Politecnico di Torino

ScuDo

Scuola di Dottorato ~ Doctoral School

WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation

Doctoral Program in Electrical, Electronics and Communications Engineering (38^{th}cycle)

NN2FPGA

Optimizing CNN Inference on FPGAs With Binary Integer Programming

By

Teodoro Urso

******

Supervisor(s):

Prof. Luciano Lavagno, Supervisor

Prof. Mario R. Casu, Co-Supervisor

Doctoral Examination Committee:

Prof. Alex Yakovlev , Newcastle University

Prof. Ioannis Sourdis, Chalmers University of Technology

Prof. Jordi Cortadella, Universitat Politècnica de Catalunya

Prof. Mihai T. Lazarescu, Politecnico di Torino

Prof. Tiziano Villa, Università di Verona

Politecnico di Torino

2025

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Teodoro Urso
2025

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

I would like to dedicate this thesis to my loving parents and my cherished sister

Abstract

The increasing complexity of Convolutional Neural Networks (CNNs) and their deployment in resource-constrained environments have emphasized the importance of efficient hardware accelerators. Field Programmable Gate Arrays (FPGAs) provide an attractive balance of performance, energy efficiency, and flexibility for deep learning inference. However, implementing modern CNNs, particularly those with architectural features such as skip connections, on FPGAs presents considerable challenges related to performance, latency and resource utilization.

This thesis presents nn2FPGA, a novel high-level synthesis based framework that automatically compiles quantized CNN models into highly efficient static dataflow accelerators targeting embedded FPGAs. The framework performs graph-level optimizations for managing skip connections, provides a templated C++ library supporting various CNN layers and data types, and a binary integer programming (BIP) based design space exploration strategy that balances throughput and hardware resource usage.

A key contribution is the introduction of buffering-aware optimizations that minimize on-chip memory requirements for residual blocks through techniques such as temporal reuse and loop merging. These enable efficient fusion of operations across branches while preserving dataflow parallelism. Furthermore, the BIP formulation enables optimal selection of loop unroll factors across layers, maximizing throughput under DSP and memory constraints and subsequently minimizing resource usage for fixed performance targets.

The proposed framework is validated on several state-of-the-art CNN models, including ResNet8, ResNet20, and MobileNetV2, using datasets such as CIFAR-10 and ImageNet, and deployed on various Xilinx/AMD FPGA platforms. Experimental results demonstrate throughput improvements of up to 7× compared to other HLS-

based tools and even a 10% speedup over a handcrafted RTL design, confirming the viability of HLS-based CNN acceleration when guided by rigorous optimization.

This thesis proposes nn2FPGA as a scalable and robust methodology for hardware-aware CNN compilation and provides the basis for future research to support more neural network topologies and application domains, including object detection and semantic segmentation.

Contents

List of Figures x

List of Tables xii

1 Introduction 1

1.1 Motivation: Efficient CNN inference for edge devices 1

1.2 Problem Statement: Mapping modern CNNs onto FPGAs 3

1.3 Research Objectives 4

2 Background 5

2.1 Convolutional Neural Networks: An Overview 5

2.2 FPGA Fundamentals 6

2.3 CNN Acceleration on FPGAs 7

2.3.1 Accelerator Paradigms: Overlay vs. Dataflow 8

2.3.2 High-Level Synthesis for FPGA-based CNNs 9

2.4 Related Work in FPGA-based CNN Frameworks 11

3 The nn2FPGA Compiler Architecture 13

3.1 Introduction to the Compiler Framework 13

3.2 Quantization and Model Preparation 14

3.2.1 Quantization Methods 14

3.2.2 Brevitas and QONNX Generation 16
3.2.3 Batch Normalization Folding 16
3.3 Compiler Flow and Intermediate Representations 18
3.3.1 From QONNX to HLS C++ Code 18
3.3.2 RTL Generation with Vitis HLS 18
3.3.3 Vivado Integration and Bitstream Generation 19
3.4 Accelerator Architecture 20
3.4.1 Static Dataflow Model 20
3.4.2 Task Categories 20
3.4.3 Streaming and DMA Execution Model 21
3.4.4 Hierarchical Pipelining 21
3.5 Convolution Computation Engine 22
3.5.1 Parallelization Strategies 22
3.5.2 DSP Optimizations 25
3.6 Window Buffer Organization 28
3.6.1 Line Buffer Design 28
3.6.2 Partitioning Strategies 29
3.6.3 Impact of Stride and Parallelism 30
3.6.4 Optimization Techniques 31

4 Resource Allocation and Design Space Exploration 34
4.1 Problem Formulation 35
4.2 Parallelization Strategies 37
4.2.1 Output Width Parallelism 37
4.2.2 Output Channel Parallelism 38
4.2.3 Input Channel Parallelism 39
4.2.4 Combined Parallelization 39

4.3 Binary Integer Programming Model 40
4.3.1 Decision Variables 40
4.3.2 Layer Latency and Resource Usage 41
4.3.3 Objective Function 41
4.3.4 Resource Constraints 42
4.3.5 Complete BIP Formulation 42
4.4 Optimization Flow 43
4.4.1 Enumeration of Feasible Configurations 43
4.4.2 Throughput Maximization 44
4.4.3 Resource Minimization for Non-Critical Layers 44
4.4.4 Integration with the Compiler Toolchain 44
4.4.5 Summary of the Flow 45
4.5 Discussion 45

5 Experimental Evaluation 47
5.1 Setup 47
5.2 Results and Comparisons 48
5.2.1 Object Detection with YOLOv5n 50
5.3 Summary 52

6 Conclusions and Future Works 53
6.1 Summary of Contributions 53
6.2 Experimental Findings 54
6.3 Limitations 54
6.4 Future Work 55
6.4.1 Mixed-Precision and Adaptive Quantization 55
6.4.2 Model-Generalization Beyond CNNs 55

6.5 Final Remarks 56

References **57**

List of Figures

2.1 Vitis HLS workflow (from [1]) 11

3.1 Implementation flow 19

3.2 Accelerator architecture with Direct Memory Access (DMA) blocks for memory transfers (blue boxes) and concurrent tasks communicating through data streams. 22

3.3 Convolution architectures. Activations follow horizontal lines, while accumulators follow vertical ones. (a) depicts a 3×3 standard convolution. (b) depicts a 3×3 depthwise convolution with two overlapping windows convolved simultaneously. 25

3.4 DSP chain propagating the accumulator. 26

3.5 Visual representation of how a 2×2 window is spread in the flattened tensor and then partitioned to form the line buffer. The activation tensor (leftmost image) is streamed in depth-first order, resulting in the flattened representation stored in the line buffer (center image). The line buffer is partitioned so that a whole input window is simultaneously available at its read points (rightmost image). . . . 28

3.6 Buffer partitioning in case of a 3×3 window, with $ow^{par} = 1$. . . . 30

3.7 Buffer partitioning in case of a 3×3 window, with $ow^{par} = 2$. The separate flows of activations belonging to the first and second windows are represented in red and blue. 31

3.8 Complete graph optimization of residual blocks. (a) and (b) depict the residual block without and with the downsampling convolution in the skip connection. 32

List of Tables

3.1 Comparison of integer PTQ using power-of-two versus floating point scaling factors on ImageNet[2]. 17
3.2 Symbol definitions. 23

4.1 Binary integer programming solver execution time. 44

5.1 Resources of the boards used for the experiments. 47
5.2 Performance comparison with SOTA accelerators. Power was measured using sensors on the power rails and is representative of the entire board consumption. Throughput was evaluated with a batch size of 400 for ImageNet and 1000 for CIFAR10. 49
5.3 Resource utilization comparison. Reported data are collected from the Vivado report after place and route. 50
5.4 On-chip activation resource reduction achieved through various graph optimizations. ResNets are generated for KRIA KV260, while MobileNetV2 for ZCU102. All kernels are synthesized at 300MHz in out-of-context mode, with results obtained from the synthesis report of Vivado. 50
5.5 Resource utilization of the YOLOv5n accelerator on ZCU102 (8-bit quantization). 52

List of Acronyms

AI Artificial Intelligence

ASIC Application-Specific Integrated Circuit

AXI Advanced eXtensible Interface

BIP Binary Integer Programming

BN Batch Normalization

BRAM Block RAM

CLB Configurable Logic Block

CNN Convolutional Neural Network

CPU Central Processing Unit

DDR Double Data Rate

DMA Direct Memory Access

DNN Deep Neural Network

DRAM Dynamic RAM

DSE Design Space Exploration

DSP Digital Signal Processing Unit

EDA Electronic Design Automation

FIFO First-In First-Out

FPGA Field Programmable Gate Array

FPS Frame Per Second

GPU Graphics Processing Unit

HDL Hardware Description Language

HBM High Bandwidth Memory

HLS High Level Synthesis

II Initiation Interval

IP Intellectual Property

LUT Look-Up Tables

MAC Multiply and Accumulate

MLIR Multi-Level Intermediate Representation

NMS Non-Maximum Suppression

OCM On-Chip Memory

ONNX Open Neural Network Exchange

PE Processing Element

PL Programmable Logic

PS Processing System

PTQ Post-Training Quantization

QAT Quantization-Aware Training

QONNX Quantized ONNX

RTL Register Transfer Level

SGD Stochastic Gradient Descent

SOC System on Chip

TPU Tensor Processing Unit

VHDL VHSIC Hardware Description Language

Chapter 1

Introduction

1.1 Motivation: Efficient CNN inference for edge devices

In recent years, CNNs have become fundamental for a wide range of applications in modern computer vision, improving the performance in tasks such as image classification, object detection, and semantic segmentation [3] [4]. Their effectiveness comes from increasingly deep and complex architectures that exploit design innovations, including skip connections[5] and depthwise separable convolutions [6], to reach high levels of precision.

The deployment of these models on edge devices, such as drones, autonomous sensors, and IoT nodes, remains challenging, as such platforms operate under tight constraints in terms of computation, memory, and energy consumption [7] [8]. Conventional deployment strategies based on cloud inference or power-hungry Graphics Processing Units (GPUs), are generally unsuitable in these scenarios. In contrast, FPGAs offer a compelling alternative: their fine-grained parallelism capacity, their ability to achieve low-latency inference in real time, and their adaptability to application-specific requirements come with significantly reduced power consumption compared to general-purpose accelerators [9].

Beyond these aspects, several architectural features make FPGAs particularly appropriate for the deployment of CNNs on constrained platforms. Their spatially reconfigurable pipelines allow convolutional operators and activation flows to be

implemented directly in hardware. This property enables predictable latency and sustained throughput, which are essential in real-time scenarios. In addition, the presence of large on-chip memories mitigates external reliance Dynamic RAM (DRAM), reducing one of the main contributors to energy consumption in deep learning workloads. The ability to customize arithmetic precision and degree of parallelism on a per-layer basis also provides an advantage when handling irregular structures such as depthwise convolutions or skip connections, which often lead to imbalanced utilization on fixed architectures. Taken together, these characteristics reinforce the suitability of FPGAs for edge deployments where power, memory, and latency constraints dominate system design.

Despite their promise, multiple obstacles limit the efficient use of FPGAs for CNN inference. The structural complexity of modern architectures, particularly those that employ skip connections, presents irregular data dependencies and demands on-chip buffering [10]. Standard high-level synthesis flows and generic compilers typically struggle to obtain optimal dataflows, resulting in inefficient allocation of resources. At the same time, hand-crafted Register Transfer Level (RTL) implementations, although more efficient, are prohibitively time-consuming, error-prone, and impractical to maintain across different models and devices. Furthermore, the problem of Design Space Exploration (DSE) deciding how to distribute scarce hardware resources across the many layers of a CNN remains computationally complex and strongly impacts performance. Suboptimal allocations often introduce bottlenecks in critical layers, which limit the throughput of the entire pipeline.

These challenges motivate the thesis work, which aims to design a framework that automates and optimizes the deployment of CNNs on embedded FPGAs. The main goal is to achieve maximum throughput within the resource budgets of edge platforms; this requires a co-design strategy that integrates compiler-level optimizations, hardware-aware graph transformations, and mathematically grounded approaches to resource allocation.

By combining an analysis on neural architectures, FPGA capabilities, and automated optimization flows, this thesis aims to improve the efficiency of edge Artificial Intelligence (AI) and broaden the accessibility of high-performance inference in real-world embedded applications.

1.2 Problem Statement: Mapping modern CNNs onto FPGAs

Although FPGAs are promising candidates for accelerating CNN inference due to their reconfigurability and energy efficiency [11] [9], mapping modern architectures to these platforms remains an open challenge. The core difficulties lie not in the general motivation for edge inference, but in the fundamental tension between the CNN computational characteristics and the hardware constraints of FPGA.

Memory limitations represent a persistent bottleneck and CNNs feature hundreds of layers with large parameter sets and huge feature maps. Typical FPGAs provide only a few megabytes of On-Chip Memory (OCM), forcing frequent off-chip DRAM accesses; this issue increases latency and energy consumption reducing the benefits of FPGA acceleration.

The heterogeneity of CNN layer types complicates accelerator design. While early CNNs were dominated by convolution and pooling layers, modern networks employ diverse operators such as depthwise and group convolutions, residual connections, and attention modules [6][12]. Efficiently supporting this variety on a single FPGA design requires flexible yet optimized hardware templates, which is a difficult balance to achieve.

Optimization of resource allocation and parallelism represent significant challenges, CNNs have high parallelism, but optimal scheduling across layers must carefully balance computations, memory, and bandwidth. The finite availability of Digital Signal Processing Unit (DSP) slices, Look-Up Tabless (LUTs), and interconnections restrict the level of parallel execution, making layer-by-layer optimization essential [13].

Moreover, the design productivity time limits practical deployment. Hand-coded Hardware Description Language (HDL) can produce efficient implementations, but requires substantial engineering effort. High Level Synthesis (HLS) offers higher abstraction, yet often produces suboptimal hardware without expert guidance, as a result, accelerator development remains time consuming, limiting portability and scalability [14].

Finally, the architectural paradigm for mapping CNNs to FPGAs is itself a key problem. Two main approaches dominate: overlay-based accelerators, which

offer programmability and easier re-targeting across models, and dataflow-based accelerators, which provide superior throughput and energy efficiency by directly mapping operations onto spatial pipelines. However, both approaches present trade-offs, and no universally optimal strategy has emerged [15][16].

