A

Project Report

Entitled

Design and Implementation of Efficient AI Accelerator for Drone Vision System

Submitted to the Department of Electronics Engineering in Partial Fulfilment for the Requirements for the Degree of

Bachelor of Technology

(Electronics and Communication)

: Presented & Submitted By :

Krish Patel, Dhairya Pandit, Aditya Chandel

Roll No. (U22EC013, U22EC017, U22EC070)

B. TECH. IV(EC), 7th Semester

: Guided By :

Dr. A. D. Darji

(Professor, DoECE)

(Year: 2025-26)

DEPARTMENT OF ELECTRONICS ENGINEERING

SARDAR VALLABHBHAI NATIONAL INSTITUTE OF TECHNOLOGY

Surat-395007, Gujarat, INDIA.

Sardar Vallabhbhai National Institute Of Technology

Surat - 395 007, Gujarat, India

DEPARTMENT OF ELECTRONICS ENGINEERING

CERTIFICATE

This is to certify that the Project Report entitled "**Design and Implementation of Efficient AI Accelerator for Drone Vision System**" is presented & submitted by Krish Patel, Dhairya Pandit, Aditya Chandel, bearing Roll No. U22EC013, U22EC017, U22EC070 of B.Tech. IV, 7th Semester in the partial fulfillment of the requirement for the award of B.Tech. Degree in Electronics & Communication Engineering for academic year 2025-26.

They have successfully and satisfactorily completed their **Project Exam** in all respects. We certify that the work is comprehensive, complete and fit for evaluation.

Dr. A. D. Darji
(Professor& Project Guide)

PROJECT EXAMINERS:

Name of Examiners	**Signature with Date**
1. Dr. A. D. Darji	
2. Dr. Z. M. Patel	
3. Dr. P. J. Engineer	
4. Dr. Sandeep Mishra	

Dr. Shilpi Gupta
Head, DoECE, SVNIT

Seal of The Department
(December 2025)

Acknowledgements

We would like to express our profound gratitude and deep regards to our guide Dr. A. D. Darji and our mentor Mr. Rinkesh Patel for their guidance. We are heartily thankful for suggestion and the clarity of the concepts of the topic that helped us a lot for this work. We would also like to thank Dr. Shilpi Gupta, Head of the Electronics Engineering Department, SVNIT and all the faculties of DoECE for their co-operation and suggestions. We are very much grateful to all our classmates for their support.

Krish Patel, Dhairya Pandit, Aditya Chandel
Sardar Vallabhbhai National Institute of Technology
Surat

December 2025

Abstract

This project presents a complete hardware–software co-design of a real-time monocular depth estimation and autonomous path planning system optimized for deployment on resource-constrained embedded platforms. A lightweight encoder–decoder network inspired by MobileNetV3-Small and AvioDepth is used for depth estimation, while the computationally intensive spatial filtering stage is accelerated using a custom FPGA-based fixed-function 8×8 convolution engine. The accelerator incorporates a pipelined Q8.8 multiply–accumulate (MAC) datapath, an efficient line-buffer–based sliding-window generator, and single-cycle FSM control, enabling real-time streaming of 224×224 images.

The FPGA convolution accelerator was validated using 1×1, 3×3, 5×5, and 7×7 kernels, achieving near bit-accurate matching with the Python golden model, with a pixel mismatch of only 0.00213%. Post-implementation results show extremely low hardware cost—only 209 LUTs and 203 registers, consuming less than 1% of FPGA resources. Power analysis confirms a low total on-chip power dominated by dynamic components, making the architecture ideal for battery-powered embedded vision systems.

For autonomous navigation, a 7-state path-planning FSM was developed to process the depth map and compute the optimal yaw angle for obstacle-free motion. The algorithm divides the scene into sectors, identifies navigable valleys, applies median smoothing, and computes a stable steering angle using fixed-point arithmetic. Hardware validation across 30 test cases shows excellent agreement between Python and Verilog outputs, with yaw angle error consistently below 0.05%, along with low power consumption for real-time deployment.

While the STM32N6 microcontroller integration was explored and the workflow understood, full deployment on the MCU was not performed in this project.

Overall, the proposed system achieves high numerical accuracy, extremely low FPGA resource usage, strong power efficiency, and provides a solid foundation for future integration with embedded MCUs. These results confirm the feasibility of deploying monocular depth estimation and autonomous path planning on compact FPGA hardware, enabling applications in small UAVs, ground robots, and low-power edge AI platforms.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviation	**Description**
AI	Artificial Intelligence
ARM	Advanced RISC Machine
BN	Batch Normalization
BRAM	Block Random Access Memory
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DSP	Digital Signal Processing / Processor
DWConv	Depthwise Convolution
EDM	Encoder–Decoder Model
FLOPs	Floating-Point Operations Per Second
FSM	Finite State Machine
FPGA	Field-Programmable Gate Array
GPIO	General-Purpose Input/Output
HW	Height–Width Dimensions
I/O	Input/Output
INT8/INT16	8-bit / 16-bit Integer Precision
IR	Inverted Residual Block
LUT	Look-Up Table
MAC	Multiply–Accumulate
MCU	Microcontroller Unit
MDE	Monocular Depth Estimation
MLP	Multi-Layer Perceptron
NPU	Neural Processing Unit
PW	Pointwise Convolution
Q1.15	1-bit Integer + 15-bit Fraction
Q8.8	8-bit Integer + 8-bit Fraction
RAM	Random Access Memory
ReLU	Rectified Linear Unit
RGB	Red, Green, Blue
SE	Squeeze-and-Excitation
STM32N6	STM32 Neural Processor MCU Series
TFLite	TensorFlow Lite
ULWA	Ultra-Lightweight Attention
UAV	Unmanned Aerial Vehicle
WFI	Wait For Interrupt
YAW	Rotational Angle Around Vertical Axis

Chapter 1
Introduction

Unmanned Aerial Vehicles (UAVs), commonly known as drones, have transformed the way humans perform aerial operations. From precision agriculture and environmental monitoring to infrastructure inspection and disaster response, UAVs are increasingly deployed in scenarios that require autonomy, high adaptability, and real-time decision-making. These platforms are equipped with sensors, cameras, and onboard processors that allow them to navigate complex environments without direct human intervention. As UAV applications grow in sophistication, there is a critical demand for onboard perception and navigation systems capable of estimating depth, detecting obstacles, and planning trajectories in real time, all under stringent power, weight, and latency constraints.

Conventional UAV operations largely depend on human operators or GPS-based navigation. While effective in controlled or outdoor environments, GPS signals may be unreliable or entirely unavailable in indoor spaces, dense urban areas, disaster-stricken zones, or environments with signal jamming. Relying solely on human pilots is also limiting, as human reaction times and cognitive processing cannot match the rapid decision-making requirements of autonomous UAVs flying at high speeds. This necessitates onboard intelligence capable of perceiving the environment, estimating depth from visual data, and executing obstacle-aware navigation autonomously.

1.1 Motivation

The motivation for this research stems from the need to develop UAV systems that are capable of real-time perception and navigation with high accuracy and low latency. Despite advances in deep learning and computer vision, many state-of-the-art perception algorithms are computationally intensive, relying on convolution-heavy or transformer-based models. Executing these algorithms on resource-constrained embedded hardware is challenging due to limitations in memory bandwidth, processing speed, and energy availability. Hardware accelerators, particularly FPGA- and microcontroller-based implementations, provide a solution by offering parallelism, deterministic execution, and energy-efficient computation.

The key motivations behind this work are as follows:

- **High-Speed Navigation:** Autonomous UAVs must react quickly to dynamic obstacles and changing environments. Low-latency computation is essential for safe flight.

- **Energy Efficiency:** UAVs are battery-powered, making power-efficient computation critical for extended mission duration.
- **Deterministic Execution:** Embedded hardware accelerators ensure predictable computation times, which is vital for real-time control loops.
- **Embedded Deployment:** Efficient hardware utilization allows complex AI models to run on microcontrollers or FPGA platforms without exceeding resource constraints.

1.2 Need for Autonomous Drone Vision Systems

Manual UAV operation is inherently limited in scalability, reliability, and environmental adaptability. Some of the key challenges include:

- **Human Operator Dependence:** Requires trained personnel for each UAV, increasing operational complexity and cost.
- **Latency in Decision Making:** Human reflexes are insufficient for high-speed obstacle avoidance or emergency maneuvers.
- **Communication Limitations:** Remote piloting is dependent on uninterrupted communication links, which may be disrupted in urban, indoor, or disaster-prone environments.
- **Limited Situational Awareness:** Humans cannot process high-speed sensory data effectively, particularly from multiple sensors simultaneously.

Autonomous onboard perception enables UAVs to estimate depth, detect obstacles, and plan trajectories instantaneously. Modern approaches combine monocular or stereo vision with sensor fusion techniques to create high-fidelity environmental maps, improving navigation in complex or GPS-denied environments.

1.3 Recent Advances in UAV Perception

Significant progress has been made in perception and autonomous navigation for UAVs:

- **Viewpoint Selection and Guidance Generation:** Liu *et al.* demonstrated that UAVs can efficiently explore unknown environments by selecting optimal viewpoints and guidance paths, improving mapping and navigation efficiency [4].

- **Sensor Fusion:** Yu *et al.* fused radar with monocular vision to enhance obstacle detection robustness, particularly in low-light or visually ambiguous conditions [5].
- **Lightweight Depth Estimation:** Patel *et al.* introduced AvioDepth, a lightweight monocular depth estimation framework designed for embedded UAV platforms, capable of providing accurate geometric cues under strict computational constraints [1].

These studies emphasize the necessity of **hardware-efficient perception modules** that can execute in real-time on embedded UAV platforms.

1.4 Project Objectives

This project focuses on designing and implementing **hardware-accelerated modules** for UAV perception and navigation. The primary objectives include:

- Designing a Verilog-based depth estimation accelerator capable of processing visual input and generating relative scene depth efficiently.
- Developing a real-time path planning module that computes safe trajectories using depth information and obstacle detection.
- Ensuring synthesizability of the hardware modules on FPGA platforms and compatibility with STM32 microcontrollers with AI acceleration capabilities.
- Evaluating the modules through simulation, latency analysis, resource utilization, and power efficiency metrics.
- Providing a foundational hardware framework for potential integration into full UAV autonomy stacks.

