48550/ARXIV.1705.06963`.

A Appendix

Table A.1: Network architectures and hardware dimensioning used in Binary-LoRAX and BinaryCoP. Pool, ReLU and batch normalization layers not shown. FC_3 | [25/4] indicates YCB (25 classes) or MaskedFace-Net (4 classes).

Network	(v)-CNV	m-CNV	μ-CNV	n-CNV
Arch. $L \mid [C_i, C_o]$ $K = 3 \; \forall$ Conv	Conv_1_1 | [3, 64] Conv_1_2 | [64, 64] Conv_2_1 | [64, 128] Conv_2_2 | [128, 128] Conv_3_1 | [128, 256] Conv_3_2 | [256, 256] FC_1 | [512] FC_2 | [512] FC_3 | [25/4]	Conv_1_1 | [3, 32] Conv_1_2 | [32, 32] Conv_2_1 | [32, 64] Conv_2_2 | [64, 64] Conv_3_1 | [64, 128] Conv_3_2 | [128, 128] FC_1 | [256] FC_2 | [256] FC_3 | [25/4]	Conv_1_1 | [3,16] Conv_1_2 | [16, 16] Conv_2_1 | [16, 32] Conv_2_2 | [32, 32] Conv_3_1 | [32, 64] FC_1 | [128] FC_2 | [25/4]	Conv_1_1 | [3, 16] Conv_1_2 | [16, 16] Conv_2_1 | [16, 32] Conv_2_2 | [32, 32] Conv_3_1 | [32, 64] Conv_3_2 | [64, 64] FC_1 | [128] FC_2 | [128] FC_3 | [4]
PE Count **SIMD lanes**	16, 32, 16, 16, 4, 1, 1, 1, 4 3, 32, 32, 32, 32, 32, 4, 8, 1		4, 4, 4, 4, 1, 1, 1 3, 16, 16, 32, 32, 16, 1	16, 16, 16, 16, 4, 1, 1, 1, 1 3, 16, 16, 32, 32, 32, 4, 8, 1
YCB-Objects	`mug, banana, toy_airplane, chips_can, tomato_soup_can, windex_bottle, apple, scissors, sugar_box, master_chef_can, mustard_bottle, orange, pudding_box, lemon, plate, pitcher_base, potted_meat_can, mini_soccer_ball, gelatin_box, large_clamp, power_drill, tennis_ball, cracker_box, adjustable_wrench, knife`			
MaskedFace-Net	`Correctly Masked, Nose Exposed, Nose + Mouth Exposed, Chin Exposed`			

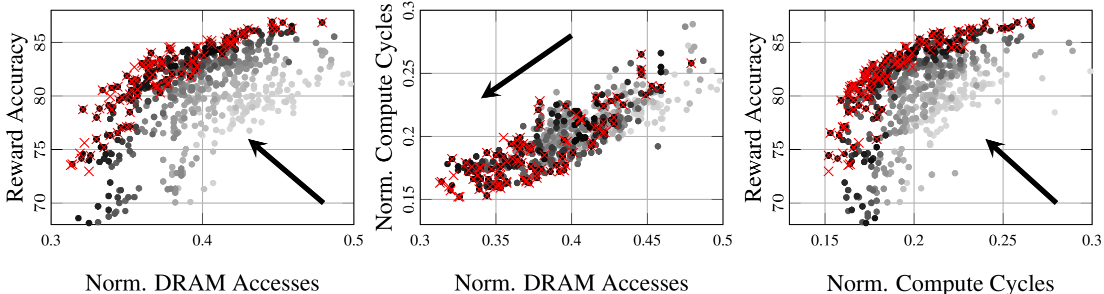


Figure A.1: QSS: 2-D projections of a 3-D Pareto-front for optimal quantization with respect to accuracy, compute cycles, and DRAM accesses on HW3. Compute cycles and DRAM accesses are normalized to an 8-bit execution on HW3. "Reward Accuracy" is with minimal fine-tuning (not fully trained). **It is recommended to view this figure in color.**

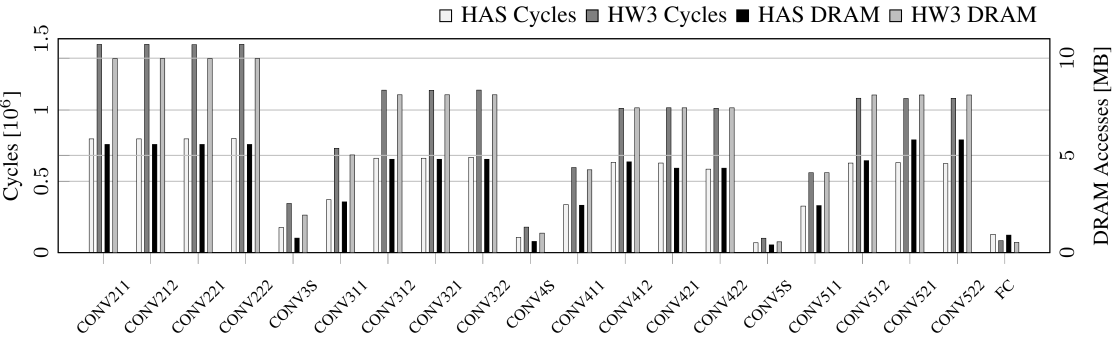


Figure A.2: Comparison of a HAS solution ($\mathrm{D}_m, \mathrm{D}_n, \mathrm{D}_k$ = 8, 14, 96) found for ResNet18-ImageNet 4-bit against the larger standard symmetric hardware configuration HW3. The CONV1 layer follows the same trend but is not shown to maintain plot scale.