1.3 Research Objectives

The challenges outlined in 1.2 highlight the need for systematic methodologies, the overall objective of this thesis is to bridge the gap between the computational requirements of state-of-the-art models and the resource constraints available in the devices, enabling practical and scalable inference at the edge.

For this purpose, the research conducted in this thesis has the following objectives:

- Analyze the impact of CNN architectural trends (e.g., depthwise convolutions, residual connections) on FPGA mapping efficiency and identify bottlenecks in computation, memory, and scheduling across representative FPGA devices.
- Investigate the trade-offs between overlay-based and dataflow-based accelerator designs, with respect to performance, flexibility, and design productivity.
- Develop an optimization strategy for dataflow accelerators that enhances CNN-to-FPGA mapping by optimizing thememory hierarchy, balancing computation–communication, and exploiting efficient parallelism.
- Implement and evaluate the proposed strategies on representative CNN benchmarks (e.g., ResNet, MobileNet, EfficientNet) across different FPGA platforms.
- Demonstrate improvements in throughput and energy efficiency while preserving the accuracy of the model.

The thesis aims to deliver a comprehensive framework that allows FPGA acceleration of CNNs, facilitating their deployment in resource-constrained edge scenarios.

Chapter 2

Background

2.1 Convolutional Neural Networks: An Overview

CNNs have become the dominant architecture for computer vision tasks, enabling breakthroughs in image classification, object detection, and semantic segmentation. Unlike traditional approaches that relied on handcrafted feature extraction, CNNs learn hierarchical feature representations directly from data, yielding superior accuracy and robustness [17] [18].

The modern era of deep CNNs began with AlexNet [19], which demonstrated that large-scale convolutional models trained on GPUs could dramatically outperform classical methods in the ImageNet classification challenge. Following this, architectures such as VGGNet [20] explored the impact of network depth by stacking multiple small convolutional filters, while GoogLeNet [21] introduced multibranch modules to capture features at different scales efficiently.

A major advancement in this field was the introduction of residual connections in ResNet [5], which allowed deeper networks to be trained by alleviating issues related to the vanishing gradient. Subsequent models, such as DenseNet [22], further improved information flow through dense connectivity patterns and research on efficiency led to lightweight models such as MobileNet [6] and ShuffleNet [23], which leveraged depthwise separable and grouped convolutions to reduce computational complexity, making CNNs more suitable for mobile and embedded devices.

In addition to architectural innovations, CNNs have increasingly incorporated specialized operators, normalization, and activation strategies; these advances significantly improve accuracy but they also introduce heterogeneity and irregularity in computation patterns, complicating hardware acceleration.

From a hardware perspective, the evolution of models caused an ever-increasing demands for computational throughput and memory bandwidth. For example, models like VGG-16 require hundreds of millions of parameters and tens of billions of floating-point operations per inference. Even more compact models such as MobileNet or EfficientNet, while optimized for mobile Central Processing Units (CPUs), still pose challenges for deployment in resource-constrained devices, so these trends underscore the need for specialized hardware acceleration strategies, including FPGAs and Application-Specific Integrated Circuits (ASICs).

2.2 FPGA Fundamentals

An FPGA is a type of integrated circuit that can be configured after manufacturing to implement custom hardware architectures; unlike ASICs, which are fixed at design time, or GPUs, which are optimized for general-purpose parallel workloads, FPGAs provide balance between flexibility, parallelism, and energy efficiency. This makes them particularly suitable for accelerating computationally intensive workloads such as CNNs, especially in resource-constrained scenarios.

FPGAs are made up of:

- Configurable Logic Blocks (CLBs): programmable elements built around logic gates, LUTs and flip-flops that can implement arbitrary Boolean logic.
- DSP slices: dedicated hardware units inside the fabric designed for digital signal processing and Multiply and Accumulate (MAC) operations.
- Block RAM (BRAM): embedded memory banks of dual port RAM blocks that are capable of working with different clocks, aiding in the construction of building FIFOs and dual port buffers that bridge clock domains.
- Routing interconnects: reconfigurable network that connects CLBs, DSPs, and BRAM according to the topology of the target circuit.

- I/O interfaces: connections to off-chip memory (Double Data Rate (DDR), High Bandwidth Memory (HBM)) and peripheral devices.

This configurable architecture allows designers to implement deeply pipelined, parallel compute units that directly match the structure of neural network workloads controlling the degree of parallelism, the scheduling of operations, and the placement of data in memory. In this way FPGAs can also deliver deterministic latency and energy efficiency, eventually surpassing GPUs in performance per watt for specific tasks.

Modern FPGA design is implemented by Electronic Design Automation (EDA) flows that typically begin with a high-level description of the application, traditionally relying on HDLs (e.g. VHSIC Hardware Description Language (VHDL) and Verilog) but the advent of HLS tools has made it possible to use C/C++ or SystemC as input.

The design process starts with a high-level specification, where the algorithm is described in high-level language (e.g. C/C++); then the code is synthesized into RTL, mapping the operations to logic elements, DSPs, and memory resources. Then place-and-route is performed and the synthesized design is physically mapped to the FPGA platform. Finally the bitstream is generated, which produces the binary configuration required for deployment on the target device (fig. 2.1).

2.3 CNN Acceleration on FPGAs

CNNs are naturally mapped onto the FPGA fabric due to the high degree of regularity in their operations. Convolution layers, for example, can be parallelized across output channels, input channels, and kernel elements. Pooling and activation functions are lightweight and pipelinable, while fully connected layers can be expressed as large matrix-vector multiplications. Moreover, dataflow-style execution, where layers are connected in a streaming pipeline, fits the FPGA execution model particularly well.

Despite these advantages, the deployment of CNNs on FPGAs also introduces several challenges. A first limitation is the scarcity of resources, particularly DSPs and BRAMs, which is critical on embedded devices. Another issue concerns memory bandwidth, since off-chip DRAM accesses are expensive in terms of both latency and energy. Finally, the process of DSE remains inherently complex, since the selection of unrolling factors, tiling strategies, and precision formats strongly influences both

performance and feasibility. These challenges motivate the adoption of structured methodologies, such as the accelerator paradigms discussed in the following section.

2.3.1 Accelerator Paradigms: Overlay vs. Dataflow

Unlike CPUs or GPUs, which offer fixed programming models, FPGAs require explicit design decisions about parallelism, pipelining, and memory organization. In order to address this problem two primary paradigms have emerged for Deep Neural Network (DNN) acceleration: **overlay-based** accelerators and **dataflow** accelerators.

Overlay-based approaches aim to provide a general-purpose hardware template that can support a multiple networks through reconfiguration, they are typically based on systolic arrays or matrix multiplication engines, similar to GPU or Tensor Processing Unit (TPU) tensor cores and they achieve programmability by time-multiplexing a fixed compute fabric across layers.

Overlay-based accelerators offer a core advantage: they can be reused across multiple models, thereby avoiding the need to regenerate a bitstream for each new network. This results in shorter compilation times and facilitates integration with existing deep learning frameworks. These benefits come at the cost of reduced efficiency, fixed array dimensions often lead to underutilization of resources, while the programmability required for general-purpose operation introduces logic overheads; moreover, overlays typically provide limited opportunities for streaming data reuse, which results in suboptimal memory utilization.

By contrast, dataflow accelerators implement each CNN layer, or groups of layers, as dedicated hardware pipelines. Computation progresses in a streaming fashion: as soon as the first layer produces its initial output, subsequent layers can begin processing without waiting for the entire input image. This form of layer-parallelism enables both high throughput and low latency. Dataflow architectures allow hardware resources to be tailored to the requirements of individual layers, while keeping intermediate results on-chip reduces costly off-chip memory traffic. Their performance is also predictable, being determined by the throughput of the slowest pipeline stage, which can be balanced through careful DSE. On the other hand, dataflow designs also have limitations. Each new network topology may require recompilation, and the design space exploration is inherently complex, as

optimal unrolling factors and resource allocations vary across layers and must be coordinated globally.

The NN2FPGA framework propose the dataflow paradigm and extends it with graph-level optimizations for skip connections combined with a binary integer programming approach to DSE this enabling near optimal throughput while minimizing resource waste.

2.3.2 High-Level Synthesis for FPGA-based CNNs

HLS has become a fundamental approach for accelerating neural networks for FPGAs[24]; by enabling designers to specify functionality in high-level languages such as C, C++, SystemC, or OpenCL, this approach drastically reduces the gap between algorithm development and hardware realization: instead of manually crafting RTL descriptions, developers can use automated transformations to map constructs into parallel hardware structures. This paradigm shortens the design cycle, allowing researchers to focus on architectural exploration, while still exploiting FPGA features such as deep pipelining, fine-grained parallelism, and on-chip memory hierarchies.

Vitis HLS [1] is one of the most common commercial solutions offered by AMD/Xilinx. It provides an ecosystem with a set of directives that help designers to restructure computations for performance through loop pipelining, unrolling, array partitioning, and dataflow pragmas, eventually improving parallelism in convolutional loops, reducing initiation intervals, and increasing data reuse. These optimizations are indispensable for CNN inference, where throughput is highly dependent on balancing computational parallelism with memory bandwidth. Despite its strengths, Vitis HLS remains vendor-dependent and requires careful hardware-aware programming: standard C implementations typically fail to exploit the FPGA fabric efficiently without proper pragma tuning.

An alternative commercial tool is the Intel HLS Compiler [25], integrated into the Quartus Prime toolchain. Intel's approach places emphasis on OpenCL kernels as the entry point, which makes it attractive for heterogeneous computing environments where FPGAs are deployed alongside CPUs and GPUs. The recent shift towards the oneAPI model illustrates Intel's attempt to unify the programming of CPUs, GPUs, and FPGAs under a single framework. While promising in terms of portability, its

ability to generate highly optimized streaming accelerators for CNNs has not been demonstrated as extensively as in the Xilinx ecosystem.

Other industrial-grade tools, including Cadence Stratus HLS [26] and Siemens Catapult HLS [27], are frequently used in ASIC/FPGA co-design flows. These tools rely on SystemC and provide scheduling, verification, and design space exploration capabilities; they are particularly valued in industrial contexts where accelerators must be integrated into complex System on Chips (SOCs) so their adoption in academic research for CNN inference is less widespread, but remains important for state-of-the-art design methodologies.

In addition to these commercial solutions, several open-source HLS frameworks have been developed to improve the accessibility to FPGA design. PandA-Bambu [28], developed at the Politecnico di Milano, supports C/C++ input and translates it into RTL through a transparent compilation flow comprising front-end parsing, middle-end optimizations, and back-end RTL synthesis. Its ability to perform aggressive bitwidth analysis, resource sharing, and operator customization makes it well suited for experimental research where fine-grained architectural control is essential. Another project is LegUp [29], developed at the University of Toronto and later commercialized as SmartHLS [30] by Microchip, it showed how automatically mixed software–hardware partitioning could be achieved, highlighting the potential of HLS for the design of FPGA accelerators.

MLIR-based HLS flows have emerged more recently as an evolution, projects such as ScaleHLS [31] and the SODA Synthesizer [32] leverage the Multi-Level Intermediate Representation (MLIR) to introduce optimization passes at multiple abstraction levels. These flows can automatically generate optimized HLS code annotated with pragmas by analyzing loop transformations, dataflows, and graph-level dependencies. Although still less mature than industrial-grade tools, they show a growing convergence between compiler technology and hardware synthesis, paving the way for more systematic and portable accelerator generation.

In summary, the landscape of HLS tools is very dynamic, commercial solutions such as Vitis HLS and Intel HLS dominate due to their maturity and integration with vendor ecosystems, while open-source frameworks provide transparency and flexibility for academic innovation. Emerging MLIR-based approaches suggest that the future of FPGA compilation will increasingly focus on software compiler technology with multi-level optimizations driving automated design space explo-

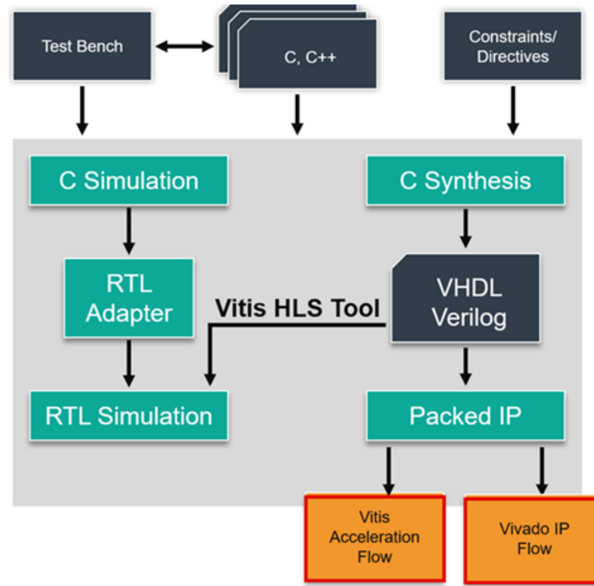

Fig. 2.1 Vitis HLS workflow (from [1])

ration. Despite their differences, existing HLS tools share limitations when applied to modern CNNs because they are often vendor specific, not easily portable, and lack integrated mechanisms to optimize across entire network graphs. These limitations set the stage for the contributions of this thesis, which extend the capabilities of HLS with graph-level transformations and optimization-driven design methodologies.

2.4 Related Work in FPGA-based CNN Frameworks

Several frameworks have been proposed to automate CNN deployment on FPGAs:

- FINN (Xilinx) [33]: targets quantized neural networks (QNNs) with a dataflow architecture. It provides a library of templated operators enriched with HLSs pragmas. FINN achieves high efficiency for binarized or ternary networks but struggles with more complex topologies like ResNet or MobileNet due to limited skip connection support.
- hls4ml [34]: originally developed for physics experiments at CERN, it converts fully connected or shallow CNNs into HLS descriptions. It prioritizes low-latency inference by fully unrolling loops and storing weights on-chip, but this approach poorly scales to deeper models.

- Vitis AI: provides a compilation stack for deploying CNNs on AMD/Xilinx FPGAs, it relies on predefined overlays and does not support dataflow-style full-network pipelining, limiting efficiency compared to specialized frameworks.
- MLIR-based tools [35] (e.g. ScaleHLS, SODA Synthesizer): explore compiler-driven generation of accelerators. These frameworks combine graph-level optimizations with HLS backends, promising portability and extensibility. However, their maturity is still limited compared to that of industrial tools.
- fpgaConvNet [36]: provides an automated toolflow to generate FPGA-based CNN accelerators through graph-level transformations and a simulated annealing (SA) DSE guided by an analytical performance model. The framework provides dataflow-style accelerators and achieves high throughput on conventional convolutional layers. However, SA-based exploration is heuristic, meaning that the selected configuration is not guaranteed to be globally optimal.