1.5 Advantages of On-Device Inference

Executing inference locally on embedded UAV hardware provides several key benefits:

- **Ultra-Low Latency:** Real-time decision-making without transmission delays.
- **Energy Efficiency:** Eliminates dependence on external processing, conserving UAV battery life.
- **Reliability:** Maintains functionality even in communication-challenged or GPS-denied environments.

- **Data Privacy:** Sensitive visual and sensor data remain onboard.
- **Cost Effectiveness:** Reduces operational dependence on cloud computing.

1.6 Hardware Platforms

1.6.1 Zybo Z7 FPGA Board

The Zybo Z7 features an ARM Cortex-A9 processor coupled with programmable logic, providing:

- Parallel computation for accelerating convolution operations.
- Deterministic timing suitable for real-time control.
- Flexibility for prototyping hardware accelerators before embedded deployment.

1.6.2 STM32N657X0-Q Board

The STM32N657X0-Q includes:

- ARM Cortex-M55 CPU with Helium (MVE) vector extensions.
- Neural-ART accelerator optimized for embedded neural network execution.
- Sufficient memory bandwidth for deploying lightweight AI models in real time.

1.7 Report Organization

The structure of this thesis is as follows:

- **Chapter 2: Literature Survey** — Covers UAV perception, depth estimation, obstacle avoidance, sensor fusion, and hardware acceleration approaches.
- **Chapter 3: Depth Estimation Hardware Architecture** — Details Verilog design for depth estimation and simulation results.
- **Chapter 4: Path Planning Hardware Architecture** — Presents hardware-based trajectory computation and obstacle-aware navigation modules.
- **Chapter 5: Monocular Depth Estimation on STM32N6 NPU** — Embedded deployment of the depth estimation module and evaluation.

- **Chapter 6: Results** — Comprehensive analysis including accuracy, latency, FPGA resource utilization, and power efficiency.
- **Chapter 7: Conclusion and Future Scope** — Summarizes findings and highlights potential extensions for UAV autonomy.

1.8 Chapter Summary

This chapter introduced the need for autonomous UAV perception and navigation, highlighted challenges associated with deploying AI algorithms on resource-constrained hardware, and presented the motivation for hardware-accelerated depth estimation and path planning. It also described the project objectives, advantages of on-device inference, and the hardware platforms selected for implementation. The next chapter provides a detailed literature survey covering previous research in embedded UAV perception, depth estimation, and hardware acceleration.

Chapter 2
Literature Survey

This chapter reviews the existing research in monocular depth estimation (MDE) and reactive path planning for autonomous UAV navigation. The survey highlights the evolution of lightweight depth estimation frameworks suitable for embedded platforms, along with sector-based reactive planning techniques that enable real-time obstacle avoidance. The discussion integrates insights from recent UAV exploration, sensor fusion, and embedded perception studies to establish the motivation for the proposed hardware-accelerated system.

2.1 Depth Estimation for UAV Navigation

Depth estimation forms the backbone of UAV perception, enabling the reconstruction of the three-dimensional environment from two-dimensional sensory input. Accurate depth inference is essential for tasks such as obstacle avoidance, terrain assessment, autonomous landing, and exploration in unstructured or GPS-denied environments. Traditional depth acquisition systems—including LiDAR, structured light, stereo-camera rigs, and time-of-flight sensors—offer high measurement accuracy but introduce significant penalties in terms of size, weight, power, and cost. These SWaP constraints limit their suitability for small UAVs that require lightweight and energy-efficient perception solutions.

Monocular depth estimation (MDE) addresses these limitations by generating dense depth maps from a single RGB camera. However, the challenge lies in designing depth networks that maintain high accuracy while meeting stringent real-time requirements on embedded processors commonly used in UAV platforms.

2.1.1 Evolution of Monocular Depth Estimation Approaches

Initial deep-learning-based MDE frameworks predominantly used encoder–decoder CNN architectures. These architectures extracted hierarchical features using convolutional encoders and reconstructed full-resolution depth maps using multi-scale decoders. Although accurate, such architectures are computationally expensive and unsuitable for onboard microcontrollers or FPGA-based UAV systems.

Transformer-based MDE models improved global spatial reasoning through multi-head self-attention, offering better depth continuity and long-range scene understanding. However, their quadratic memory complexity, higher arithmetic operations, and large parameter size make them impractical for real-time aerial deployment where latency and energy efficiency are mission-critical.

These limitations motivated research into lightweight MDE models optimized specifically for embedded deployment, balancing accuracy with low inference time, low memory footprint, and hardware-friendly operations.

2.1.2 MobileNetV3-Small as an Embedded-Friendly Backbone

MobileNetV3-Small, introduced by Howard et al. [2], is widely used as a backbone for resource-constrained vision systems due to its hardware-aware design and computational efficiency. Several architectural features contribute to its suitability for monocular depth estimation in UAVs:

- **Depthwise Separable Convolutions:** Reduce the computational complexity of convolution operations significantly, enabling efficient feature extraction.
- **Inverted Residual Blocks:** Preserve representational richness while maintaining a narrow computational bottleneck, ideal for real-time embedded inference.
- **Squeeze-and-Excitation (SE) Modules:** Introduce channel-wise attention at minimal cost, improving robustness in environments with varying textures and illumination.
- **Hard-Swish Activation:** A quantization-friendly activation function that improves accuracy while remaining efficient for integer and fixed-point accelerators.

These design choices make MobileNetV3-Small highly suitable as a backbone for UAV depth estimation tasks where both efficiency and perceptual quality are required.

2.1.3 AvioDepth: A Lightweight Depth Estimation Framework for UAVs

Patel et al. [1] proposed AvioDepth, a monocular depth estimation framework designed specifically for UAV navigation and real-time embedded systems. Unlike generic MDE networks, AvioDepth emphasizes low computational overhead while maintaining sufficient geometric fidelity for safe obstacle avoidance.

The architecture integrates three important components:

- **MobileNetV3-Small Encoder:** Provides compact yet expressive feature maps optimized for real-time execution on platforms such as Raspberry Pi, Jetson Nano, and microcontroller-based inference engines.

- **Ultra-Lightweight Sequential Attention (ULWA):** A two-stage attention mechanism consisting of channel attention followed by spatial attention. This design enhances salient geometric features and structural edges without incurring the computational cost typical of transformer-based attention mechanisms.
- **Sub-Pixel Convolution Decoder:** Utilizes pixel-shuffle operations to reconstruct high-resolution depth maps efficiently. This avoids checkerboard artifacts and reduces the parameter count compared to transposed convolutions.

Fig. 2.1 shows the architecture of AvioDepth, which includes attention-augmented feature extraction and multi-scale depth reconstruction.

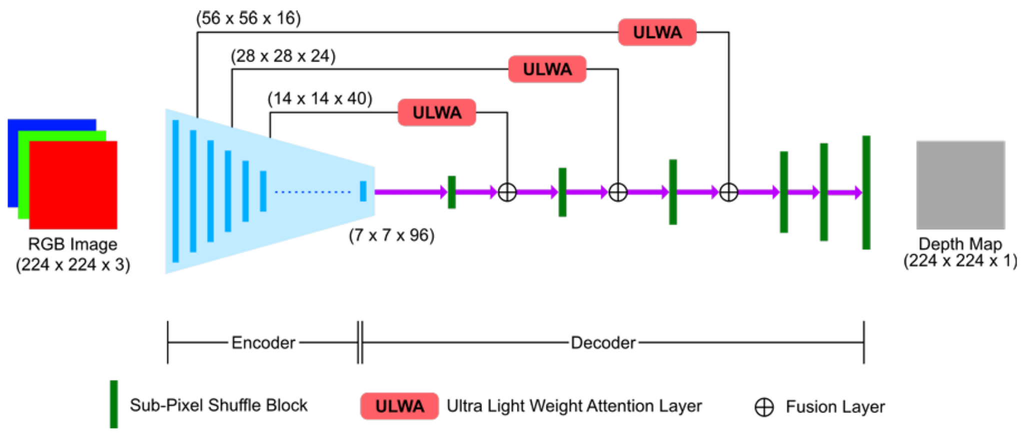


Figure 2.1: Architecture of AvioDepth showing the MobileNetV3-Small encoder, sequential attention modules, skip connections, and sub-pixel decoder [1].

AvioDepth demonstrates high inference efficiency, achieving near real-time framerates even on low-cost embedded platforms. This makes it highly suitable for UAV missions involving fast obstacle avoidance, confined-space navigation, and autonomous exploration.

2.1.4 Depth Estimation in High-Level UAV Tasks

Depth estimation plays a crucial role beyond immediate obstacle detection. Liu et al. [4] highlight the importance of depth-aware viewpoint selection for autonomous exploration, demonstrating that depth cues enable UAVs to identify unexplored regions and plan efficient coverage paths. High-quality depth maps help drones maximize environmental understanding while maintaining safe distances from obstacles.

Yu et al. [5] further show that integrating monocular depth inference with complementary sensing (e.g., radar) enhances robustness under challenging conditions such as poor illumination, low-texture surfaces, and atmospheric disturbances. Their fusion approach emphasizes the importance of reliable depth estimation pipelines in practical UAV deployments.

2.2 Path Planning for UAV Obstacle Avoidance

Reactive path planning provides the UAV with immediate steering decisions derived directly from sensor input, without relying on global map reconstruction or computationally expensive planning algorithms. This approach offers low latency and high responsiveness, which are crucial for micro-UAVs flying at high speeds in cluttered environments.

A common strategy for computationally efficient reactive navigation is to transform the camera's horizontal field of view (FOV) into a discrete angular representation. Depth information is compressed into directional obstacle distances, allowing fast evaluation of free-space corridors.

2.2.1 Field-of-View Segmentation

The UAV camera's horizontal FOV is uniformly divided into N angular sectors. Each pixel column x in an image of width W maps to a sector index:

$$s = \left\lfloor \frac{x \cdot N}{W} \right\rfloor .$$

This conversion compresses thousands of raw depth values into a small set of angular descriptors. Such a representation aligns well with FPGA and microcontroller architectures, enabling rapid and deterministic processing.

2.2.2 Sector-wise Minimum Depth Extraction

To ensure safety, each sector records the minimum depth value observed within its pixel range:

$$d_{\min}(s) = \min_{x \in s} D(x).$$

This minimum-depth strategy provides a conservative estimate of obstacle proximity in each direction. It is analogous to directional occupancy mapping and ensures the UAV selects the safest steering actions even when parts of a sector contain occluded or partially visible obstacles.