These frameworks present three main limitations. First, they exhibit a strong dependence on vendor-specific HLS tools and predefined operator libraries, which impedes portability and flexibility. Second, frameworks offer only limited support for complex network structures, such as skip connections or other irregular topologies that restrict their applicability to modern CNN architectures. Finally, they generally lack global strategies for resource allocation and throughput optimization, resulting into suboptimal performance when mapping large and heterogeneous models onto the available FPGA.

These issues justify nn2FPGA framework, which integrates graph rewriting, buffering-aware compilation, and Binary Integer Programming (BIP) DSE, aiming to combine the efficiency of dataflow accelerators with the flexibility of automated compilation.

Chapter 3

The nn2FPGA Compiler Architecture

3.1 Introduction to the Compiler Framework

The design of efficient FPGA-based accelerators for DNN requires a comprehensive toolchain that integrates the description of software-level model with the corresponding hardware implementation. HLS tools such as Vitis HLS simplify the generation of RTL from C/C++ code, but the challenge of converting complex neural network graphs into efficient dataflow architectures still remains. The nn2FPGA compiler was designed to address this challenge by automating the entire flow from a quantized deep learning model to a deployable FPGA bitstream.

The framework focuses specifically on CNNs with skip connections, such as ResNets and MobileNetV2. These kind of networks introduce irregular computation patterns that are difficult to map into static dataflow pipelines due to the synchronization requirements of residual blocks. The compiler provides both graph-level optimizations and resource-aware scheduling to mitigate buffering overheads and maximize throughput under hardware constraints.

The flow includes quantization-aware training and integer-only inference, ensuring that the generated circuits are efficient from resources point of view and compatible with FPGA DSP blocks. The network is first exported as a standardized intermediate representation Quantized ONNX (QONNX) [37], optimized and transformed into synthesizable C++ code, and finally compiled into RTL and FPGA bitstreams using AMD Vitis and Vivado tools. Through its modular design, nn2FPGA provides three main contributions:

- A quantization pipeline based on integer arithmetic and batch normalization folding, ensuring minimal accuracy loss while improving hardware compatibility.
- A static dataflow accelerator template library that can be customized for each network layer, with built-in support for hierarchical pipelining and streaming execution.
- A set of graph optimization steps for residual blocks that reduces buffering requirements in skip connections, enabling efficient acceleration of modern CNNs under resource-constrained devices.

This chapter describes the architecture of nn2FPGA, starting with the quantization process and model preparation (Section 3.2), followed by the compiler flow and intermediate representations (Section 3.3), and the accelerator architecture (Section 3.4). Then optimizations for convolutions (Section 3.5) are discussed before concluding with the description of window buffer and graph-level optimizations for skip connections (Section 3.6).

3.2 Quantization and Model Preparation

Quantization plays a fundamental role in the nn2FPGA compilation flow, as it reduces both the memory footprint and the computational cost of deep neural networks (DNNs) while enabling efficient mapping onto FPGA resources. In practice, quantization replaces floating-point arithmetic, typically in single precision, with fixed-point integer operations. This reduction in bitwidth allows for more compact storage of weights and activations, decreases memory bandwidth requirements, and facilitates direct exploitation of DSP blocks available on modern FPGAs.

3.2.1 Quantization Methods

Two main approaches can be adopted to perform quantization: Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT). PTQ is applied to a pretrained floating-point model and is attractive for its simplicity and rapid deployment. However, PTQ often suffers from accuracy degradation, especially for complex

architectures or low bitwidth configurations. In contrast, QAT integrates the quantization process into the training loop: the forward and backward passes explicitly simulate reduced precision, allowing the model to adapt to quantization effects and thus mitigating accuracy loss. Although QAT requires additional training effort, it typically achieves superior trade-offs between accuracy and hardware efficiency.

In nn2FPGA, quantization is performed using a symmetric and uniform scheme with power-of-two scaling factors. This choice simplifies hardware implementation, since rescaling operations can be mapped to shift operations instead of costly multiplications. Given a floating-point value b and a target precision of bw bits, quantization is defined as:

$$Q(b) = \text{clip}\Big(\text{round}(b \cdot 2^{-s}), a_{\min}, a_{\max}\Big) \cdot 2^{s}, \tag{3.1}$$

where s is the scaling exponent, and $a_{\min}$, $a_{\max}$ represent the clipping bounds:

$$a_{\min} = act_{\min}(s) = \begin{cases} 0 & \text{if unsigned} \\ -2^{bw-1-s} & \text{if signed} \end{cases} \tag{3.2}$$

$$a_{\max} = act_{\max}(s) = \begin{cases} 2^{bw-s} - 1 & \text{if unsigned} \\ 2^{bw-1-s} & \text{if signed} \end{cases} \tag{3.3}$$

Zero-points are omitted as the scheme is symmetric around zero. This formulation ensures efficient alignment across layers while avoiding unnecessary overheads during inference.

Illustrative example

To clarify the integer quantization process described in the previous subsection, a simple example consistent with the nn2FPGA integer quantization format [38] is reported.

Consider a floating-point activation value $b = 0.37$, quantized to an 8-bit signed integer using a power-of-two scaling factor. Let $2^s = 2^4 = 16$. The quantized representation is

$$\hat{b} = \text{round}(b \cdot 2^4) = \text{round}(5.92) = 6.$$

The dequantized result used during inference is:

$$Q(b) = \hat{b} \cdot 2^{-4} = 0.375.$$

This example illustrates how to implement hardware-friendly quantization with power-of-two scaling factors, since multiplications reduce to bit-shifts during computation.

3.2.2 Brevitas and QONNX Generation

The training and quantization process is implemented using the Brevitas [2] library, a PyTorch-based [39] framework for quantization-aware training. Brevitas provides fine-grained control over weight and activation bitwidths and exports the quantized model into the Open Neural Network Exchange (ONNX) format. QONNX is an extension of the standard ONNX representation that includes quantization parameters such as bitwidth, scaling factors, and signedness for each kind of layer.

This enriched representation is used as the input for the nn2FPGA compiler. By embedding quantization metadata directly into the model graph, QONNX enables full automation of the compilation process. It also guarantees interoperability with other toolchains, ensuring that models can be exchanged and benchmarked across different frameworks and tools.

Table 3.1 compares integer quantization using power-of-two scaling factors versus floating point scaling factors. This comparison is based on Torchvision pretrained models and demonstrates that the choice of either method results in minimal accuracy loss with respect to the floating point baseline.

3.2.3 Batch Normalization Folding

An important optimization applied during preprocessing is the removal of batch normalization (Batch Normalization (BN)) layers through parameter folding. During

Table 3.1 Comparison of integer PTQ using power-of-two versus floating point scaling factors on ImageNet[2].

Model	Quant.	Weight bit	Act. bit	Scaling	Accuracy
MobilenetV2	Floating point baseline				71.90%
	INT	8	8	FP32	-0.83%
	INT	8	8	Po2	-0.77%
ResNet18	Floating point baseline				69.76%
	INT	8	8	FP32	-0.06%
	INT	8	8	Po2	-0.14%
ResNet50	Floating point baseline				76.13%
	INT	8	8	FP32	-0.44%
	INT	8	8	Po2	-0.23%
ResNet101	Floating point baseline				77.37%
	INT	8	8	FP32	-0.49%
	INT	8	8	Po2	-0.34%

inference, BN can be expressed as an affine transformation, which can be merged with the preceding convolution by modifying the values of its weights and biases [40].

Let $y = Wx + b$ be the output of a convolutional layer, followed by batch normalization with parameters γ, β, μ, and σ^2. The transformation becomes:

$$\hat{y} = \gamma \cdot \frac{(y-\mu)}{\sqrt{\sigma^2+\varepsilon}} + \beta = W'x + b', \tag{3.4}$$

where W' and b' are the updated weights and biases. By folding BN operations into convolution kernels at compile time, the number of operators to be synthesized is reduced and memory accesses are minimized, thus decreasing both latency and resource usage on the target FPGA.

Given a filter weight w, batch-normalization parameters γ, β, and variance σ, the folded weight W' becomes:

$$W' = \frac{\gamma}{\sqrt{\sigma+\varepsilon}} \cdot w, \qquad b' = \beta - \frac{\gamma\mu}{\sqrt{\sigma+\varepsilon}}.$$

The resulting W' and b' are then quantized using the same integer format as above. This preprocessing stage produces a graph consisting only of quantized operators (convolutions, pooling, element-wise functions), optimized for subsequent HLS-based synthesis.

3.3 Compiler Flow and Intermediate Representations

The proposed nn2FPGA compiler automates the mapping of a quantized neural network to an efficient FPGA accelerator by combining graph-level transformations, automated code generation, and vendor supported HLS synthesis. The compilation flow is structured into three main phases: translation of the quantized model into HLS C++ code, generation of RTL with Vitis HLS, and integration into a deployable system through Vivado.

3.3.1 From QONNX to HLS C++ Code

The input of the compiler is a QONNX description of the neural network obtained through Brevitas. The compiler parses this representation and automatically generates synthesizable C++ code. Each operator in the network is mapped to a corresponding module from the nn2FPGA template library, which contains hardware-optimized implementations of convolutions, pooling, fully connected layers, and element-wise operators.

As described in [38], this step also performs graph-level optimizations. In particular, batch normalization layers are folded into convolution operators, and redundant element-wise additions are merged when possible. These transformations reduce both memory traffic and the number of compute kernels to be generated. The final output is a set of parameterized C++ functions annotated with HLS pragmas that expose parallelism and enforce streaming communication between operators.

3.3.2 RTL Generation with Vitis HLS

The generated C++ code is synthesized into RTL using Vitis HLS. Each network operator is implemented as an independent hardware process, while inter-layer

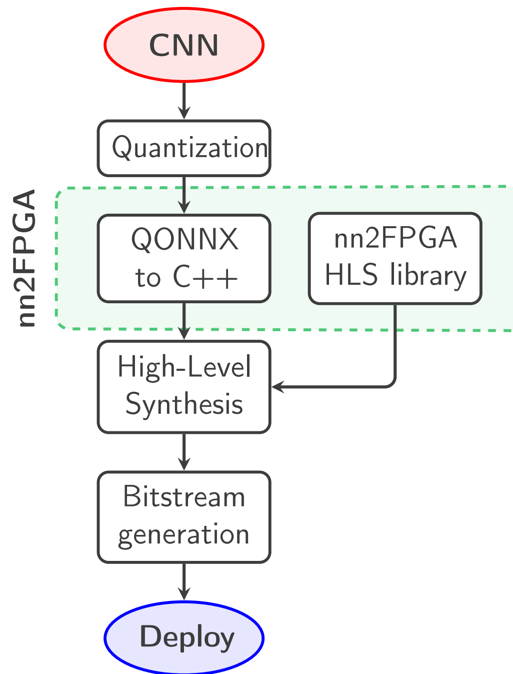

Fig. 3.1 Implementation flow

communication is managed through First-In First-Out (FIFO)-based channels. The use of the `DATAFLOW` and `PIPELINE` directives allows the compiler to instantiate deeply pipelined execution units, enabling concurrent operation of different layers in a static dataflow fashion.

At this stage, the design parameters, such as loop unrolling and memory partitioning factors, have already been determined by the BIP-based DSE, ensuring an optimized allocation of computational resources. The subsequent HLS synthesis provides detailed performance and utilization reports, including latency, initiation interval, and the occupation of LUTs, flip-flops, BRAMs, and DSPs. These reports are used to validate that the synthesized design meets the expected resource constraints and to confirm the correctness of the design space exploration results.

3.3.3 Vivado Integration and Bitstream Generation

After HLS synthesis, the model is exported as an Intellectual Property (IP) block and integrated into a complete block design system through Vivado where standard bus protocols, such as AXI4-Stream for data transfers and AXI4-Lite for configuration, ensure interoperability with embedded processors and external memory controllers.

Finally, the accelerator is synthesized, placed and routed to generate the final bitstream for the target FPGA. Timing closure is a crucial stage, the tool applies balancing strategies to avoid resource overutilization and mitigate routing congestion, thus increasing the probability of achieving stable operating frequencies on different FPGA platforms.

3.4 Accelerator Architecture

3.4.1 Static Dataflow Model

nn2FPGA compiler generates a hardware accelerator that follows a static dataflow paradigm, in which each operator of the neural network is implemented as a dedicated hardware block connected through streaming interface; the resulting architecture enables high throughput by allowing multiple layers to execute concurrently, with the global performance bounded by the initiation interval of the slowest stage in the pipeline.

The accelerator consists of three main elements: operator tasks derived from the QONNX model, streaming and memory interfaces for data transfer, and a hierarchical pipelining scheme that overlaps computations efficiently. Each task corresponds to a network operator such as a convolution, pooling, or activation layer and communicates with others through FIFO channels. Concurrent execution of different layers is enabled by the use of the `DATAFLOW` pragma in Vitis HLS, it ensures that the computation starts as data become available, avoiding the need to store intermediate feature maps in off-chip memory.

In this way, the performance of the model is predictable: latency and throughput can be analytically estimated from the structure of the tasks. This static mapping strategy is particularly good for edge inference workloads, where deterministic and predictable performance are essential.

3.4.2 Task Categories

Tasks in the accelerator can be classified into two main categories:

- **Compute tasks**: these include convolution, fully connected, and element-wise operations. They dominate resource usage and determine the overall throughput of the system.
- **Memory tasks**: these handle data transfers between on-chip buffers and external memory. They are implemented with DMA engines connected via Advanced eXtensible Interface (AXI) interfaces.

This categorization make simpler the compiler backend mapping each operator in the QONNX graph to one of these hardware task templates.

3.4.3 Streaming and DMA Execution Model

In order to achieve high throughput, the accelerator relies on a fully streaming execution model. Intermediate activations are transmitted between tasks through point-to-point FIFO channels generated by HLS. Data entering or leaving the accelerator is handled by DMA modules connected to the off-chip memory. The compiler automatically generates these interfaces and assigns memory address spaces to input and output tensors.

This design avoids redundant memory traffic: feature maps are not stored in external memory that is only used for inputs or at the final output stage. All intermediate data are kept within the pipeline while weights are stored into on-chip memory (BRAM and URAM) improving latency and energy efficiency compared to designs where each layer is executed sequentially with off-chip storage of partial results.

3.4.4 Hierarchical Pipelining

The accelerator employs hierarchical pipelining at two levels:

1. **Loop-level pipelining**: achieved through the `PIPELINE` pragma ensuring that loop bodies in compute tasks have an Initiation Interval (II) equal to 1.
2. **Task-level pipelining**: achieved using the `DATAFLOW` pragma, allowing multiple operators to run concurrently as soon as their inputs are available.

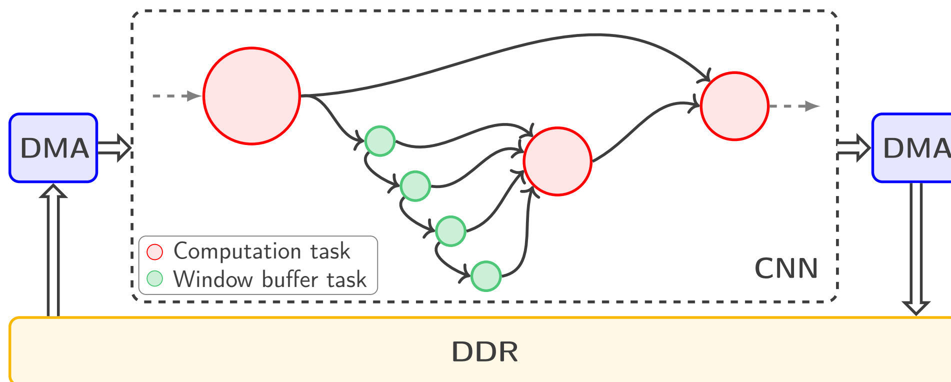

Fig. 3.2 Accelerator architecture with Direct Memory Access (DMA) blocks for memory transfers (blue boxes) and concurrent tasks communicating through data streams.

This approach allows the accelerator to maintain steady throughput once the pipeline is filled, with latency dominated only by the depth of the pipeline. By the minimization of the II of all the tasks, the compiler ensures that no single operator becomes a bottleneck in the execution chain.

3.5 Convolution Computation Engine

Convolutional layers dominate the computational cost of modern CNNs, making their efficient mapping onto FPGAs a crucial point of accelerator design. In the *nn2FPGA* compiler, convolutional operators are implemented through a parameterized *Convolution Computation Engine*. This module supports multiple convolution types, including standard, depthwise, and pointwise convolutions, and is designed to maximize throughput by exploiting parallelism at different levels. Furthermore, several optimizations targeting the utilization of DSP blocks are applied to sustain performance while respecting resource constraints.

3.5.1 Parallelization Strategies

Each convolution receives a window of input activations from a window buffer task, while reading the needed filter windows from the local on-chip memory. The C++ HLS code outlined in Alg. 1 shows the convolution process and examples of how

Table 3.2 Symbol definitions.

Symbol	Description
ich	Input tensor channels
ih	Input tensor height
iw	Input tensor width
och	Output tensor channels
oh	Output tensor height
ow	Output tensor width
fh	Filter tensor height
fw	Filter tensor width
s	Convolution stride

the computation pipeline receives input data and computes the partial results. Figure 3.3a provides a visual representation of the convolution architecture.

The innermost loops are completely unrolled over the filter dimensions and partially over the input channels, output channels and output width dimensions. In a typical convolution, *nn2FPGA* exploits three types of parallelization:

- Over input channels (ich^{par}): processing multiple 2D-windows of the same pixel simultaneously, along the depth of the tensor.
- Over output channels (och^{par}): processing multiple filters concurrently.
- Over output width (ow^{par}): processing multiple output activations in parallel.

The unroll factors denoted by "*par*" imply fully data-parallel execution and are closely tied to the number of Processing Elements (PEs) utilized. The relation between unrolling factors and resources is analyzed in chapter 4, as they are leveraged by DSE to achieve low latencies and high global throughput.

Depthwise convolutions are a special case of the generic convolution, in which a different convolutional filter is applied to each channel, separately. From the perspective of Alg. 1, depthwise convolution can be reinterpreted as a generic convolution with just one filter and without the sum of partial results across input channels. Following the *nn2FPGA* notation, this translates into having *och*, and by consequence och^{par}, equal to 1. fig. 3.3b depicts the resulting depthwise convolution architecture. The overall throughput of the kernel is proportional to the product of these parallelism factors. However, their combined effect must be balanced

```
1  template <typename a_in_t, typename a_out_t,
2            typename w_t, typename b_t, size_t ich,
3            size_t iw, size_t ih, size_t och,
4            size_t ow, size_t oh, size_t fh,
5            size_t fw, size_t ich_par, size_t och_par,
6            size_t ow_par, ...>
7  void conv(hls::stream<a_in_struct_t> &in_stream,
8            hls::stream<a_out_struct_t> &out_stream,
9            hls::stream<w_struct_t> &weights_stream,
10           hls::stream<b_struct_t> &bias_stream)
11 {
12   t_acc s_acc_buff[och / och_par][och_par * ow_par];
13 #pragma HLS array_partition variable=s_acc_buff dim=3
14 #pragma HLS expression_balance off
15
16   for (auto wh_i=0; wh_i < (ow*oh)/ow_par; wh_i++){
17     for (auto ich_i=0; ich_i < ich; ich_i+=ich_par){
18       for (auto och_i=0; och_i < och; och_i+=och_par){
19 #pragma HLS pipeline II=1 style=stp
20
21         w_t weights[och_par][ich_par][fh][fw];
22         b_t bias[och_par];
23         a_in_t input[ich_par][fh][fw + (ow_par - 1)];
24         a_out_t output[och_par];
25
26         weights = weights_stream.read();
27         if (ich_i == 0) bias = bias_stream.read();
28         if (och_i == 0) input = in_stream.read();
29
30         // Completely unrolled section
31         for (auto s_och=0; s_och < och_par; s_och++){
32           for (auto s_ow=0; s_ow < ow_par; s_ow++){
33             for (auto s_ich=0; s_ich < ich_par; s_ich++){
34               compute_core<fh, fw>(s_acc_buff,
35                 input, weights, bias, output);
36             }
37           }
38         }
39
40         if (ich_idx == ich - ich_par)
41           out_stream.write(output);
42       }
43     }
44   }
45 }
```

Algorithm 1 Simplified convolution code from the *nn2FPGA* HLS library. The convolutional loop is split into two parts, allowing the unrolled section to be adjusted based on template parameters.

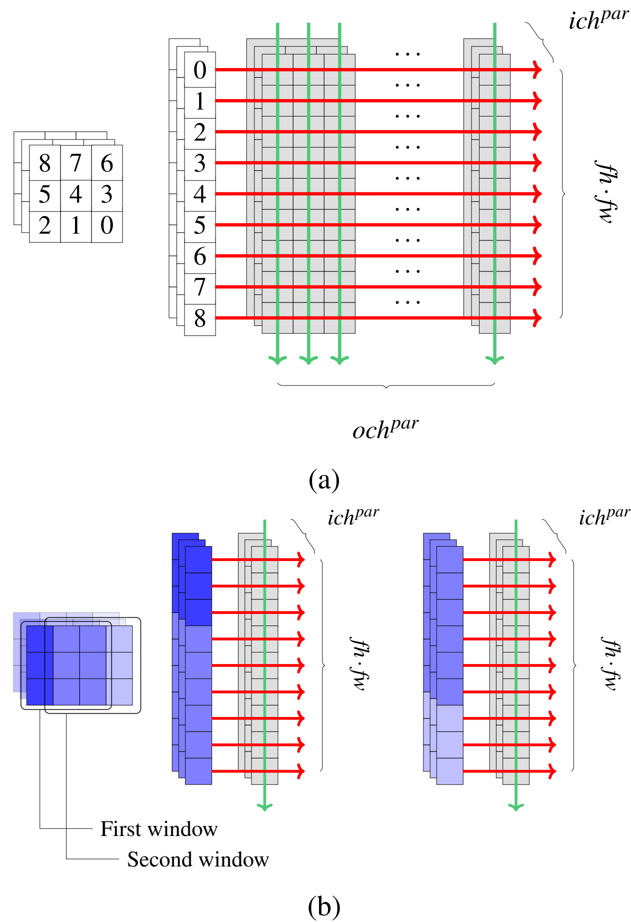

Fig. 3.3 Convolution architectures. Activations follow horizontal lines, while accumulators follow vertical ones. (a) depicts a 3×3 standard convolution. (b) depicts a 3×3 depthwise convolution with two overlapping windows convolved simultaneously.

against the availability of DSP, BRAM, and routing resources on the target device. The compiler explores this design space through a BIP formulation, selecting the parallelization parameters that maximize throughput under resource constraints.

3.5.2 DSP Optimizations

Modern FPGAs provides a limited number of DSP blocks, which represent the most efficient resources for implementing MAC operations. To maximize their utilization, the *nn2FPGA* compiler introduces two complementary optimization techniques: *DSP packing* [41] and *DSP chaining*.

DSP Packing

Since quantization reduces operands to 4-8 bits, multiple low-bitwidth multiplications can be executed within a single DSP block. For instance, two INT8 multiplications can be performed on a single DSP configured in SIMD mode. This packing strategy effectively doubles the arithmetic throughput without increasing the number of DSP resources. Moreover, when even smaller precisions are employed, the compiler can map up to four multiplications per DSP slice, further improving efficiency. Experimental evidence shows that DSP packing provides a substantial boost in arithmetic density, enabling the deployment of larger networks on resource-constrained FPGAs [42][43].

DSP Chaining

In addition to packing, DSP blocks can be configured in cascade mode to support chained operations (fig. 3.4). This feature reduces routing delays by connecting the output of one DSP directly to the input of the next without requiring additional logic or interconnect resources. Chaining is particularly effective when implementing long accumulation chains for convolutions with high channel counts. By minimizing routing overhead, chaining allows the accelerator to achieve higher operating frequencies, sustaining throughput even under aggressive parallelization. Although this technique increases pipeline depth, it effectively reduces resource usage by leveraging the internal adders within the DSPs. The Vitis HLS pragma `EXPRESSION_BALANCE OFF` can force the tool to create longer chains of operation.

DSP packing and chaining ensure that the computation block achieves high arithmetic intensity and scales efficiently to large convolutional layers. These optimizations are important for mapping modern networks with high number of MACs operations within the limited DSP budgets of embedded FPGAs.

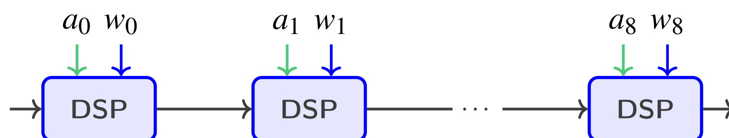

Fig. 3.4 DSP chain propagating the accumulator.

Parallelization Example

To illustrate the effect of the three parallelization factors (`ich_par`, `och_par`, `ow_par`), consider a convolution layer with:

$$ICH = 32, \quad OCH = 64, \quad FH = FW = 3, \quad OW = 56.$$

Assume the design-space exploration selects:

$$\texttt{ich_par} = 4, \qquad \texttt{och_par} = 8, \qquad \texttt{ow_par} = 2.$$

From Algorithm 1 the resulting computation parallelism is:

$$c_{\text{par}} = \texttt{ich_par} \cdot \texttt{och_par} \cdot \texttt{ow_par} \cdot FH \cdot FW = 4 \times 8 \times 2 \times 3 \times 3 = 576,$$

meaning that 576 multiply–accumulate operations are initiated every cycle inside the unrolled region of Algorithm 1.

The number of DSP used by this configuration is proportional to the parallelism:

$$D_i = \frac{c_{\text{par}}}{\mathsf{pack}},$$

where pack is the number of MACs that can be computed by a single DSP with quantized arithmetic. With 8-bit operands (pack = 2), the DSP count becomes:

$$D_i = \frac{576}{2} = 288 \text{ DSPs}.$$

If 4-bit packing is enabled (pack = 4), the usage reduces to:

$$D_i = \frac{576}{4} = 144 \text{ DSPs}.$$

This example highlights the direct connection between the chosen parallelization factors and the number of DSPs required. Any increase in `ich_par`, `och_par`, or `ow_par` increases the count of MAC operations carried out each cycle, and therefore the associated DSP usage, unless the design benefits from a higher packing capability. The optimization model included in the DSE explicitly encodes this dependency,

ensuring that throughput improvements are weighed against the available DSP and memory resources during the **ILP!** (**ILP!**)-based selection of the unrolling parameters.

3.6 Window Buffer Organization

Efficient management of input feature maps is crucial for sustaining the throughput of the Convolution Computation Engine. Convolutional layers are based on sliding windows, which require multiple overlapping accesses to the same pixels. A design that fetches each pixel from external memory whenever needed would lead to prohibitive bandwidth requirements and increased latency. To address this challenge, the *nn2FPGA* compiler generates a *window buffer* architecture, organized as line buffers and partitioned to exploit parallelism. The main purpose of this structure is to minimizes redundant memory transfers and enables high reuse of input activations directly on-chip.

3.6.1 Line Buffer Design

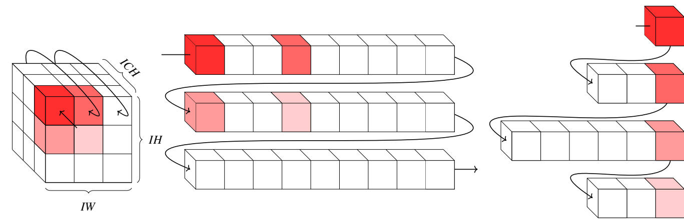

Fig. 3.5 Visual representation of how a 2×2 window is spread in the flattened tensor and then partitioned to form the line buffer. The activation tensor (leftmost image) is streamed in depth-first order, resulting in the flattened representation stored in the line buffer (center image). The line buffer is partitioned so that a whole input window is simultaneously available at its read points (rightmost image).

In nn2FPGA, each convolution or pooling operator has his own dedicated *window buffer task*, which provides the activation windows required for the associated computation (fig. 3.5). Instead of using a traditional image-processing line buffer that explicitly stores multiple rows of the feature map, this module is implemented as a set of partitioned FIFO structures that stream the input activations in the order required by the convolution block. Each FIFO holds the subset of data elements

necessary to form an input window and forwards them to the computation task once they become available.