2.2.3 Valley Detection for Free-Space Identification

A *valley* refers to a sequence of neighboring sectors whose minimum depth values exceed a predefined safety threshold:

$$d_{\min}(s) \geq d_{\text{safe}}.$$

Valleys represent free-space corridors suitable for safe navigation. The midpoint of each valley is considered a candidate steering direction, and wider valleys are prioritized due to their inherent maneuvering margin. This approach is computationally lightweight, making it ideal for on-chip real-time implementation.

2.2.4 Yaw Computation Using Fixed-Point Q1.15 Representation

Once the optimal valley midpoint s_{mid} is selected, the yaw deviation relative to the center sector is computed:

$$\Delta s = s_{\text{mid}} - s_{\text{center}}.$$

This deviation is converted into a steering angle and encoded using a Q1.15 fixed-point format:

$$\theta_{\text{Q15}} = \theta_{\text{float}} \times 32768.$$

Fixed-point arithmetic ensures deterministic execution latency and avoids the overhead of floating-point units, making it suitable for FPGA logic and low-power embedded processors. Together, sector-wise depth compression, valley detection, and fixed-point yaw computation form the basis of efficient reactive path planning for UAVs.

2.3 Summary

The literature demonstrates that accurate depth estimation and efficient path planning are essential for autonomous UAV navigation. Lightweight MDE networks such as AvioDepth leverage MobileNetV3-Small and sequential attention modules to achieve real-time performance on embedded platforms. Exploration and obstacle avoidance studies emphasize the critical role of depth in maintaining safe and informed flight behavior. Sector-wise reactive path planning provides a computationally efficient mechanism for real-time UAV steering, supported by depth compression, valley detection, and fixed-point yaw generation. These insights collectively motivate the development of hardware-accelerated depth estimation and path planning modules in this project.

Chapter 3
Depth Estimation

This chapter presents a comprehensive description of the monocular depth estimation (MDE) pipeline along with the design and implementation of a custom FPGA-based convolution accelerator for real-time spatial filtering. The primary objective is to execute depth-estimation–related convolution operations efficiently on a resource-constrained FPGA by employing a compact, fixed-function hardware engine, rather than relying on large, GPU-oriented convolution accelerators that require extensive memory and computational resources.

To establish this unified framework, the chapter first describes the lightweight MDE pipeline developed using a MobileNetV3-Small encoder paired with a ULWA-enhanced decoder, enabling low-latency depth prediction suitable for edge devices. This software pipeline provides the basis for identifying essential convolution kernels that must be accelerated in hardware.

The second part of the chapter details the design of the Verilog-based convolution engine, capable of performing both 1D and 2D spatial filtering using fixed-point Q8.8 arithmetic. The architecture supports multiple kernel sizes, employs an efficient line-buffering mechanism, and incorporates a pipelined datapath to achieve high throughput on a low-cost FPGA. The integration of control logic, memory interfaces, and arithmetic blocks is captured in the high-level RTL schematic shown in Figure 3.2, which illustrates the overall organization of the datapath, buffering structure, and computational modules that form the complete convolution accelerator.

3.1 Monocular Depth Estimation Overview

Monocular depth estimation refers to the task of predicting per-pixel scene depth using a single RGB image. Unlike stereo-vision systems or LiDAR-based sensors that require multiple viewpoints or mechanical components, monocular approaches offer significant benefits including low power consumption, reduced hardware complexity, and ease of deployment on embedded systems such as UAVs and portable robotic platforms. Traditional geometric approaches such as Structure-from-Motion, shape-from-shading, and depth-from-focus rely heavily on multiple frames or controlled illumination conditions, which limits their applicability in dynamic outdoor environments. Deep learning-based monocular depth estimation models overcome these limitations by exploiting semantic and contextual patterns present in natural scenes.

Despite their advantages, CNN-based methods require a large number of convolution operations. Real-time execution of such operations on embedded devices is often constrained by power, compute resources and memory bandwidth. This challenge motivates the integration of a custom hardware accelerator specifically optimized for spatial convolution operations, which form the computational backbone of modern MDE networks.

3.2 MobileNetV3-Small Encoder

MobileNetV3-Small is adopted in this work due to its efficient balance between computational speed and prediction accuracy. Its architecture is designed specifically for mobile and embedded platforms and relies on a combination of Hard-Swish activations, Squeeze-and-Excitation (SE) modules and inverted residual (IR) bottleneck blocks. Hard-Swish provides a smooth nonlinear activation at low computational cost, while SE modules recalibrate channel-wise features by emphasizing informative dimensions. The inverted residual block is constructed using an expand–depthwise–project sequence that reduces the number of multiplications while maintaining spatial detail. Mathematically, the IR block is represented as

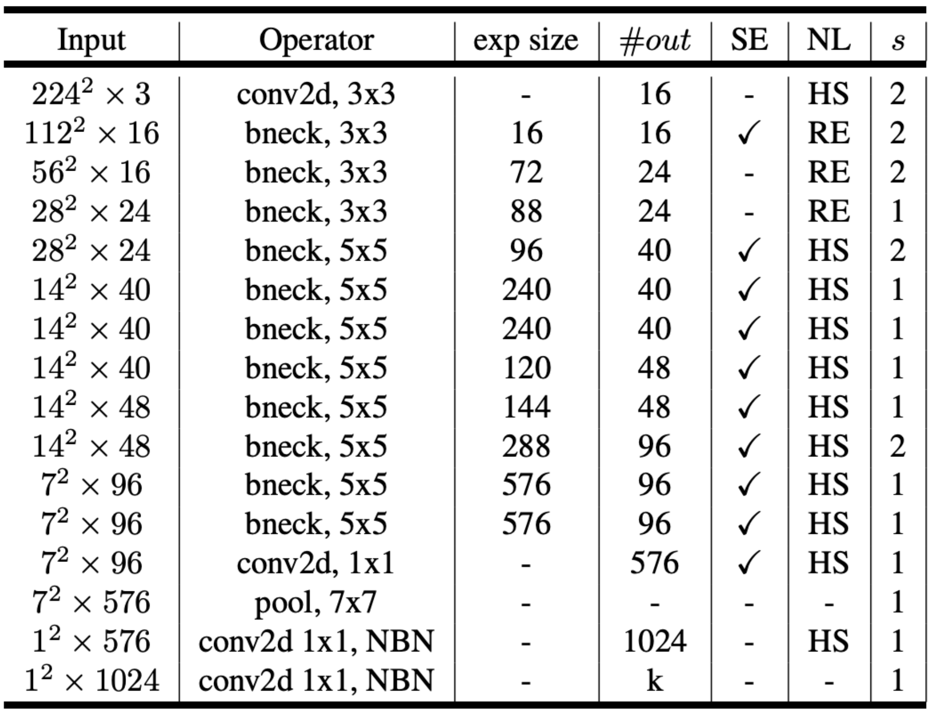
$$\mathrm{IR}(x) = W_p(W_s(\mathrm{DWConv}(x))),$$

where the depthwise convolution extracts channel-wise spatial features and the projection layer reduces the feature dimensionality back to its compact form.

The encoder architecture used in this project is illustrated in Figure 3.1. This diagram highlights the hierarchical arrangement of inverted residual blocks, SE units and depthwise separable convolutions across multiple stages of the network. The encoder outputs multi-scale feature maps at spatial resolutions of 56×56, 28×28, 14×14 and 7×7. The highest-resolution layers generally preserve edges and fine structures, the mid-resolution layers capture surface-level textures and the lowest-resolution feature maps encode high-level semantic information. These multi-scale representations collectively enhance the decoder's ability to reconstruct dense and visually coherent depth maps.

3.3 ULWA-Based Decoder

The decoder reconstructs the full-resolution depth map by progressively upsampling the encoded features while preserving spatial structure and fine-grained details. In this design, upsampling is performed through sub-pixel convolutions implemented using the PixelShuffle operation. A 1×1 convolution first expands the number of channels to $C \cdot r^2$, and the PixelShuffle operation rearranges these channels into a spatially enlarged



Input	Operator	exp size	$\#out$	SE	NL	s
$224^2 \times 3$	conv2d, 3x3	-	16	-	HS	2
$112^2 \times 16$	bneck, 3x3	16	16	✓	RE	2
$56^2 \times 16$	bneck, 3x3	72	24	-	RE	2
$28^2 \times 24$	bneck, 3x3	88	24	-	RE	1
$28^2 \times 24$	bneck, 5x5	96	40	✓	HS	2
$14^2 \times 40$	bneck, 5x5	240	40	✓	HS	1
$14^2 \times 40$	bneck, 5x5	240	40	✓	HS	1
$14^2 \times 40$	bneck, 5x5	120	48	✓	HS	1
$14^2 \times 48$	bneck, 5x5	144	48	✓	HS	1
$14^2 \times 48$	bneck, 5x5	288	96	✓	HS	2
$7^2 \times 96$	bneck, 5x5	576	96	✓	HS	1
$7^2 \times 96$	bneck, 5x5	576	96	✓	HS	1
$7^2 \times 96$	conv2d, 1x1	-	576	✓	HS	1
$7^2 \times 576$	pool, 7x7	-	-	-	-	1
$1^2 \times 576$	conv2d 1x1, NBN	-	1024	-	HS	1
$1^2 \times 1024$	conv2d 1x1, NBN	-	k	-	-	1

Figure 3.1: MobileNetV3-Small encoder architecture highlighting Inverted Residual blocks, SE modules and depthwise separable convolutions [2].

feature map,

$$Y = \mathrm{PixelShuffle}(X') \in \mathbb{R}^{(Hr)\times(Wr)},$$

thereby avoiding the high computational cost associated with transposed convolutions.

To further improve reconstruction accuracy, the decoder integrates a ULWA attention mechanism. The ULWA module operates in two stages. The first stage enhances channel significance through global average pooling followed by a two-layer MLP and a sigmoid gate, resulting in a channel attention weight

$$A_c = \sigma(W_2(\delta(W_1(\mathrm{GAP}(X))))), \quad X' = X \cdot (1 + A_c).$$

The second stage refines spatial patterns by computing a spatial descriptor using the mean and max maps, followed by a 7×7 convolution,

$$M = \mathrm{Max}(X') + \mathrm{Mean}(X'), \quad X'' = X' \cdot (1 + \sigma(\mathrm{Conv}_{7\times7}(M))).$$

Through these two attention components, ULWA produces depth maps that exhibit improved sharpness, continuity and robustness to texture ambiguity.