The buffer size for each task is analytically computed from the convolution parameters and is equal to:

$$B_i = \big[(fh_i - 1) \cdot iw_i + fw_i - 1\big] \cdot ich_i \tag{3.5}$$

where fh and fw represent the height and width of the filter, iw is width of the input tensor and ich the number of input channels. This formulation guarantees that the buffer stores exactly the number of activations needed to provide a complete window at any time; as the input stream advances, new activations enter the buffer while the old ones are shifted out, allowing continuous reuse of data across successive windows without redundant off-chip memory transfers.

These buffers are synthesized using either on-chip BRAM blocks or distributed LUT-based memory, depending on the buffer depth and resource availability on the target FPGA. The Vitis HLS backend automatically configures the memory type to ensure an optimal trade-off between latency, area, and bandwidth.

3.6.2 Partitioning Strategies

The window buffer is partitioned into several parallel FIFO segments to allow concurrent execution of multiple processing elements. The partitioning strategy directly follows the unrolling factors determined during the BIP-based DSE based on the spatial (ow^{par}) and channel (ich^{par}) parallelization parameters. Each segment provides a portion of the input window, enabling simultaneous access by different convolution pipelines.

For instance, when several output pixels are computed in parallel ($ow^{par} > 1$), activations participate in multiple overlapping input windows. To provide sufficient read bandwidth, the main FIFO is divided into $f_h \times f_w$ sub-FIFOs, each corresponding to a position within the convolution window. These are connected sequentially to ensure that activations advance through the pipeline at the correct rate and reach the appropriate processing elements without duplication. When $ow^{par} = 1$ (fig. 3.6, each activation traverses every FIFO slice, as only one window is evaluated at a time.

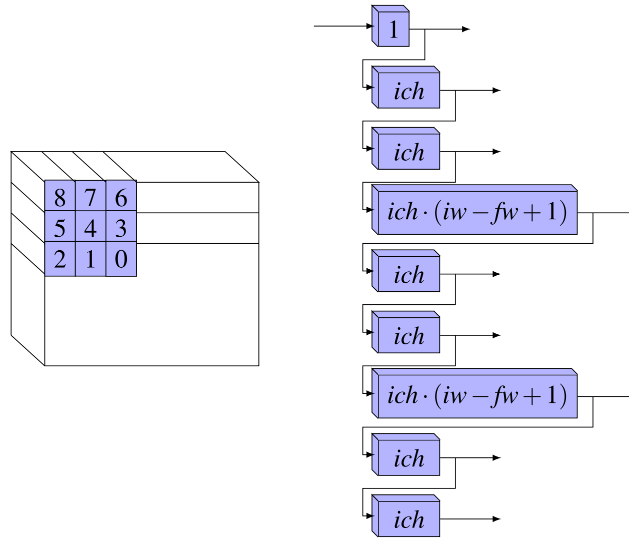

Fig. 3.6 Buffer partitioning in case of a 3×3 window, with $ow^{par} = 1$.

In contrast, for $ow^{par} > 1$ (fig. 3.7), the activations skip a number of FIFO segments equal to the unrolling factor, ensuring that multiple windows are formed concurrently. This partitioned structure provides the required data reuse and memory parallelism without replicating entire buffers, minimizing BRAM overhead while maintaining continuous throughput.

3.6.3 Impact of Stride and Parallelism

The window buffer organization also accounts for convolution stride and the type of parallelism applied. When a stride $s > 1$ is used, only one out of every s input activations contributes to the computation. The buffer control logic therefore includes selective forwarding mechanisms that discard unused activations as the data stream progresses, without interrupting the pipeline. This allows the design to preserve streaming efficiency even in subsampled convolution layers.

High spatial parallelism (ow^{par}) combined with stride introduces additional challenges, as activations are consumed in non-contiguous positions of the input tensor. In these cases, the compiler adapts the FIFO structure to maintain consistent access patterns by inserting appropriate gaps between active positions.

The nn2FPGA window buffer architecture efficiently reuse of activations while maintaining deterministic throughput; by relying on sized and partitioned FIFO

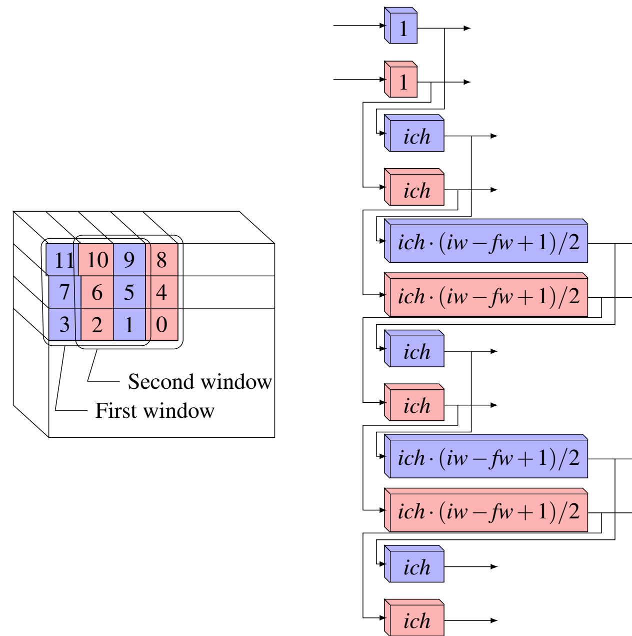

Fig. 3.7 Buffer partitioning in case of a 3×3 window, with $ow^{par} = 2$. The separate flows of activations belonging to the first and second windows are represented in red and blue.

structures instead of replicated line buffers, the design minimizes redundant data movement reducing off-chip memory accesses and achieving balanced utilization of BRAMs and LUTs. This streaming approach is a key element of the static dataflow architecture that reinforce the entire accelerator pipeline.

3.6.4 Optimization Techniques

The implementation of residual connections on FPGAs poses a major challenge due to the need to synchronize multiple parallel streams while avoiding excessive buffering overhead. In conventional dataflow designs, skip branches require large on-chip memories to temporarily store feature maps until their counterparts in the main branch become available. This overhead not only increases memory footprint but also risks reducing the throughput. To address these limitations, the nn2FPGA framework integrates specialized graph-level optimizations that reshape the computation and dataflow structure of residual blocks. In particular, two key techniques have been devised to handle different types of residual connections efficiently: *temporal reuse* for blocks without downsampling, and *loop merging* for those with a downsampling

convolution in the short branch. These optimizations reduce memory requirements and improve synchronization between parallel processes, thus enabling a more resource-efficient mapping of modern network architectures onto static dataflow accelerators.

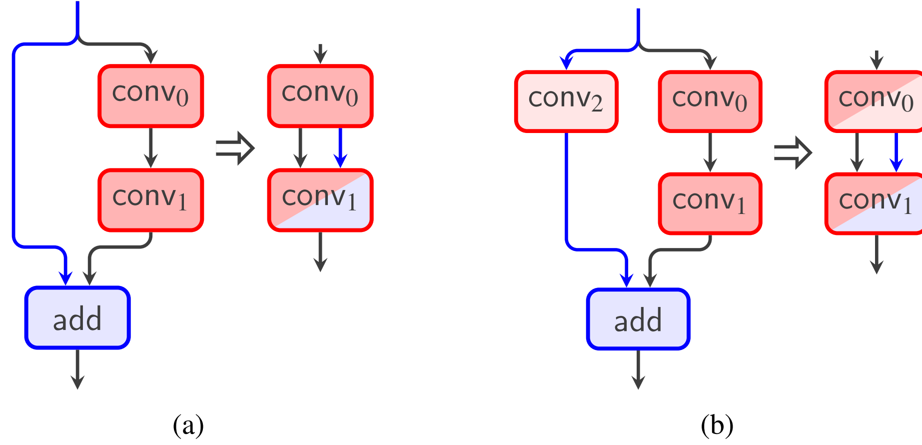

Fig. 3.8 Complete graph optimization of residual blocks. (a) and (b) depict the residual block without and with the downsampling convolution in the skip connection.

Temporal Reuse

Residual blocks without downsample typically require duplicating the input tensor to feed both the skip and the convolution branches. The nn2FPGA framework optimizes this redundancy through a temporal reuse mechanism, so the compiler introduces a secondary output stream that forwards activations once they have been consumed by the window buffer of the main convolution task avoiding the buffering of the same tensor. As a result, the skip connection only requires storage equal to the size of the convolution window buffer, drastically reducing the buffering overhead.

Formally, the optimized buffering requirement is:

$$B_{sc}^{opt} = \Big[(f_{h1} - 1) \cdot i_{w0} + f_{w1}\Big] \cdot i_{ch0}$$

which replaces the larger receptive-field-based storage of the baseline implementation. The reduction is therefore:

$$B_{sc} - B_{sc}^{opt} = \Big[(f_{h0} - 1) \cdot i_{w0} + f_{w0} - 1\Big] \cdot i_{ch0}$$

This optimization lowers the memory footprint of residual connections. Experimental results confirmed significant savings in on-chip activation storage for ResNet models, where both downsampled and non-downsampled residual bottlenecks are present (fig.3.8a).

Loop Merging

For residual blocks with downsampling, the short branch includes an additional pointwise convolution. Executing this layer independently would require extra buffering and duplicate memory accesses. To avoid this inefficiency, nn2FPGA applies loop merging: the two convolutions acting on the same tensor (the downsample convolution in the skip branch and the convolution in the main branch) are fused into a single pipeline.

This transformation allows the two layers to share the same window buffer and prevents duplication of input data. Furthermore, the add operation that merges the skip and main branches is incorporated directly into the convolution pipeline by initializing the accumulator with the skip branch contribution. Consequently, the summation occurs within the same loop nest as the convolution, adopting the same unrolling factors, thus avoiding computation (fig.3.8b).

Overall, loop merging enables more efficient residual block implementation by reducing redundant memory storage and decreasing latency by sharing buffers,

This optimization is particularly relevant for downsampled residual connections in ResNets, while models such as MobileNetV2 benefit only from temporal reuse due to their inverted residual.

Chapter 4

Resource Allocation and Design Space Exploration

The design of efficient accelerators for CNNs on FPGAs requires a careful balance between computational parallelism and the limited hardware resources available on the device. Although the architectural techniques discussed in Chapter 3 (e.g., pipelining, window buffering, and graph-level transformations) are crucial to improve the efficiency of individual operators, the overall throughput of a static dataflow accelerator is ultimately determined by the slowest task in the pipeline. Therefore, a global optimization strategy is needed to allocate FPGA resources in a way that maximizes system performance under tight constraints of DSPs, BRAMs, and memory bandwidth.

Convolutions vary significantly in terms of input and output dimensions, filter size, and channel depth, which directly impact their computational load and memory access patterns; if resources are allocated uniformly across layers, bottlenecks may appear, leading to suboptimal throughput. Over-parallelizing noncritical layers can cause unnecessary resource consumption, complicating placement and routing, and reducing the maximum achievable clock frequency, hence an intelligent resource allocation strategy must prioritize layers that limit the global pipeline performance while avoiding redundant parallelism in other stages.

To address this problem, the nn2FPGA framework introduces a DSE methodology based on BIP [38]. The model systematically enumerates feasible parallelization factors for each layer and formulates the allocation problem as a constrained op-

timization task. The first optimization step focuses on maximizing throughput, ensuring that the latency of the slowest concurrent process is minimized, then a second optimization step reduces the parallelism of non-bottleneck layers to free unused resources, improving design feasibility and lowering the risk of implementation failures in the FPGA backend flow.

This chapter presents in detail the resource allocation used in nn2FPGA. Section 4.1 introduces the mathematical formulation of the problem, including latency and resource models for convolutional and buffering tasks. Section 4.2 discusses the impact of different parallelization strategies and their relation to DSPs and memory utilization. Section 4.3 formalizes the BIP algorithm while Section 4.4 details the optimization flow implemented by the compiler. Section 4.5 concludes with a critical discussion of the trade-offs and implications of this approach.

4.1 Problem Formulation

The throughput of a static dataflow accelerator for CNNs is determined by the slowest concurrent task in the processing pipeline. Each convolutional or pooling layer in the network graph is mapped onto two main hardware tasks: a computation task and a window buffer task. To maximize the global throughput, the resource allocation problem must be formulated in a way that balances these tasks under the constraints of the target FPGA.

Let S denote the latency of the slowest concurrent task. The optimization objective is then defined as the minimization of S subject to hardware resource limits, namely the number of available DSPs and memory banks. Formally, the problem can be expressed as:

$$\min_{S \in \mathbb{N}^+} S \tag{4.1}$$

subject to:

$$L_i^c \leq S \quad \forall i \in G, \tag{4.2}$$

$$L_i^w \leq S \quad \forall i \in G, \tag{4.3}$$

$$\sum_{i \in G} D_i \leq DSP, \tag{4.4}$$

$$\sum_{i \in G} M_i \leq MEM, \tag{4.5}$$

where G is the set of convolutional and pooling layers in the network, L_i^c and L_i^w are respectively the computation and window buffer latencies of layer i, D_i is the number of DSPs required by the layer, and M_i is the number of memory banks required. The constants DSP and MEM denote the total number of available resources on the target FPGA.

The computation cost of each convolutional layer depends on its output dimensions, filter size, and input depth. For a generic convolutional layer i, the number of operations is given by:

$$c_i = o_h^i \cdot o_w^i \cdot o_{ch}^i \cdot i_{ch}^i \cdot f_h^i \cdot f_w^i, \tag{4.6}$$

where o_h^i and o_w^i are the output height and width, o_{ch}^i and i_{ch}^i are the number of output and input channels, and f_h^i, f_w^i are the filter height and width.

To reduce the latency, FPGA accelerators exploit parallelism through loop unrolling. Let the parallelization factors be defined as ow_i^{par}, och_i^{par}, and ich_i^{par}, respectively for the output width, output channels, and input channels. The resulting computation parallelism of layer i is then:

$$c_i^{par} = ow_i^{par} \cdot och_i^{par} \cdot ich_i^{par} \cdot f_h^i \cdot f_w^i, \tag{4.7}$$

which corresponds to the number of two-dimensional windows computed simultaneously. Consequently, the computation latency of the layer can be expressed as:

$$L_i^c = \frac{c_i}{c_i^{par}} + \varepsilon, \tag{4.8}$$

where ε is the additional latency introduced by pipeline depth (typically negligible with respect to the dominant term).