3.4 Hardware Convolution Implementations

The hardware implementation comprises two convolution modules: a 1D convolution engine used for Q8.8 arithmetic verification and functional debugging, and a fully parallel 2D convolution accelerator that performs actual spatial filtering. Both modules utilize fixed-point Q8.8 arithmetic to minimize FPGA resource consumption while maintaining sufficient numerical precision for image processing tasks.

The 1D convolution module computes

$$y[n] = \sum_{k=0}^{K-1} x[n+k]\,h[k],$$

with 8-bit signed inputs and coefficients, a Q16.16 multiplication result and a 32-bit accumulator. The module served as an early-stage validation tool for verifying multiplier correctness, checking overflow behavior and evaluating pipeline register insertion.

The 2D convolution accelerator computes

$$O(y,x) = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} I(y+i, x+j)\,W(i,j)$$

for supported kernel sizes of 1×1, 3×3, 5×5 and 7×7. All kernels are internally padded to 7×7 dimensions according to

$$W_{7\times7}(i,j) = \begin{cases} W(i,j), & i < k,\ j < k, \\ 0, & \text{otherwise}, \end{cases}$$

ensuring a unified datapath regardless of the selected kernel size.

3.5 Linear vs. Systolic Convolution Architecture

Two architectural approaches were evaluated for the 2D convolution engine: a systolic array and a linear parallel array. Systolic architectures provide excellent scalability and data reuse for large matrix operations, although their reliance on inter-processing-element communication leads to higher latency, increased routing complexity and unnecessary overhead for small kernel sizes. In contrast, the linear parallel array architecture uses direct parallel multiplication combined with a multi-stage adder tree and a pipelined structure. This arrangement produces one filtered pixel per clock cycle with deterministic latency and lower resource usage, making it ideally suited for the limited kernel sizes employed in depth-estimation-related spatial filtering. A detailed comparison between the two architectures is summarized in Table 3.1.

Table 3.1: Comparison of Linear and Systolic Convolution Architectures

Feature	**Linear Array**	**Systolic Array**
Processing Structure	Parallel MAC + adder tree	Interconnected 2D grid of PEs
Dataflow	Direct streaming	Rhythmic data movement between PEs
Latency	Low and deterministic	Higher due to multi-hop propagation
Routing Complexity	Low	High
Scalability	Suitable for small kernels	Ideal for large matrices
Resource Usage	Straightforward multiplier use	More registers + interconnects
Best Use Cases	Small–moderate convolutions	Deep CNN layers and matrix ops

3.6 FPGA Hardware System Architecture

The complete hardware architecture of the convolution accelerator is shown in Figure 3.2. The system integrates a BRAM-based memory subsystem for input image storage, a control finite-state machine (FSM), a sliding-window line-buffering engine and a parallel multiply-accumulate computation core. The BRAM subsystem provides sequential streaming access for the accelerator while allowing random writes from the host. The FSM manages transitions between IDLE, EXECUTE and FINISH states while controlling kernel selection, processing flow and completion signaling.

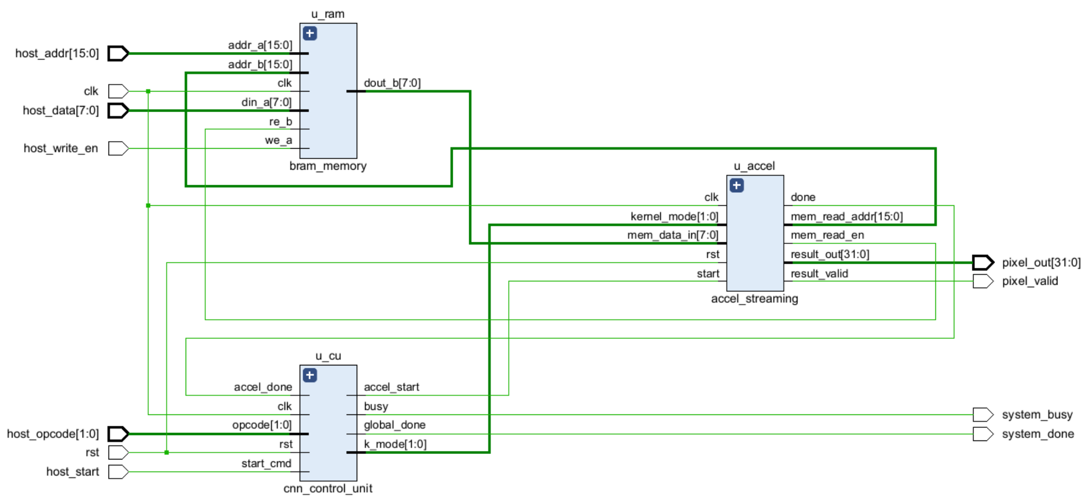


Figure 3.2: Complete RTL schematic of the FPGA-based convolution accelerator.

The sliding-window generator uses six line buffers along with a row register to construct a 7×7 window of pixels. A warm-up time of 735 clock cycles is required before the first valid window becomes available. After warm-up, the engine produces one valid window per clock cycle. The convolution core processes each window through a four-stage pipeline that includes parallel multiplication, two-stage adder-tree reduction and final accumulation. When the computation completes for a pixel, the signal `result_valid` is asserted, and the pixel is written to the output formatter in 32-bit signed hexadecimal form.

The system supports an output resolution of 217×217. Each pixel is stored using a 32-bit fixed-point representation, allowing precise evaluation during verification. A Python-based fixed-point simulation pipeline was used to validate the accelerator output. The verification process applied Q8.8 scaling, kernel padding, fixed-point convolution and pixel-wise comparison against FPGA results. The maximum observed mismatch was only 0.00213%, which is attributable to rounding effects and pipeline timing shifts and is well within acceptable tolerance for fixed-point computation.

3.7 Chapter Summary

This chapter described the complete monocular depth estimation pipeline and the corresponding hardware accelerator designed for efficient spatial filtering on FPGA. The software pipeline is based on MobileNetV3-Small combined with a ULWA-attention decoder that generates visually coherent depth representations. On the hardware side, a line-buffered fixed-point 2D convolution accelerator was developed using a linear parallel architecture, providing deterministic one-pixel-per-cycle throughput. Validation against a Python golden model demonstrated a negligible pixel mismatch of 0.00213%, confirming both correctness and reliability. Future extensions of this work include the implementation of activation functions such as ReLU and Hard-Swish directly in Verilog, enabling further migration of the depth estimation pipeline into FPGA hardware.

Chapter 4
Path Planning

This chapter presents the Verilog-based RTL (Register Transfer Level) module designed for real-time depth-to-yaw path planning in autonomous UAV navigation. The implementation follows a modular, finite state machine architecture that transforms depth sensor data into steering commands for obstacle avoidance.

4.1 System Architecture and Design Overview

The path planner module is a synchronous digital circuit that processes 224×224 pixel depth images to generate yaw angle commands for UAV steering. The design divides the horizontal field of view into 91 angular sectors, reducing the complex navigation decision from analyzing thousands of pixels to evaluating 91 directional bins. This sector-based approach simplifies the computational requirements while retaining essential obstacle proximity information needed for safe navigation.

The module interface consists of three control inputs (clock, reset, start) and five outputs (done, yaw angle, forward clearance, best sector index, and emergency flag). As shown in Figure 4.1, the processing pipeline flows through seven distinct states, each performing a specific function in transforming raw depth data into navigation commands.

The system operates on a constrained vertical region of the depth image, specifically rows 78 through 168, which corresponds to the ground-level obstacle zone most critical for safe flight.

4.1.1 Module Interface and Signal Definitions

The input signals include a system clock for synchronous operation, an asynchronous reset for initialization, and a start signal that triggers processing of a new depth frame. The depth image data is pre-loaded into an internal memory buffer before asserting the start signal.

The primary output is a 16-bit signed yaw angle encoded in Q1.15 fixed-point format, representing steering commands from negative pi to positive pi radians with sub-degree precision. The front clearance output provides a 16-bit unsigned distance metric indicating obstacle proximity ahead, useful for velocity control. The best sector output exposes the selected navigation direction as a 7-bit index for debugging. The done flag signals completion of processing, and the no safe flag alerts when no navigable path exists.

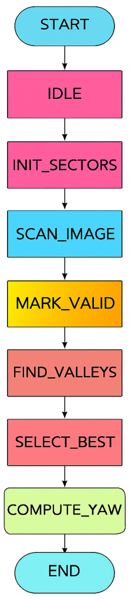


Figure 4.1: Seven-state Flowchart.

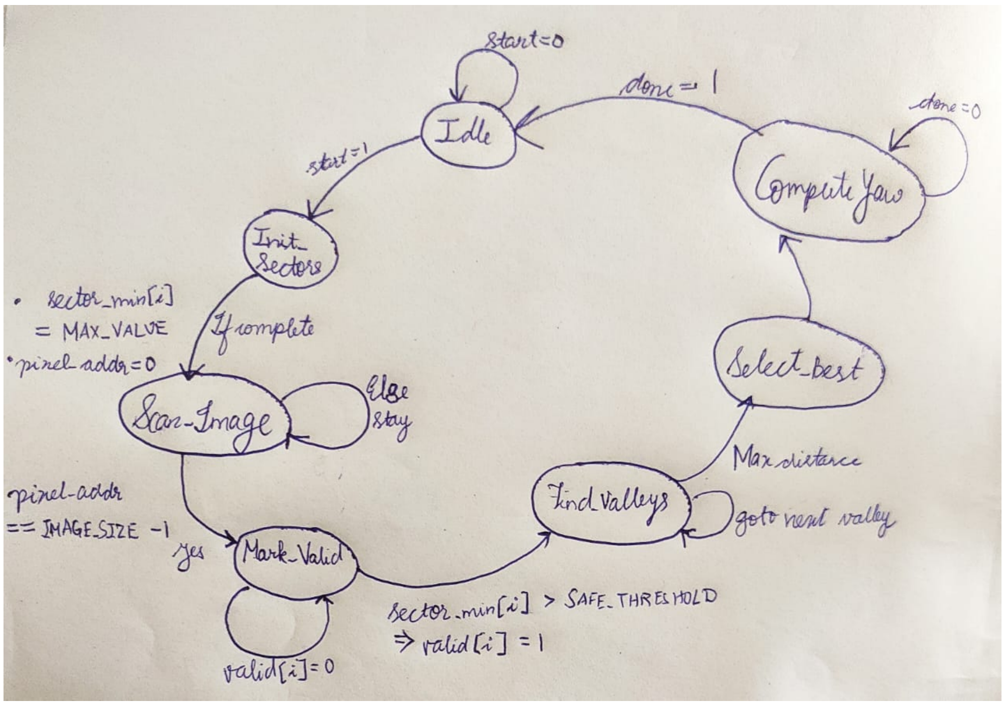


Figure 4.2: Complete FSM diagram showing state transitions for the path-processing module.

4.1.2 Design Parameters

The module is configured with several key parameters. The image dimensions are set to 224×224 pixels, a common resolution for embedded depth sensors. The number of sectors is 91, providing approximately 2-degree angular resolution across the field of view. The scanning region spans from row 78 to row 168, processing 90 rows per frame. The safety threshold is set to 1800 in unsigned 16-bit depth units, defining the minimum acceptable obstacle distance. The minimum valley width parameter requires safe corridors to span at least 8 contiguous sectors, filtering noise and preventing selection of dangerously narrow gaps.

4.2 Seven-State Processing Pipeline

The control logic implements a finite state machine with seven sequential states, as illustrated in the flowchart of Figure 4.1. Each state performs a specific processing task, and state transitions occur on clock edges based on completion conditions.

4.2.1 IDLE State

The idle state is the system's resting condition, waiting for a new depth frame. When the start signal asserts, the module initializes processing variables including the best cost register (set to maximum value 255), the best run midpoint (set to a value 127), and the center sector reference (calculated as 45). The state then advances to sector initialization.

4.2.2 INIT_SECTORS State

This state prepares the sector minimum array by writing the maximum 16-bit value (0xFFFF) to all 91 entries. This initialization ensures that subsequent minimum-finding operations correctly capture the closest obstacle in each direction. The state iterates through all sectors using a counter, requiring exactly 91 clock cycles. Upon completion, the row and column indices are positioned at the start of the scanning region.

4.2.3 SCAN_IMAGE State

The scanning state performs the core depth processing, iterating through each pixel in the defined region. For each pixel, the module reads the depth value from memory, checks if it is non-zero (valid), and applies any body radius offset. The pixel's column index is mapped to a sector using a linear function that multiplies the column by 91 and divides by 224. If the adjusted depth is less than the current sector minimum, the sector minimum is updated.

The address logic handles both horizontal and vertical progression. After each pixel, the column index and pixel address increment. At row boundaries, the column resets to zero, the row increments, and the pixel address is recalculated as the new row index times 224. This continues until reaching row 168, consuming 20,160 clock cycles (90 rows × 224 columns).

4.2.4 MARK_VALID State

The safety classification state evaluates each sector's navigability. For each of the 91 sectors, the module compares the sector minimum depth against the safety threshold of 1800. If the depth meets or exceeds this threshold, the sector is marked safe (bit set to 1 in the valid vector); otherwise it is marked unsafe (bit set to 0). This state requires 91 clock cycles, one per sector.

4.2.5 FIND_VALLEYS State

Valley detection identifies contiguous runs of safe sectors using run-length encoding. The algorithm maintains a run-active flag and a run-start index. When transitioning from an unsafe to safe sector, a new run begins. When transitioning from safe to unsafe, the run terminates and its start and end indices are stored in valley arrays. A special case handles runs extending to the final sector. This state iterates through all 91 sectors in 91 clock cycles, potentially detecting multiple valleys.

4.2.6 SELECT_BEST State

The selection state chooses the optimal valley based on width and proximity to straight-ahead. For each detected valley, the module calculates width as end minus start plus one. Valleys narrower than 8 sectors are rejected. For valid valleys, the midpoint is calculated and a cost is computed as the absolute distance from the center sector (index 45). The valley with minimum cost is selected, representing the direction requiring the least heading change. This state processes all detected valleys, taking up to 91 cycles in the worst case.

4.2.7 COMPUTE_YAW State

The yaw computation state generates the final outputs. If no valid valley was found (best midpoint still at 127), the sector is set to center and the no safe flag asserts. Otherwise, the selected sector undergoes a linear transformation to yaw angle: the sector index minus 45 is multiplied by 65,534 and divided by 90, producing a Q1.15 fixed-point value representing the angle in radians.

The front clearance calculation uses a three-way median filter on depth samples from sectors at indices 30, 45, and 60 (representing left, center, and right thirds). The median operation uses comparison logic to select the middle value without sorting, providing robustness against isolated erroneous readings. This state completes in a single clock cycle. , ready for the next frame.

4.3 Key Design Features and Benefits

4.3.1 Hardware Efficiency

The design uses fixed-point arithmetic in Q1.15 format, avoiding the complexity and area costs of floating-point hardware while achieving 0.0017-degree angular precision. The sector-based representation compresses the depth image from 50,176 pixels to 91 directional samples, dramatically reducing storage and processing requirements. The single-pass scanning architecture processes each pixel exactly once without frame buffering.

4.3.2 Modular Architecture

The seven-state FSM provides clear separation of concerns, with each state performing a well-defined function. This modularity simplifies verification, debugging, and future enhancements. The sector mapping function is implemented as a reusable combinational block, and the median filter uses efficient comparison logic requiring only six comparisons.

4.3.3 Safety Features

The explicit no safe path flag enables deterministic failure detection, allowing the flight controller to execute emergency procedures when no navigable corridor exists. The minimum valley width filter prevents selection of gaps too narrow for safe passage. The median-based clearance calculation provides robustness against sensor noise.

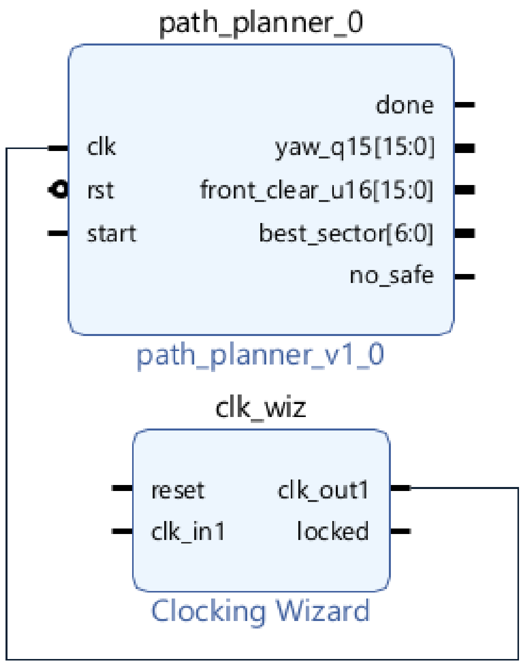


Figure 4.3: Top-level hardware integration showing the `path_planner_0` module driven by a 100 MHz clock generated through the clocking wizard. The block diagram illustrates the flow of clock, reset, and start signals, along with the computed outputs including yaw, clearance, best-sector index, and safety flag.

4.4 Applications and Use Cases

4.4.1 Autonomous Drone Navigation

The primary application is obstacle avoidance for quadcopters and fixed-wing UAVs operating in GPS-denied environments. Indoor warehouse navigation, urban canyon flight, and forest traversal all benefit from the fast reaction time. Agricultural surveying, infrastructure inspection, and search-and-rescue missions require reliable autonomous navigation in cluttered spaces.

4.4.2 Ground Robot Path Planning

The sector-based approach adapts naturally to wheeled robots and autonomous vehicles. Warehouse robots, delivery vehicles, and agricultural rovers benefit from the real-time collision avoidance capabilities. The deterministic timing supports safety-critical applications where worst-case behavior must be guaranteed.

4.4.3 Research and Education

The RTL module serves as an educational tool for teaching digital design, embedded systems, and robotics. Students learn finite state machine design, fixed-point arithmetic, and real-time processing through a practical application. Researchers developing novel navigation algorithms use the module as a baseline for comparison and as a building block for more complex systems.

4.5 Chapter Summary

This chapter presented the RTL architecture for a hardware path planning module that transforms depth sensor data into yaw steering commands. The seven-state finite state machine processes 224×224 depth images through sector-based spatial discretization, valley detection, and fixed-point arithmetic to generate navigation commands.

Chapter 5
Monocular Depth Estimation on the STM32N6 NPU

5.1 Rationale for NPU Acceleration in Dense Prediction

Monocular Depth Estimation (MDE) is a dense regression task that predicts a continuous depth value for every pixel in the input image. Unlike classification, which outputs a single label, MDE requires high-resolution feature maps throughout the network. Architectures like FastDepth perform substantial upsampling operations in the decoder, leading to extremely high Multiply–Accumulate (MAC) workloads [6, 7].

Running such operations on a general-purpose CPU such as the Cortex-M55 is inefficient. The STM32N6 addresses this through its **Neural-ART Accelerator**, capable of delivering up to 600 GOPS of INT8 performance. By accelerating convolution-heavy operations on the NPU, the Cortex-M55 remains free to handle application logic, peripheral management, and sensor data processing without becoming a bottleneck [3, 8].

To provide an overview of how neural networks transition from high-level training frameworks to embedded execution, the following diagram illustrates the complete ST Edge AI workflow, including model import, optimization, quantization, and final deployment to the Neural-ART Accelerator.

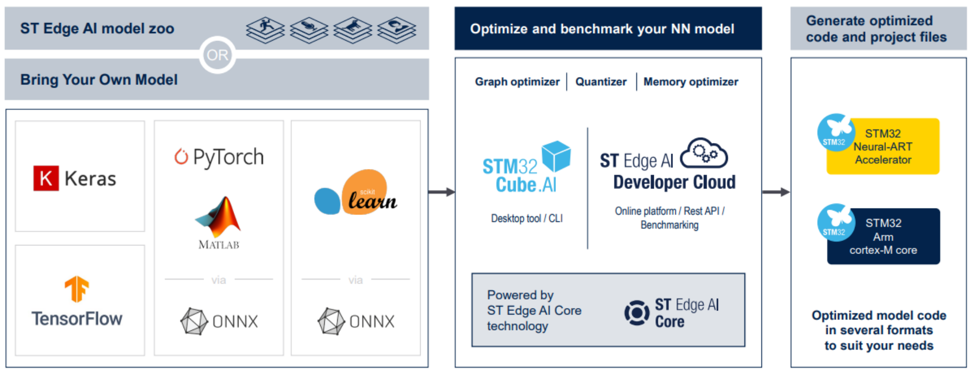

ST Edge AI model zoo
OR
Bring Your Own Model
Keras
PyTorch
MATLAB
scikit learn
TensorFlow
via
ONNX
via
ONNX
Optimize and benchmark your NN model
Graph optimizer | Quantizer | Memory optimizer
STM32 Cube.AI
Desktop tool / CLI
ST Edge AI Developer Cloud
Online platform / Rest API / Benchmarking
Powered by ST Edge AI Core technology
ST Edge AI Core
Generate optimized code and project files
STM32 Neural-ART Accelerator
STM32 Arm cortex-M core
Optimized model code in several formats to suit your needs

Figure 5.1: ST Edge AI workflow supporting model import, optimization, quantization, and deployment on STM32N6 NPUs and Cortex-M cores [3].