The window buffer latency depends on the number of activations to be shifted in parallel. For a layer i, it is modeled as:

$$L_i^w = \frac{i_{ch}^i \cdot i_h^i \cdot i_w^i}{ich_i^{par} \cdot ow_i^{par}}, \tag{4.9}$$

with i_h^i and i_w^i representing the input height and width of the layer.

Resource utilization is also tied to the parallelization factors. The number of DSPs required is:

$$D_i = \frac{och_i^{par} \cdot ow_i^{par} \cdot ic_i^{par} \cdot f_h^i \cdot f_w^i}{pack_i}, \tag{4.10}$$

where $pack_i$ indicates the number of operations packed into a single DSP (e.g., two 8-bit multiply-accumulate operations or four 4-bit operations). Similarly, the memory bank requirements M_i depend on both the filter size and the degree of parallelism, and will be detailed in Section 4.3.

This formulation shows the main trade-off: increasing parallelism will reduce latency but increases the consumption of DSPs and memory. Therefore, DSE must determine the optimal set of parallelization factors $\{ow_i^{par}, och_i^{par}, ich_i^{par}\}$ across all layers, so that throughput is maximized while respecting the device constraints.

4.2 Parallelization Strategies

To take advantage of the parallel nature of CNNs, operations are typically accelerated by unrolling nested loops in the computation kernel. In nn2FPGA three independent parallelization dimentions are considered: *output width*, *output channels*, and *input channels*. Each axis directly influences resources and the latency of the corresponding hardware tasks.

4.2.1 Output Width Parallelism

Output width parallelism (`owpar`) refers to the number of computations of multiple output activations in the same row of the output feature map. This strategy is based

on the reuse of the same filter weights in multiple positions, reducing the effective memory bandwidth required to load parameters. It also increases the size of the window buffer, since more input windows must be kept in parallel. The buffer must be partitioned accordingly to achieve higher throughput.

From the latency model, output width parallelism contributes linearly to the overall computation parallelism:

$$c_i^{par} \propto ow_i^{par}. \tag{4.11}$$

At the same time, the required memory resources scale with the number of parallel windows:

$$M_i^w \propto ow_i^{par}. \tag{4.12}$$

Hence, `owpar` represents a trade-off between reduced parameter bandwidth and increased buffer cost.

4.2.2 Output Channel Parallelism

Output channel parallelism (`ochpar`) refers to the concurrent computation of multiple filters applied to the same input window. Each filter produces an independent output channel and so this strategy increases the memory bandwidth required for weights, but does not affect the buffer size for activations. This means that the same input window is reused across all output channels, making `ochpar` highly efficient when sufficient DSPs are available.

Formally, its contribution to computation parallelism is:

$$c_i^{par} \propto och_i^{par}. \tag{4.13}$$

The required number of DSPs grows proportionally:

$$D_i \propto och_i^{par}. \tag{4.14}$$

Thus, `ochpar` is typically limited by the availability of DSPs and the memory bandwidth needed to sustain multiple filters.

4.2.3 Input Channel Parallelism

Input channel parallelism (`ichpar`) takes advantage of the fact that each convolutional output is the sum of contributions across all input channels. By processing multiple channels simultaneously, the number of cycles required per output window is reduced. Unlike `owpar` and `ochpar`, which replicate the convolutional datapath, `ichpar` naturally enables the use of DSP chaining, where the accumulator of one DSP feeds directly into another. This reduces resource overhead by reusing internal adders.

Its impact on computation parallelism is:

$$cpar_i \propto ich_i^{par}, \tag{4.15}$$

while the number of DSPs grows as:

$$D_i \propto ich_i^{par}. \tag{4.16}$$

However, since contributions from multiple channels must be accumulated, longer DSP chains increase pipeline depth, slightly impacting latency ε in Equation 4.2. This makes `ichpar` particularly effective when combined with quantization-aware optimizations such as DSP packing.

4.2.4 Combined Parallelization

In practice, the parallelization strategy for a given layer is determined by the product of all three factors:

$$c_i^{par} = ow_i^{par} \cdot och_i^{par} \cdot ich_i^{par} \cdot f_h^i \cdot f_w^i, \tag{4.17}$$

where f_h^i and f_w^i are fixed by the network architecture. Different combinations of $\{ow_i^{par}, och_i^{par}, ich_i^{par}\}$ yield the same level of parallelism but may have very different implications for resource usage. For example:

- Increasing `owpar` reduces weight bandwidth and increases buffer memory.
- Increasing `ochpar` increases the weight bandwidth but reuses input activations efficiently.
- Increasing `ichpar` allows efficient accumulation through DSP chaining but deepens the pipeline.

The choice of parallelization factors therefore constitutes a multi-objective optimization problem, balancing throughput maximization with constraints on DSPs, BRAMs, and memory bandwidth. In Section 4.3, this problem is formalized as a Binary Integer Programming model that systematically explores all feasible combinations to identify the optimal allocation strategy.

4.3 Binary Integer Programming Model

The parallelization strategies described in Section 4.2 define a vast design space, as each convolutional layer can be mapped to multiple feasible combinations of $\{ow_i^{par}, och_i^{par}, ich_i^{par}\}$. To identify the optimal configuration across all layers while respecting hardware resource limits, the problem is formulated as a BIP model.

4.3.1 Decision Variables

For each layer $i \in G$, a set of feasible configurations $\mathscr{C}_i$ is generated. Each configuration $k \in \mathscr{C}_i$ specifies a tuple of parallelization factors:

$$k = \{ow_{ik}^{par},\ och_{ik}^{par},\ ich_{ik}^{par}\}.$$

A binary variable $x_{i,k}$ is associated to each configuration:

$$x_{i,k} = \begin{cases} 1, & \text{if configuration } k \text{ is selected for layer } i, \\ 0, & \text{otherwise.} \end{cases}$$

By construction, exactly one configuration must be selected per layer:

$$\sum_{k \in \mathscr{C}_i} x_{i,k} = 1, \quad \forall i \in G. \tag{4.18}$$

4.3.2 Layer Latency and Resource Usage

Each configuration k defines a computation latency $L^c_{i,k}$, a window buffer latency $L^w_{i,k}$, a number of required DSPs $D_{i,k}$, and memory banks $M_{i,k}$. These are pre-computed using the equations from Section 4.2.

The effective latency and resource usage for each layer are expressed as:

$$L^c_i = \sum_{k \in \mathscr{C}_i} L^c_{i,k} \cdot x_{i,k}, \tag{4.19}$$

$$L^w_i = \sum_{k \in \mathscr{C}_i} L^w_{i,k} \cdot x_{i,k}, \tag{4.20}$$

$$D_i = \sum_{k \in \mathscr{C}_i} D_{i,k} \cdot x_{i,k}, \tag{4.21}$$

$$M_i = \sum_{k \in \mathscr{C}_i} M_{i,k} \cdot x_{i,k}. \tag{4.22}$$

4.3.3 Objective Function

The throughput of the entire accelerator is limited by the slowest concurrent task, i.e., the maximum latency across all computation and buffering tasks. To linearize this minimax objective, we introduce an auxiliary variable S representing the pipeline period. The objective is to minimize S:

$$\min S \tag{4.23}$$

subject to the following constraints:

$$L^c_i \leq S, \quad \forall i \in G, \tag{4.24}$$

$$L^w_i \leq S, \quad \forall i \in G. \tag{4.25}$$

4.3.4 Resource Constraints

The total number of DSPs and memory banks used by all layers must not exceed the available hardware resources:

$$\sum_{i \in G} D_i \leq DSP, \tag{4.26}$$

$$\sum_{i \in G} M_i \leq MEM. \tag{4.27}$$

These constants are taken directly from the board datasheet and provided to the compiler as inputs.

4.3.5 Complete BIP Formulation

The full problem can therefore be summarized as:

$$\begin{aligned}
\min \quad & S && && (4.28)\\
\text{subject to:} \quad & \sum_{k \in \mathscr{C}_i} x_{i,k} = 1, && \forall i \in G, && (4.29)\\
& L_i^c \leq S,\ L_i^w \leq S, && \forall i \in G, && (4.30)\\
& \sum_{i \in G} D_i \leq DSP, && && (4.31)\\
& \sum_{i \in G} M_i \leq MEM, && && (4.32)\\
& x_{i,k} \in \{0,1\}, && \forall i \in G,\ k \in \mathscr{C}_i. && (4.33)
\end{aligned}$$

This formulation guarantees that the selected parallelization factors maximize throughput by minimizing the slowest task latency, while respecting the hardware constraints of the target FPGA. The resulting problem is an instance of binary integer programming, which can be efficiently solved using modern solvers such as Gurobi or CPLEX, given the relatively small number of layers in typical CNNs.
The nn2FPGA compiler uses the PULP [44] library to solve binary integer programming problems with its default solver that provides the optimal allocation of resources across the network graph.

4.4 Optimization Flow

The BIP formulation presented in Section 4.3 provides a mathematically rigorous way to determine the optimal allocation of FPGA resources across the layers of a CNN. However, in practice, several additional steps are required to generate a feasible design within the nn2FPGA compilation framework. The optimization process can be divided into three main phases: *enumeration of feasible configurations*, *throughput maximization*, and *resource minimization*.

4.4.1 Enumeration of Feasible Configurations

The design space is initially generated by enumerating all possible parallelization factors $\{ow_i^{par}, och_i^{par}, ich_i^{par}\}$ for each convolutional or pooling layer $i \in G$. For each candidate configuration, the following metrics are pre-computed:

- computation latency $L^c_{i,k}$,
- window buffer latency $L^w_{i,k}$,
- number of required DSPs $D_{i,k}$,
- memory bank requirements $M_{i,k}$.

Configurations that do not respect hardware limits (e.g., exceeding available DSPs or memory ports) are discarded at this stage. This step ensures that only valid design points are considered in subsequent optimization.

Unroll factors are restricted to integers divisors of their respective dimension. From theory it is known that the number of distinct divisors for most integers n is lower than $log(n)$ [45]. Since we are taking into account tuples of three elements, the number of potential combinations grows in the worst case as $log(n)^3$ which is less than linear with respect to the input size. Hence the search space is limited and is further bounded by the actual number of resources available on the boards. Thus complete enumeration is feasible in practice for real CNNs and real FPGA, as shown in table 4.1.

Table 4.1 Binary integer programming solver execution time.

Model	Board	Bit	Variables	Constraints	Time (s)
ResNet8	KV260	4	523	37	0.10
ResNet8	Ultra96	8	407	37	0.06
ResNet8	KV260	8	523	37	0.07
ResNet20	Ultra96	8	1107	85	0.09
ResNet20	KV260	8	1471	85	0.94
Mobilenet	ZCU102	8	6142	217	0.97

4.4.2 Throughput Maximization

The feasible configurations are then passed to the BIP solver. The first optimization phase aims to *minimize the pipeline period* S, corresponding to the slowest concurrent task in the accelerator. By solving the model defined in Equations (4.28–4.33), the compiler identifies the combination of parallelization factors across all layers that maximizes throughput while respecting resource constraints.

4.4.3 Resource Minimization for Non-Critical Layers

The solution obtained from the throughput maximization phase often over-parallelizes layers that are not critical to the overall pipeline period. To reduce unnecessary resource usage, a second optimization step is performed. In this phase, the parallelism of non-bottleneck layers is progressively decreased, as long as their latency remains below the global period S. This reduces the overall consumption of DSPs and BRAMs, facilitating placement and routing in the backend flow and improving the likelihood of achieving timing closure at high clock frequencies.

4.4.4 Integration with the Compiler Toolchain

The optimized configuration is then integrated into the nn2FPGA toolchain. The chosen parallelization factors for each layer are translated into synthesis directives (e.g., loop unrolling pragmas, array partitioning, and pipeline depth constraints) in the generated HLS code. The design is synthesized with Vitis HLS and mapped to the target FPGA using Vivado. The post-optimization step significantly improves

the robustness of the flow, as over-constrained designs often fail in implementation due to excessive resource usage or routing congestion.

4.4.5 Summary of the Flow

To summarize, the optimization flow consists of the following pipeline:

1. Enumerate all feasible parallelization factors for each layer.
2. Solve the BIP model to minimize the pipeline period S maximizing global throughput.
3. Refine the solution by reducing non-critical parallelism to free unused resources.
4. Automatically generate the HLS code of the network with the selected configuration and synthesize the design.

This two-phase optimization strategy makes it possible to generate high-performance accelerators that fully utilize the available computational resources of the target FPGA, while keeping resource usage under control and easing backend implementation.

4.5 Discussion

The resource allocation and DSE methodology described in this chapter show the importance of combining mathematical optimization with hardware-aware modeling. The two-phase strategy based on BIP ensures that accelerator throughput is maximized by minimizing the latency of the slowest task, while the subsequent step avoids resource over-consumption in layers that are not critical to the pipeline. This balance is essential, as an aggressive parallelization of all layers would exceed the available resources on the target FPGA, making the design infeasible.

A key observation is that different layers of a CNN exhibit highly heterogeneous computational characteristics. Early convolutional layers typically operate on large input feature maps with relatively few channels, whereas deeper layers have small

input dimensions but a much larger number of channels. This heterogeneity implies that a uniform allocation of parallelization factors across layers is inherently suboptimal. By adapting the parallelism to each layer's structure, the proposed approach achieves a better distribution of DSPs and memory, avoiding situations where resources are wasted in non-bottleneck layers while other layers limit the overall throughput.

Another important aspect is the scalability of the optimization process. Although the problem is formulated as a BIP, the number of decision variables remains moderate because the number of layers in common CNNs is limited and the enumeration phase prunes infeasible configurations in advance. As a result, modern solvers can find the optimal solution in seconds or minutes, which is acceptable in the context of a compilation flow that is typically dominated by HLS synthesis and backend implementation. This makes the methodology practical for real-world deployment, where the design cycle may involve multiple iterations.

The refinement phase plays a crucial role in ensuring backend feasibility. Over-parallelization of non-critical layers can lead not only to wasted DSPs and BRAMs, but also to routing congestion and reduced maximum operating frequency. By minimizing parallelism where it is not needed, the design becomes more compact and regular, easing placement and routing and improving the probability of meeting timing closure at high frequencies. This step highlights the necessity of considering not only the theoretical throughput but also the practical limitations of FPGA implementation.