5.2 X-CUBE-AI and Model Processing

To deploy our custom-trained depth estimation model on the STM32N6 hardware, we use **X-CUBE-AI** (STEdgeAI Core v2.2+), which acts as an optimizing compiler. It imports the ONNX model, validates its operators, performs graph-level optimizations, assigns memory layouts, and finally generates C code along with NPU microcode required for Neural-ART execution [9].

Before optimization, the tool must convert the high-level framework model into a form compatible with the hardware pipeline. The following diagram shows how X-CUBE-AI imports a model from TensorFlow, Keras, or ONNX and processes it through its validation and conversion stages.

Once the model is imported, X-CUBE-AI performs several optimizations to ensure efficient NPU execution:

- **Graph Analysis:** Validates that all layers used in the model are supported by the Neural-ART accelerator or mapped efficiently to the Cortex-M55 [10].
- **Operator Fusion:** Combines sequences such as Conv + BatchNorm + ReLU to reduce intermediate memory transfers and improve latency [11].
- **Memory Allocation:** Places large model weights in external Flash (XSPI) while keeping activation buffers in high-speed AXI-SRAM for maximum throughput [12].

5.3 The Challenge of Post-Training Quantization (PTQ) in Regression

Deploying neural networks on the STM32N6 requires converting FP32 tensors into INT8 format, since the Neural-ART accelerator performs all computations using 8-bit arithmetic. While Post-Training Quantization (PTQ) works well for classification, it often introduces accuracy degradation in regression tasks such as Monocular Depth Estimation (MDE). Depth prediction is highly sensitive to small numerical variations, so quantization errors can disrupt pixel-level gradients and global scene geometry.

The general PTQ and QAT workflow used for ONNX and TFLite models is illustrated below. It shows how representative datasets, quantization scripts, and export pipelines produce INT8 models that are then processed by the ST Edge AI Core toolchain.

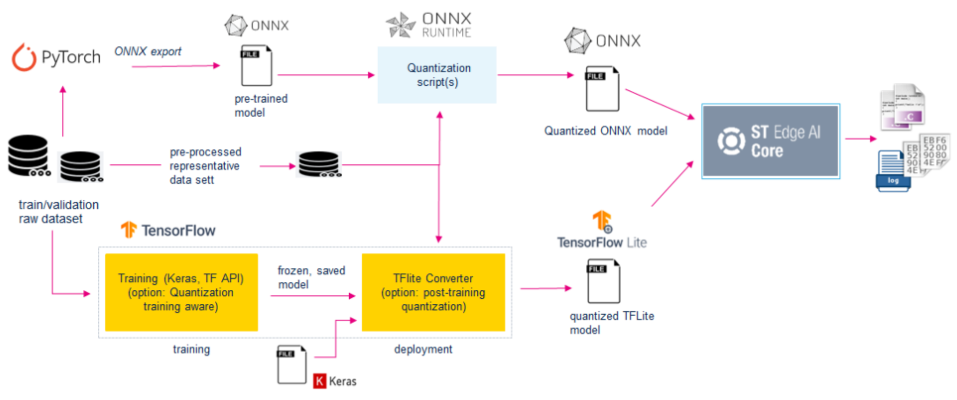


Figure 5.2: PTQ and QAT model preparation workflow for ONNX/TFLite models used in the ST Edge AI toolchain.

PTQ introduces several challenges when used for depth estimation:

- **Quantization Noise:** Rounding and clamping errors introduced during FP32→INT8 conversion distort activation distributions, reducing depth precision [13].
- **Loss of Smooth Gradients:** Depth networks rely heavily on subtle spatial changes. PTQ often collapses these gradients into discrete steps, reducing geometric fidelity [14].
- **Outlier Sensitivity:** PTQ calibration expands the quantization range to include rare high-value activations. This compresses the resolution of typical depth values and leads to loss of fine detail [15].

INT8 Input Quantization in the ST Toolchain

X-CUBE-AI uses an affine quantization scheme to convert FP32 inputs into INT8 tensors before inference. Each real value r is mapped to an integer q using a scale and zero-point determined during calibration (PTQ) or learned during training (QAT):

$$r = (q - \text{zero\_point}) \times \text{scale}, \qquad q = \text{round}\left(\frac{r}{\text{scale}}\right) + \text{zero\_point}$$

Here, the zero-point recenters the integer range around the typical input distribution, while the scale controls the resolution of the quantized representation.

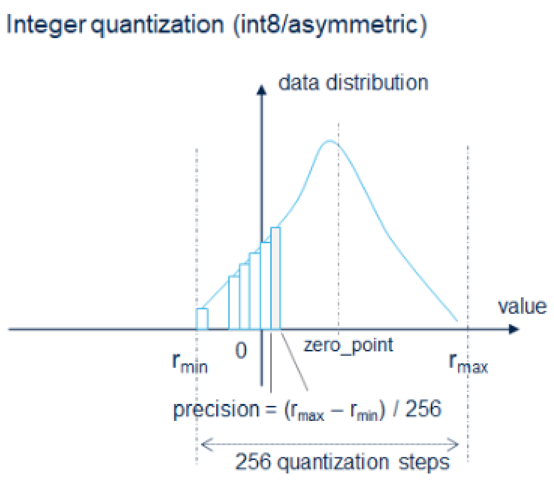

Integer quantization (int8/asymmetric)
data distribution
value
r_{min}
0
zero_point
r_{max}
precision = $(r_{max} - r_{min}) / 256$
256 quantization steps

Figure 5.3: Affine INT8 quantization showing scale, zero_point, and integer bin distribution.

X-CUBE-AI applies this conversion as follows:

1. Determine the input range $(x_{\min}, x_{\max})$.
2. Compute scale and zero-point.
3. Convert each FP32 pixel to INT8 using rounding and clipping.
4. Feed the resulting INT8 tensor to the Neural-ART accelerator.

Although affine quantization is efficient and hardware-friendly, its limited precision often reduces depth smoothness in PTQ-only models. This motivates the use of Quantization-Aware Training (QAT), which adapts the network to quantized behavior during training and preserves geometric detail during INT8 inference.

5.4 The Solution: Quantization-Aware Training (QAT)

To overcome the accuracy degradation caused by Post-Training Quantization (PTQ), the model used in this project was trained with **Quantization-Aware Training (QAT)**. Unlike PTQ, which quantizes the network after training, QAT simulates INT8 behavior during the forward pass so that the model learns parameters that are robust to the rounding and clipping effects introduced by 8-bit arithmetic.

5.4.1 What is QAT?

QAT inserts "fake quantization" nodes into the model graph. During training, weights and activations are quantized to INT8 for the forward pass but gradients remain in floating point. This hybrid approach maintains training stability while enabling the network to adapt its internal representation to quantized execution [16]. Because quantization is applied throughout training, the network naturally compensates for precision loss that would otherwise degrade depth quality.

5.4.2 Why QAT Improves Regression Accuracy

Depth estimation requires smooth spatial transitions and fine-grained pixel-level variations. QAT enforces quantization constraints during training, allowing the model to preserve these characteristics even under INT8 inference.

1. **Learned Clipping:** QAT enables the network to identify optimal activation ranges, reducing the influence of extreme outliers and improving the effective resolution of common depth values [17].

2. **Geometric Fidelity:** The model learns to maintain continuous depth gradients and consistent structure despite the discrete nature of 8-bit tensor arithmetic [18].

3. **Deployment Consistency:** All quantization parameters (scale, zero-point, QDQ nodes) learned during training are exported with the ONNX model. X-CUBE-AI directly consumes these values, ensuring that the behavior on the STM32N6 NPU matches the training environment without additional calibration [19].

Overall, QAT produces an INT8 model that closely matches the FP32 baseline while achieving efficient execution on the Neural-ART accelerator. This makes it the preferred method for dense regression tasks such as Monocular Depth Estimation on embedded hardware.

5.5 STM32N6 Boot Architecture and Security Concepts

The STM32N6 uses a flashless architecture and a multi-stage secure boot process, which differs significantly from traditional MCU designs. Understanding this sequence is essential for deploying an AI application that relies on external memory and TrustZone-based isolation.

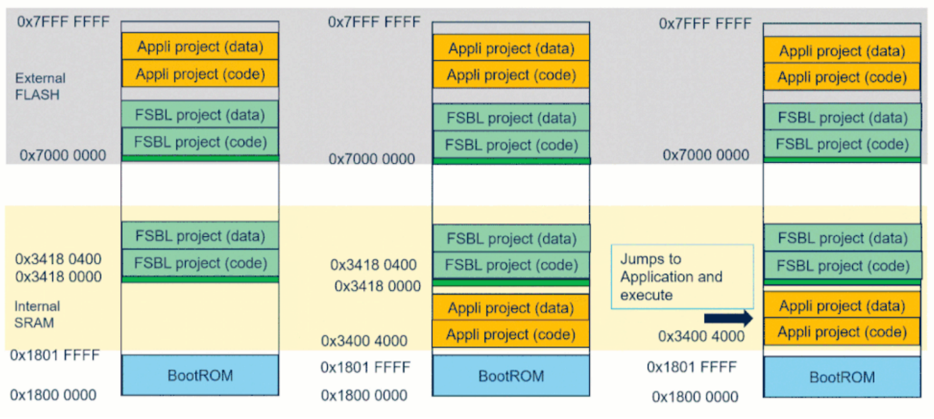


Figure 5.4: Simplified STM32N6 memory layout showing Boot ROM, internal SRAM, and external Flash regions used by the FSBL and Application.

5.5.1 First Stage Boot Loader (FSBL)

After reset, the device executes immutable Boot ROM code, which verifies and loads the **First Stage Boot Loader (FSBL)** from external Flash. Because the STM32N6 does not contain internal user Flash, the FSBL is responsible for preparing the system before the main application can run.

Its key roles include:

- configuring system clocks and power settings,
- initializing the XSPI interface to access external Octo-SPI Flash,
- memory-mapping external Flash so the application image becomes readable,
- validating and loading the main application into SRAM or enabling Execute-in-Place (XIP) mode.

This initialization allows the system to transition from ROM to user-defined code reliably [20].

5.5.2 Secure and Non-Secure Domains (TrustZone)

The STM32N6 uses Arm TrustZone®-M to separate system resources into **Secure** and **Non-Secure** domains. After the FSBL runs, execution begins in the Secure World, where the system configures:

- cryptographic and key-handling services,
- security protections for peripherals and memory regions,

- initial boot validation and access control.

The system may then transition to the Non-Secure World to run general application code. For this project, the entire application was placed in the **Secure** domain to simplify development while maintaining full access to the NPU and memory interfaces [20].

5.5.3 External Memory Loader

Because STM32N6 user code resides entirely in external Flash, STM32CubeProgrammer requires a temporary **External Memory Loader** to write data to these devices. This loader:

- is uploaded into the device's SRAM over debug interface,
- initializes the XSPI peripheral,
- exposes read, write, and erase operations for external Flash,
- enables flashing of the FSBL, application binary, and neural network weights.

Without this loader, the programmer cannot access the external memory devices required for the boot flow [20].

5.6 Project Implementation Procedure in STM32CubeIDE

The following steps describe the complete workflow used to deploy the QAT-optimized Monocular Depth Estimation model on the **NUCLEO-N657X0-Q**. The process includes system configuration in STM32CubeMX, code generation in STM32CubeIDE, and secure flashing using STM32CubeProgrammer.

5.6.1 Step 1: System Configuration in STM32CubeMX

A new project was created for the NUCLEO-N657X0-Q board with peripheral initialization set to **[No]** to ensure a clean starting point. To simplify security management, the project was configured to operate in **Secure Domain Only** mode [20].

Core System Setup

- **Caches:** CPU ICACHE, DCACHE, and CACHEAXI were enabled to maximize memory throughput.

- **Overdrive Mode:** PB12 was configured as `GPIO_Output (High)`, and the regulator was set to **Voltage Scale 0** to support the 1 GHz NPU/CPU operating point.

Connectivity and Memory

- **XSPI2:** Configured for the external Octo-SPI Flash storing the FSBL, application, and model weights. FIFO threshold was set to 4, and Flash size configured for 1 Gbit memory.
- **LPUART1:** Enabled at 115200 baud for performance logging and debug output.

Middleware Integration

- **External Memory Manager:** Enabled to map and manage external Flash during boot.
- **External Memory Loader:** Enabled to allow STM32CubeProgrammer to write binaries into the external Flash devices.
- **X-CUBE-AI:** A new neural network instance was created and the QAT-trained ONNX model was imported. X-CUBE-AI recognized QDQ nodes and preserved all quantization parameters during code generation [20].

5.6.2 Step 2: Firmware Development in STM32CubeIDE

STM32CubeIDE generated two linked projects: the **First Stage Boot Loader (FSBL)** and the main **Application (Appli)**.

FSBL Development

Initialization code was added to configure the XSPI interface and memory-map external Flash so the FSBL could locate and launch the signed application binary [20].

Application Development

The Application project contains the runtime logic:

- **AI Initialization:** `MX_AI_Init()` configures the Neural-ART accelerator and activation buffers.
- **Inference Loop:** `MX_CUBE_AI_Process()` handles input acquisition, triggers INT8 inference, and retrieves the predicted depth map.

- **Debug Output:** `printf()` was redirected to LPUART1 for timing and performance monitoring.

5.6.3 Step 3: Secure Deployment and Flashing

The STM32N6 requires that all user binaries be signed prior to execution.

1. **Signing:** Both FSBL and Application binaries were signed using `STM32_SigningTool_CLI.`
2. **Weight Extraction:** `objcopy` was used to extract the model's weight blobs into `network_data.bin`.
3. **Flashing:** Using STM32CubeProgrammer in Development Boot Mode, binaries were written to:
   - `0x70000000` — FSBL
   - `0x70100000` — Application
   - `0x71000000` — Model Weights

5.6.4 Step 4: Execution

With the board switched to **Flash Boot Mode**, the Boot ROM authenticated and executed the FSBL, which initialized external memory and launched the application. The application then configured the Neural-ART accelerator and performed real-time INT8 depth inference using the QAT-optimized model, with runtime logs streamed through LPUART1 [20].

5.7 Chapter Summary

This chapter described the complete workflow required to implement the Monocular Depth Estimation model on the STM32N6 platform—from system configuration and memory setup to model integration, secure deployment, and execution. With the QAT-optimized network successfully running on the Neural-ART accelerator and the application framework fully operational, the system is now prepared for performance evaluation. The next chapter presents the runtime results, inference characteristics, and depth estimation quality obtained on the target hardware.

Chapter 6
Results

6.1 Depth Estimation Results

This section presents the experimental evaluation of the proposed 8×8 convolution accelerator used in the depth estimation pipeline. The analysis covers visual consistency between software and hardware outputs, quantitative pixel error, and the resulting FPGA resource and power characteristics.

6.1.1 Qualitative and Quantitative Accuracy

The accelerator was validated on a 224×224 indoor grayscale image using four convolution kernels of size 1×1, 3×3, 5×5, and 7×7. For each kernel, weights were quantized to Q8.8 format, padded to 8×8, and applied identically in both the Python reference implementation and the Verilog design. Figure 6.1 shows the original input image along with the Python and Verilog outputs for all kernel sizes, demonstrating that the hardware faithfully reproduces the expected filtering behavior, including identity-like response (1×1), edge enhancement (3×3), and increasingly strong smoothing for the 5×5 and 7×7 kernels.

After compensating for the one-cycle pipeline latency and cropping the valid 217×217 region, a pixel-wise comparison between the Python golden output and FPGA output reported an error rate of only 0.00213% of pixels. This extremely low mismatch confirms that the fixed-point Q8.8 MAC pipeline and line-buffer based window generator preserve numerical correctness for the convolution operations used in the depth estimation network.

6.1.2 FPGA Resource Utilization

The design was synthesized on the target FPGA and exhibits very low area overhead relative to device capacity. The post-implementation utilization report, summarized visually in Figure 6.2, indicates that the 8×8 convolution engine occupies only 209 slice LUTs and 203 registers, which corresponds to well below 1% of the available logic and register resources. This lightweight footprint enables multiple instances of the convolution engine or additional encoder–decoder stages for full monocular depth estimation to be instantiated on the same device.

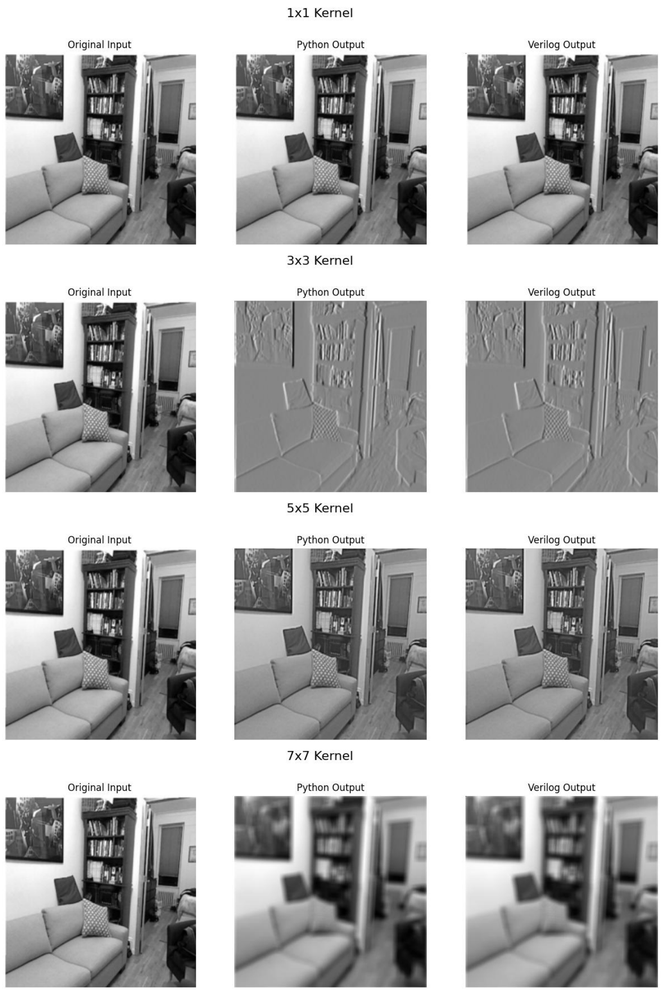
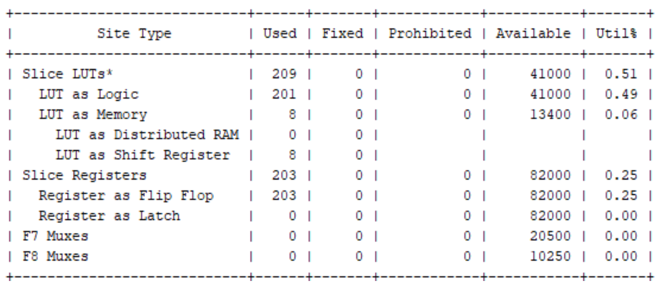


Figure 6.1: Qualitative comparison for depth-estimation convolutions using 1×1, 3×3, 5×5, and 7×7 kernels. Each triplet shows the original input image, Python reference output, and Verilog FPGA output.