Finally, the methodology underlines a general principle in accelerator design: optimization must simultaneously account for performance, resource constraints, and implementation feasibility. A model that focuses exclusively on throughput may produce designs that cannot be implemented, while an overly conservative approach may waste valuable computational potential. The binary integer programming formulation, combined with the refinement strategy, provides a structured way to explore this trade-off, ensuring that the generated accelerators are both efficient and realizable.

Chapter 5

Experimental Evaluation

5.1 Setup

The accelerators generated by nn2FPGA are evaluated on the CIFAR-10 and Imagenet datasets, that exemplify common embedded machine vision applications. The former consists of 32×32 RGB images, while the latter of 224×224 RGB images. CNN models targeting CIFAR-10 are trained using Brevitas QAT for 400 epochs with a batch size of 256, using the Stochastic Gradient Descent (SGD) optimizer and cosine annealing as the learning rate scheduler. The MobileNetV2 model is based on a pre-trained model and it is quantized using Brevitas PTQ, without performing fine-tuning. QAT is not used in this case because it requires very significant computational resources during training, and the focus of this work is on efficient HLS implementation of a given quantized network, rather than on network architecture exploration or quantization. The implementation flow uses Xilinx Vitis HLS 2024.2 for RTL code generation and Vivado 2024.2 for implementation on the Ultra96-v2, Kria KV260 and ZCU102 boards. Table 5.1 details the available resources for these three boards.

Table 5.1 Resources of the boards used for the experiments.

Board	FPGA part	LUT	FF	BRAM	DSP	URAM
Ultra96	xczu3eg	70560	141120	216	360	0
Kria KV260	xczu5eg	117120	234240	144	1248	64
ZCU102	xczu9eg	274080	548160	912	2520	0

The *nn2FPGA* compiler uses the PULP [44] library to solve binary integer programming problems with its default solver. Table 4.1 shows the execution time of the solver for various combinations of model and board, and the reported times are contained.

5.2 Results and Comparisons

The inference throughput (Frame Per Second (FPS), Gops/s) and latency (ms) measured on the boards are detailed in table 5.2, while the final resource utilization is reported in table 5.3. Both tables show highlighted in bold the results of *nn2FPGA*.

In the first section of table 5.2, the *nn2fpga* implementation of ResNet20 is compared with the AdderNet and ResNet20 implementations presented in [46], both running on the Kria KV260. The Addernet is included in this comparison since, from the point of view of the architecture, it is a ResNet20 with multiplications replaced by additions. Our implementation achieves throughput improvements (Gops/s) of 2.88× and 1.94× with respect to the ResNet20 and the Addernet in [46]. Also, the latency is reduced by 3.84× and 1.96× respectively.

The results compare on KRIA KV260 the *nn2FPGA* ResNet8 with the implementation from Vitis AI [47], achieving a throughput improvement of 6.8× and a latency improvement of 28.1×. The third section of table 5.2 shows the results of a 4-bit implementation of a ResNet8, compared to the FINN version detailed in [47]. The comparison shows how *nn2fpga* is able to efficiently use FPGA resources even at lower bit precision, achieving a speedup of 4.53× and improving latency by 4.27×.

To further prove the flexibility of *nn2FPGA*, table 5.2 also reports the performance of ResNet8 and ResNet20 deployed on the smaller Ultra96. There are no other implementations of these models on this board in the literature for comparison.

Finally, the implementation of MobileNetV2 on the ZCU102 is compared with the Vitis AI implementation [48], with a custom RTL model [49] and with an implementation based on fpgaConvNet framework [50]. The implementation achieves a throughput of 2115 fps with a frequency of 214 MHz, and an accuracy of 71.73 %. This represents a 10% speedup over a state-of-the-art highly optimized manual RTL implementation, showing that HLS can actually improve over manual design, thanks to the faster exploration of the design space. While the maximum frequency achieved

is lower than that proposed by [49], the solution demonstrates higher overall throughput, due to better utilization of the available resources provided by the ZCU102, as shown by the high DSP utilization achieved. The slight difference in accuracy of the models is attributable to the different quantization methods used, as well as the accuracy of the pre-quantization model.

The comparison includes FINN, Vitis AI, fpgaConvNet and custom RTL accelerators as baselines because these designs provide published results on the same FPGA boards and datasets used in this work and adopt similar fixed-point quantization schemes. Moreover, the selected baselines target the same or closely related network architectures (ResNet and MobileNetV2), enabling a fair, architecture- and device-matched evaluation. Works relying on floating-point arithmetic or on substantially larger FPGA devices are excluded from the quantitative tables, as such differences would dominate the comparison and obscure the effect of the proposed methodology.

Table 5.2 Performance comparison with SOTA accelerators. Power was measured using sensors on the power rails and is representative of the entire board consumption. Throughput was evaluated with a batch size of 400 for ImageNet and 1000 for CIFAR10.

Model	Dataset	FPGA	Bit	Freq. (MHz)	Throughput (FPS)	Throughput (Gops/s)	Latency (ms)	Power (W)	Accuracy (%)
ResNet20† [46]	CIFAR10	KV260	8	200	N/A	214	1.221	1.07†	90.8
AdderNet† [46]	CIFAR10	KV260	8	200	N/A	317	0.624	1.52†	89.9
ResNet20	CIFAR10	KV260	8	250	7601	616	0.318	6.10	91.3
ResNet8 Vitis AI [47]	CIFAR10	KV260	8	200	4458	109	1.293	6.42	89.2
ResNet8	CIFAR10	KV260	8	250	30153	773	0.046	6.67	88.7
ResNet8 FINN [47]	CIFAR10	KV260	4	225	13475	330	0.154	5.89	85.9
ResNet8	CIFAR10	KV260	4	250	61035	1526	0.036	6.12	86.9
ResNet20	CIFAR10	Ultra96	8	214	3254	264	0.807	1.04	91.3
ResNet8	CIFAR10	Ultra96	8	214	12971	317	0.111	0.56	88.7
MobileNetV2 Vitis AI [48]	ImageNet	ZCU102	8	281	765	N/A	N/A	N/A	67.67
MobileNetV2 [49]	ImageNet	ZCU102	8	333	1910	N/A	N/A	N/A	72.98
MobileNetV2 [50]	ImageNet	ZCU102	4	200	N/A	N/A	2.3	N/A	65.70
MobileNetV2	ImageNet	ZCU102	8	214	2115	1403	2.061	13.5	71.73

† The description of how to measure power consumption is not explicitly provided in [46]. Accounting for just the board idle consumption measured with `xmutil`, such a measure is already above the values reported in the referenced paper.

To demonstrate the efficiency of the proposed graph optimizations beyond their dataflow benefits, an ablation study is provided to highlight the reduction in on-chip activation storage required for the tested networks. table 5.4 shows the resource used to store activations depending on the chosen implementations. ResNets can leverage both optimizations thanks to the presence of both downsample and no-downsample types of residual bottlenecks. In contrast, MobileNetV2 can only benefit from

Table 5.3 Resource utilization comparison. Reported data are collected from the Vivado report after place and route.

Model	FPGA	Bit	kLUT	kLUTRAM	kFF	DSP	BRAM	URAM
ResNet20 [46]	KV260	8	41.8 (35.7 %)	17.6 (30.1 %)	34.0 (14.5 %)	545 (43.7 %)	40.0 (27.7 %)	N/A
AdderNet [46]	KV260	8	67.4 (57.6 %)	22.2 (38.6 %)	43.2 (19.1 %)	609 (48.8 %)	40.0 (27.7 %)	N/A
ResNet20	KV260	8	65.0 (55.5 %)	13.0 (22.6 %)	81.7 (34.9 %)	636 (51.0 %)	60.5 (42.0 %)	12 (18.7 %)
ResNet8 VITIS AI [47]	KV260	8	25.6 (21.8 %)	N/A	33.7 (14.4 %)	110 (8.8 %)	8.8 (8.8 %)	18 (28.1 %)
ResNet8	KV260	8	55.4 (47.3 %)	9.4 (16.4 %)	70.1 (29.9 %)	767 (61.5 %)	63.5 (44.1 %)	0 (0 %)
ResNet8 FINN [47]	KV260	4	81.4 (69.5 %)	N/A	87.6 (37.4 %)	N/A	28.5 (19.8 %)	N/A
ResNet8	KV260	4	40.4 (34.5 %)	10.3 (17.9 %)	64.8 (27.7 %)	794 (63.6 %)	58.0 (40.3 %)	0 (0 %)
ResNet20	Ultra96	8	54.4 (77.1 %)	10.2 (35.6 %)	57.6 (40.8 %)	318 (88.3 %)	89.5 (41.4 %)	0 (0 %)
ResNet8	Ultra96	8	46.4 (65.8 %)	6.2 (21.5 %)	45.1 (32.0 %)	360 (100 %)	54.0 (25 %)	0 (0 %)
MobileNetV2 VITIS AI [48]	ZCU102	8	N/A	N/A	N/A	N/A	N/A	N/A
MobileNetV2 [49]	ZCU102	8	170.4 (62.2 %)	N/A	154.3 (28.2 %)	1283 (50.9 %)	691 (75.8 %)	0 (0 %)
MobileNetV2 [50]	ZCU102	4	273 (99 %)	N/A	N/A	1222 (48.5 %)	714 (78.3 %)	0 (0 %)
MobileNetV2	ZCU102	8	139.4 (50.9 %)	63.3 (44.0 %)	162.8 (29.7 %)	1797 (71.3 %)	912 (100 %)	0 (0 %)

Temporal Reuse, as its inverted residual bottleneck does not contain convolutions in the short branch. Moreover, Temporal Reuse has a limited impact on resource savings for the inverted residual bottleneck, since the first convolution in the block is point-wise and thus does not increase the receptive field's dimension.

Table 5.4 On-chip activation resource reduction achieved through various graph optimizations. ResNets are generated for KRIA KV260, while MobileNetV2 for ZCU102. All kernels are synthesized at 300MHz in out-of-context mode, with results obtained from the synthesis report of Vivado.

	No opt.			Temporal Reuse			Loop Merge			Temporal Reuse + Loop Merge		
Model	kLUT	kFF	BRAM	kLUT	kFF	BRAM	kLUT	kFF	BRAM	kLUT	kFF	BRAM
ResNet8	14.7	11.2	8.5	14.4	11.2	8.5	14.2	11.1	7.5	13.3 (-9.5%)	11.1 (-0.9%)	7.5 (-11.7%)
ResNet20	27.2	18.1	8.5	25.9	17.8	6.0	26.7	17.9	8.5	25.6 (-6.0%)	17.7 (-2.2%)	6.0 (-29.4%)
MobileNetV2	82.0	44.1	102.5	79.7	43.7	102.5	82.0	44.1	102.5	79.7 (-2.8%)	43.7 (-0.9%)	102.5 (-0.0%)

5.2.1 Object Detection with YOLOv5n

To further validate the generalization capability of the proposed framework beyond classification tasks, a second set of experiments was conducted on the YOLOv5n [51] architecture for real-time object detection. This work was carried out in the framework of the European project *REBECCA*[1], which focused on the development of energy-efficient embedded AI solutions for critical infrastructure monitoring. The model was deployed on a Xilinx ZCU102 board, targeting multiple application

[1]This activity was partially supported by the EU Horizon project REBECCA (Reconfigurable Heterogeneous Highly Parallel Processing Platform for safe and secure AI)

provided by partners within the project, including the detection of photovoltaic panel anomalies and the identification of structural cracks in bridges.

The baseline YOLOv5n model was taken from the Ultralytics repository and quantized using the *Vitis AI Quantizer* in post-training quantization (PTQ) mode, before a few fine-tuning epochs to adapt on the condidered datasets and recover the potential accuracy loss. The primary goal of this evaluation is the analysis of the hardware implementation and timing rather than training performance so accuracy metrics are omitted. All networks share identical topology and layer configuration; therefore, resource utilization is the same independently from the specific dataset.

Compiler and Architectural Extensions. To support the heterogeneous topology of YOLOv5n, some extensions were implemented in the compiler and HLS library:

- **Concatenation layers:** support was added for multiple input streams, including concatenation nodes with more than two branches, ensuring synchronization between feature maps of different dimentions.
- **Upsampling layers:** implemented as nearest-neighbor, the module stores incoming pixels in line/row buffers; each pixel is read back multiple times for horizontal duplication, and each completed row is replayed to generate the vertically duplicated row. This design is *zero-DSP* and its memory footprint scales with feature-map width and channel count, contributing to overall usage of BRAMs.
- **Activation functions:** Leaky ReLU and Sigmoid were implemented as LUT-based interpolators saving coefficients at compile time ensuring deterministic latency and minimizing control overhead.
- **Multiple output heads:** the dataflow was extended to generate multiple detection outputs simultaneously with each head independently streaming its results.

Post-processing stages, including bounding box decoding, confidence thresholding, and Non-Maximum Suppression (NMS), were executed on the Processing System (PS), while all convolutional, upsampling, and concatenation operations were synthesized in the Programmable Logic (PL).

Experimental Results. Table 5.5 reports the post-implementation resource utilization of the YOLOv5n accelerator on the ZCU102 board at 8-bit precision. The design achieved stable timing with 83 FPS at 200 MHz. The results highlight that the overall BRAM usage is dominated by FIFOs for streaming communication between concurrent tasks. A significant fraction of these FIFOs are conservatively over-dimensioned, suggesting room for improvement through analytical buffer sizing in future versions of the compiler.

Table 5.5 Resource utilization of the YOLOv5n accelerator on ZCU102 (8-bit quantization).

Model	FPGA	LUTs	FFs	BRAM	DSPs	URAM
YOLOv5n	ZCU102	209.1k (60.7%)	212.2k (38.7%)	857.5 (94.0%)	672 (26.7%)	0 (0%)

An inspection of the post-synthesis reports revealed that approximately 60 % of the LUTs are used as logic, while nearly 30 % are configured as distributed memory. The high BRAM occupation is mainly due to the streaming FIFOs which tend to be oversized for throughput safety. Nevertheless, the design achieves consistent utilization across logic and DSP resources, maintaining balanced operation and no routing congestion. These results confirm that the proposed compiler extensions enable the deployment of complex multi-branch architectures such as YOLOv5n, validating the scalability of the nn2FPGA flow towards detection tasks and heterogeneous workloads within the REBECCA project.

5.3 Summary

Across three devices and multiple models, nn2FPGA delivers consistent gains over similar flows: up to $6.8\times$ higher throughput and $28.1\times$ lower latency on CIFAR-10 ResNet8 versus Vitis AI, $4.53\times$ speedup over FINN at 4-bit on KV260, significant improvements over literature ResNet/AdderNet baselines, and a $\sim$10 % throughput uplift over a state-of-the-art hand-optimized RTL MobileNetV2 on ZCU102. These improvements arise from the BIP-driven, two-stage allocation combined with graph-level optimizations that lower activation buffering, all validated by the utilization profiles in Table 5.3 and the ablation in Table 5.4.

Chapter 6

Conclusions and Future Works

6.1 Summary of Contributions

This work presented nn2FPGA, a complete high-level compilation framework for the automated deployment of quantized CNNs onto FPGAs. The proposed methodology bridges the gap between deep neural network design and hardware synthesis, combining high-level graph analysis, quantization awareness, and binary integer programming (BIP)–based resource allocation.

Unlike traditional HLS toolflows or template-based accelerators, nn2FPGA formulates the mapping problem as a global optimization task on the network graph. The method enumerates feasible parallelization factors per layer and solves a constrained optimization problem that maximizes throughput under available device resources. A second optimization step then reduces redundant parallelism in non-bottleneck layers, improving design feasibility and backend implementation success rate. This two-step approach implemented through an efficient BIP formulation represents one of the central technical contributions of this thesis.

The compiler integrates architectural-level optimizations such as *Temporal Reuse* and *Loop Merge*, which exploit residual-block structure to minimize on-chip activation buffering. These graph-level transformations, discussed in Chapter 3, complement the dataflow allocation strategy, jointly reducing memory footprint while sustaining full computational throughput.

The framework targets standard HLS tools (Vitis HLS 2024.2) and is portable across multiple Xilinx SoC-FPGA platforms. Experimental validation on CIFAR-10 and ImageNet demonstrated consistent and reproducible improvements in performance and efficiency compared to academic and industrial baselines.

6.2 Experimental Findings

As shown in Chapter 5, benchmarking on Ultra96-V2, Kria KV260, and ZCU102 boards confirmed the effectiveness of the proposed approach:

- nn2FPGA outperformed Vitis AI and FINN on CIFAR-10, by up to 6.8× in throughput and 28.1× in latency, while preserving accuracy within 1 % of the software baselines.
- Compared to hand-optimized designs such as AdderNet and custom RTL implementations, nn2FPGA achieved between 1.9× and 3.8× lower latency, proving that systematic compiler-driven optimization can rival or surpass manual hardware design.
- The MobileNetV2 implementation on ZCU102 reached 2115 FPS at 214 MHz, a 10 % throughput gain over a state-of-the-art handcrafted RTL accelerator, thus demonstrating the viability of HLS-based synthesis for complex architectures.

Ablation experiments confirmed that graph-level optimizations substantially reduced activation memory requirements (up to 29 % for ResNet20) without affecting throughput, validating the analytical models proposed in Chapter 4. In general, the results highlight that the combination of algorithmic analysis with formal resource optimization provides both scalability and portability across devices.

6.3 Limitations

Although the proposed framework delivers state-of-the-art results, several limitations remain:

- The compiler does not explore mixed-precision quantization or dynamic fixed-point scaling, which eventually improve the accuracy and efficiency trade-off for large models.
- Training-based feedback (e.g., co-design loops between quantization and hardware mapping) is not integrated; this restricts the optimization to inference-only flows.
- The backend remains tied to Vitis HLS and Vivado; integrating vendor agnostic IRs or integrating open-source toolchains portability and reproducibility could be improved.

6.4 Future Work

Based on the results of this research, several promising extensions can be envisaged.

6.4.1 Mixed-Precision and Adaptive Quantization

nn2FPGA could support per-channel quantization and/or not only power-of-two scaling factor, automatically balancing accuracy and hardware cost. This could be achieved by integrating DSE loops with quantization-aware training feedback to enable end-to-end optimization of precision, dataflow, and resource allocation.

6.4.2 Model-Generalization Beyond CNNs

The two-phase optimization strategy presented in this work first maximizing throughput and then refining resource allocation has demonstrated its effectiveness for convolutional neural networks, where computation and data reuse patterns are highly regular. In order to extend this methodology to a broader class of models, a compelling research direction is required. Future developments should aim to generalize the compiler flow to support architectures that deviate from the canonical convolutional structure, including transformer-based models, recurrent networks, and graph neural networks.

These models introduce different computational characteristics, such as irregular data dependencies, variable-length operations and attention mechanisms that require dynamic memory access and fine-grained scheduling. Adapting the optimization flow to these paradigms will involve extending the current cost models to capture different forms of parallelism and temporal data reuse. For instance, the binary integer programming formulation could be expanded to represent variable tensor dimensions or sparsity patterns enabling the compiler to reason about static and dynamic dataflows.

Generalizing the nn2FPGA methodology beyond CNNs would enable the automatic generation of accelerators for a wide range of machine learning workloads. By abstracting the underlying optimization principles rather than the specific layer types, future versions of the framework could target not only vision tasks but also applications such as natural-language processing, graph analytics, and scientific computing, preserving the same balance between flexibility, efficiency, and hardware awareness that characterizes the present work.

6.5 Final Remarks

This thesis presents a systematic optimization and formal resource allocation to transform HLS design from a manual, heuristic process into an automated, performance-driven compilation flow. nn2FPGA integrates analytical modeling, integer programming, and graph-level optimizations into a coherent methodology, generating both efficient and portable hardware accelerators. The results achieved across multiple devices prove that HLS can not only approximate but sometimes exceed the performance of hand-optimized RTL designs when guided by structured optimization.

The concepts and tools developed here are expected to serve as a foundation for future research in compiler-assisted hardware–software co-design, contributing toward a new generation of flexible and high-performance neural network accelerators.

References

[1] Xilinx Inc. *Vitis High-Level Synthesis User Guide*, 2022.

[2] Alessandro Pappalardo. Xilinx/brevitas, 2022.

[3] Zewen Li, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. A survey of convolutional neural networks: Analysis, applications, and prospects. *IEEE Transactions on Neural Networks and Learning Systems*, 33(12):6999–7019, 2022.

[4] Evan Shelhamer, Jonathan Long, and Trevor Darrell. Fully convolutional networks for semantic segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(4):640–651, 2017.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[6] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.

[7] Jiasi Chen and Xukan Ran. Deep learning with edge computing: A review. *Proceedings of the IEEE*, 107(8):1655–1674, 2019.

[8] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[9] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, page 26–35. Association for Computing Machinery, 2016.

[10] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. Optimizing the convolution operation to accelerate deep neural networks on fpga. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(7):1354–1367, 2018.

[11] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, page 161–170, New York, NY, USA, 2015. Association for Computing Machinery.

[12] Jie Hu, Li Shen, Samuel Albanie, Gang Sun, and Enhua Wu. Squeeze-and-excitation networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(8):2011–2023, 2020.

[13] Anouar Nechi, Lukas Groth, Saleh Mulhem, Farhad Merchant, Rainer Buchty, and Mladen Berekovic. Fpga-based deep learning inference accelerators: Where are we standing? *ACM Trans. Reconfigurable Technol. Syst.*, 16(4), October 2023.

[14] S. Srilakshmi and G.L. Madhumati. A comparative analysis of hdl and hls for developing cnn accelerators. In *2023 Third International Conference on Artificial Intelligence and Smart Energy (ICAIS)*, pages 1060–1065, 2023.

[15] Mohamed S. Abdelfattah, David Han, Andrew Bitar, Roberto DiCecco, Shane O'Connell, Nitika Shanker, Joseph Chu, Ian Prins, Joshua Fender, Andrew C. Ling, and Gordon R. Chiu. Dla: Compiler and fpga overlay for neural network inference acceleration. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 411–4117, 2018.

[16] Lucian Petrica, Tobias Alonso, Mairin Kroes, Nicholas Fraser, Sorin Cotofana, and Michaela Blott. Memory-efficient dataflow inference for deep cnns on fpga. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, pages 48–55, 2020.

[17] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[18] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.

[20] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[21] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[22] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2261–2269, 2017.

[23] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. *CoRR*, abs/1707.01083, 2017.

[24] Serena Curzel, Fabrizio Ferrandi, Leandro Fiorin, Daniele Ielmini, Cristina Silvano, Francesco Conti, Luca Bompani, Luca Benini, Enrico Calore, Sebastiano Fabio Schifano, Cristian Zambelli, Maurizio Palesi, Giuseppe Ascia, Enrico Russo, Valeria Cardellini, Salvatore Filippone, Francesco Lo Presti, and Stefania Perri. A survey on design methodologies for accelerating deep learning on heterogeneous architectures, 2025.

[25] Intel. *Intel® High Level Synthesis Compiler Reference Manual*, 2022.

[26] Cadence. *Stratus High-Level Synthesis*, 2022.

[27] Siemens. *Catapult C++/Systemc Synthesis*, 2022.

[28] Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, Marco Minutoli, Christian Pilato, and Antonino Tumeo. Invited: Bambu: an open-source research framework for the high-level synthesis of complex applications. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1327–1330. IEEE, Dec 2021.

[29] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz S. Czajkowski, Stephen Dean Brown, and Jason Helge Anderson. Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems. *ACM Trans. Embed. Comput. Syst.*, 13(2):24:1–24:27, 2013.

[30] Microchip. Smart high-level synthesis tool suite. https://www.microchip.com/en-us/products/fpgas-and-plds/fpga-and-soc-design-tools/smarthls-compiler, 2020.

[31] Hanchen Ye, HyeGang Jun, Hyunmin Jeong, Stephen Neuendorffer, and Deming Chen. ScaleHLS: A Scalable High-Level Synthesis Framework with Multi-Level Transformations and Optimizations. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*, pages 1355–1358, 2022.

[32] Nicolas Bohm Agostini, Serena Curzel, Ankur Limaye, Vinay Amatya, Marco Minutoli, Vito Giovanni Castellana, Joseph Manzano, Antonino Tumeo, and Fabrizio Ferrandi. The SODA Approach: Leveraging High-Level Synthesis for Hardware/Software Co-Design and Hardware Specialization. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*, pages 1359–1362, 2022.

[33] Michaela Blott, Thomas B Preußer, Nicholas J Fraser, Giulio Gambardella, Kenneth O'brien, Yaman Umuroglu, et al. FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks. *ACM Transactions on Reconfigurable Technology and Systems*, 11(3):1–23, 2018.

[34] Javier Duarte, Song Han, Philip Harris, Sergo Jindariani, Edward Kreinar, Benjamin Kreis, J Ngadiuba, M Pierini, N Tran, and Z Wu. Fast inference of deep neural networks in FPGAs for particle physics. *Journal of Instrumentation*, 13(07):P07027, 2018.

[35] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021.

[36] Stylianos I. Venieris and Christos-Savvas Bouganis. fpgaconvnet: Mapping regular and irregular convolutional neural networks on fpgas. *IEEE Transactions on Neural Networks and Learning Systems*, 30(2):326–342, 2019.

[37] Alessandro Pappalardo, Yaman Umuroglu, Michaela Blott, Jovan Mitrevski, Benjamin Hawks, Nhan Tran, Vladimir Loncar, Sioni Paris Summers, Hendrik Borras, Jules Muhizi, Matthew Trahms, Shih-Chieh Hsu, and Javier Mauricio Duarte. QONNX: Representing Arbitrary-Precision Quantized Neural Networks. In *4th Workshop on Accelerated Machine Learning (AccML) at HiPEAC 2022 Conference*, 6 2022.

[38] Roberto Bosio, Filippo Minnella, Teodoro Urso, Mario R. Casu, Luciano Lavagno, Mihai T. Lazarescu, and Paolo Pasini. Nn2fpga: Optimizing cnn inference on fpgas with binary integer programming. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 44(5):1807–1818, 2025.

[39] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[40] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2018.

[41] Giovanni Brignone, Roberto Bosio, Fabrizio Ottati, Claudio Sansoè, and Luciano Lavagno. Silvia: Automated superword-level parallelism exploitation via hls-specific llvm passes for compute-intensive fpga accelerators. *ACM Trans. Reconfigurable Technol. Syst.*, 18(2), March 2025.

[42] Ephrem Wu Yao Fu and Ashish Sirasao. 8-bit dot-product acceleration (wp487), 2017.

[43] Convolutional Neural Network with INT4 Optimization on Xilinx Devices, 2020.

[44] Iain Dunning, Stuart Mitchell, and Michael O'Sullivan. Pulp: A linear programming toolkit for python, 2011.

[45] G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers*. Oxford, fourth edition, 1975.

[46] Yunxiang Zhang, Biao Sun, Weixiong Jiang, Yajun Ha, Miao Hu, and Wenfeng Zhao. Wsq-addernet: Efficient weight standardization based quantized addernet fpga accelerator design with high-density int8 dsp-lut co-packing optimization. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '22, New York, NY, USA, 2022. Association for Computing Machinery.

[47] Fumio Hamanaka, Takashi Odan, Kenji Kise, and Thiem Van Chu. An exploration of state-of-the-art automation frameworks for fpga-based dnn acceleration. *IEEE Access*, 11:5701–5713, 2023.

[48] Vitis AI Model Zoo, 2023.

[49] Weixiong Jiang, Heng Yu, and Yajun Ha. A high-throughput full-dataflow mobilenetv2 accelerator on edge fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(5):1532–1545, 2023.

[50] Zhewen Yu and Christos-Savvas Bouganis. Auto ws: Automate weights streaming in layer-wise pipelined dnn accelerators. In *2024 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, 2024.

[51] Glenn Jocher, Alex Stoken, Jirka Borovec, NanoCode012, ChristopherSTAN, Liu Changyu, Laughing, tkianai, Adam Hogan, lorenzomammana, yxNONG, AlexWang1900, Laurentiu Diaconu, Marc, wanghaoyang0106, ml5ah, Doug, Francisco Ingham, Frederik, Guilhen, Hatovix, Jake Poznanski, Jiacong Fang, Lijun Yu , changyu98, Mingyu Wang, Naman Gupta, Osama Akhtar, PetrDvoracek, and Prashant Rai. ultralytics/yolov5: v3.1 - bug fixes and performance improvements, October 2020.