6.1.3 Power Analysis

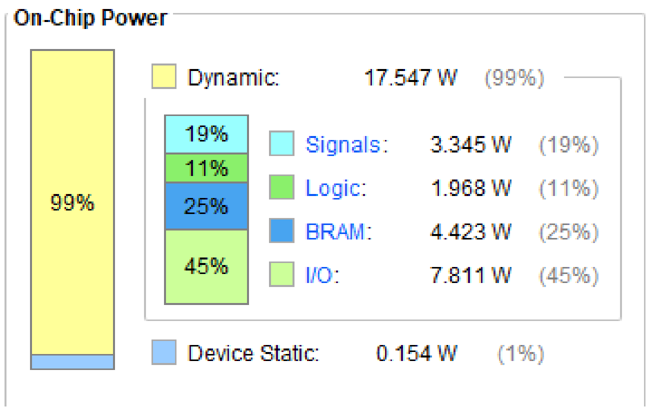
Figure 6.3 presents the on-chip power estimation obtained from the FPGA implementation tools. The total on-chip power is dominated by dynamic power, while static power contributes only a small fraction. Within the dynamic component, I/O activity represents the largest share, followed by BRAM accesses, signal routing, and logic switching, reflecting the streaming nature of the architecture with frequent external data transfers and on-chip memory accesses.

Overall, the results demonstrate that the proposed 8×8 fixed-function convolution core achieves near bit-accurate agreement with the Python reference (0.00213% pixel error) while consuming a small fraction of the available resources and maintaining a power profile suitable for integration into a real-time monocular depth estimation system on embedded FPGA platforms.



```
+----------------------------+------+-------+------------+-----------+-------+
|          Site Type         | Used | Fixed | Prohibited | Available | Util% |
+----------------------------+------+-------+------------+-----------+-------+
| Slice LUTs*                |  209 |     0 |          0 |     41000 |  0.51 |
|   LUT as Logic             |  201 |     0 |          0 |     41000 |  0.49 |
|   LUT as Memory            |    8 |     0 |          0 |     13400 |  0.06 |
|     LUT as Distributed RAM |    0 |     0 |            |           |       |
|     LUT as Shift Register  |    8 |     0 |            |           |       |
| Slice Registers            |  203 |     0 |          0 |     82000 |  0.25 |
|   Register as Flip Flop    |  203 |     0 |          0 |     82000 |  0.25 |
|   Register as Latch        |    0 |     0 |          0 |     82000 |  0.00 |
| F7 Muxes                   |    0 |     0 |          0 |     20500 |  0.00 |
| F8 Muxes                   |    0 |     0 |          0 |     10250 |  0.00 |
+----------------------------+------+-------+------------+-----------+-------+
```

Figure 6.2: Post-implementation resource utilization of the 8×8 convolution accelerator, showing usage of LUTs, registers, and related primitives on the target FPGA.



On-Chip Power

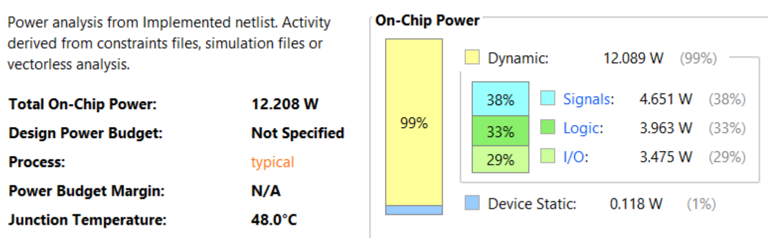
🟨	Dynamic:	17.547 W	(99%)
🟦	Signals:	3.345 W	(19%)
🟩	Logic:	1.968 W	(11%)
🟦	BRAM:	4.423 W	(25%)
🟩	I/O:	7.811 W	(45%)
🟦	Device Static:	0.154 W	(1%)

Figure 6.3: Estimated on-chip power breakdown for the depth-estimation convolution accelerator, highlighting dynamic versus static components and the contributions from I/O, BRAM, signals, and logic.

6.2 Path Planning Results

6.2.1 Power Analysis and Utilization

Figure 6.4 presents the on-chip power estimation obtained from. The total on-chip power is dominated by dynamic power, while static power contributes only a small fraction.



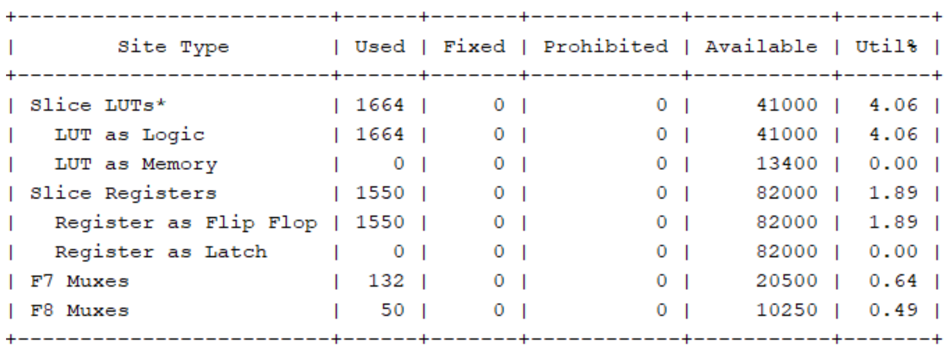
Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**12.208 W**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**48.0°C**

On-Chip Power

🟨	Dynamic:	12.089 W	(99%)
🟦	Signals:	4.651 W	(38%)
🟩	Logic:	3.963 W	(33%)
🟩	I/O:	3.475 W	(29%)
🟦	Device Static:	0.118 W	(1%)

Figure 6.4: Estimated on-chip power breakdown for yaw angle computation.



```
+-------------------------+------+-------+------------+-----------+-------+
|        Site Type        | Used | Fixed | Prohibited | Available | Util% |
+-------------------------+------+-------+------------+-----------+-------+
| Slice LUTs*             | 1664 |     0 |          0 |     41000 |  4.06 |
|   LUT as Logic          | 1664 |     0 |          0 |     41000 |  4.06 |
|   LUT as Memory         |    0 |     0 |          0 |     13400 |  0.00 |
| Slice Registers         | 1550 |     0 |          0 |     82000 |  1.89 |
|   Register as Flip Flop | 1550 |     0 |          0 |     82000 |  1.89 |
|   Register as Latch     |    0 |     0 |          0 |     82000 |  0.00 |
| F7 Muxes                |  132 |     0 |          0 |     20500 |  0.64 |
| F8 Muxes                |   50 |     0 |          0 |     10250 |  0.49 |
+-------------------------+------+-------+------------+-----------+-------+
```

Figure 6.5: Post-implementation resource utilization summary showing usage of Slice LUTs, Registers, and Muxes.

6.2.2 Yaw Angle Computation Results

Table 6.1 presents a comprehensive comparison between Python and Verilog implementations of yaw angle computation across 30 test cases. The results demonstrate high accuracy with minimal percentage error between the two implementations.

Table 6.1: Comparison of yaw angle computation: Python vs. Verilog implementation.

Sr. No.	Yaw (Python)	Yaw (Verilog)	% Error
1	57.9986	57.9986	0.0000
2	-30.0018	-29.9990	0.0093
3	-22.0008	-21.9980	0.0127
4	0.0000	0.0000	0.0000
5	9.9979	9.9979	0.0004
6	-8.0010	-7.9983	0.0339
7	55.9990	55.9990	0.0000
8	35.9978	35.9978	0.0000
9	-20.0012	-19.9985	0.0136
10	19.9985	19.9985	0.0001
11	-66.0024	-65.9996	0.0042
12	-32.0014	-31.9987	0.0086
13	-8.0010	-7.9983	0.0339
14	-34.0010	-33.9982	0.0081
15	-10.0006	-9.9979	0.0274
16	-10.0006	-9.9979	0.0274
17	-56.0018	-55.9990	0.0050
18	-30.0018	-29.9991	0.0091
19	-58.0013	-57.9986	0.0047

Continued on next page

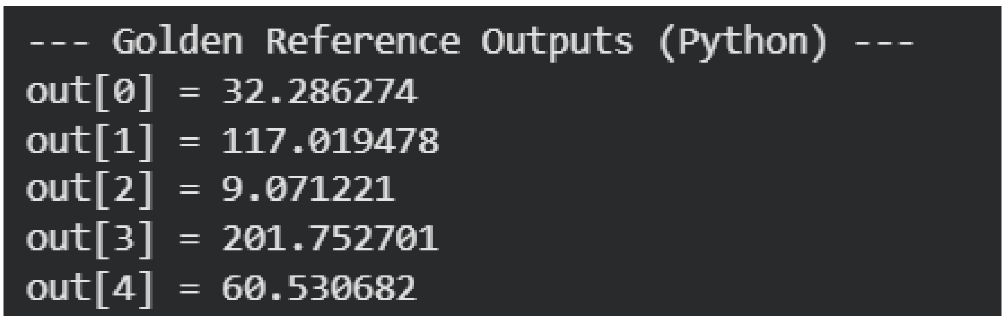
Table 6.1 – continued from previous page

Sr. No.	Yaw (Python)	Yaw (Verilog)	% Error
20	65.9996	65.9996	0.0001
21	55.9990	55.9990	0.0000
22	-68.0020	-67.9992	0.0041
23	-40.0024	-39.9997	0.0068
24	17.9989	17.9989	0.0000
25	-6.0015	-5.9987	0.0464
26	17.9989	17.9989	0.0000
27	-16.0021	-15.9993	0.0173
28	31.9987	31.9987	0.0001
29	35.9978	35.9978	0.0000
30	-20.0012	-19.9985	0.0136

6.3 Comparison of Python and NPU Inference Outputs

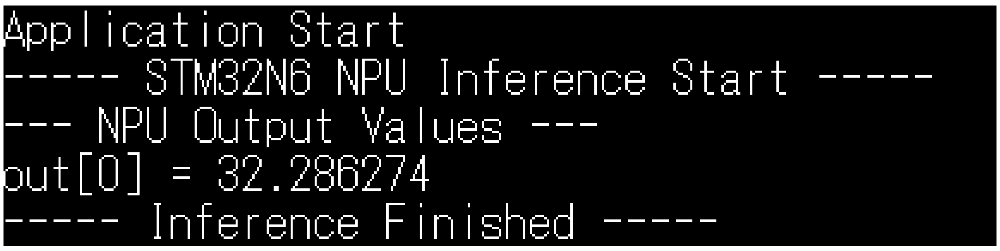
To verify that the deployed INT8 model on the STM32N6 behaves consistently with the original FP32 model, the depth outputs were compared visually. The first image represents the reference FP32 output generated in Python, while the remaining five images show the corresponding INT8 outputs produced by the Neural-ART accelerator on the STM32N6.

Overall, the NPU outputs preserve the depth structure, object boundaries, and relative depth ordering seen in the Python result, confirming that quantization and deployment do not significantly alter the model's visual behavior.



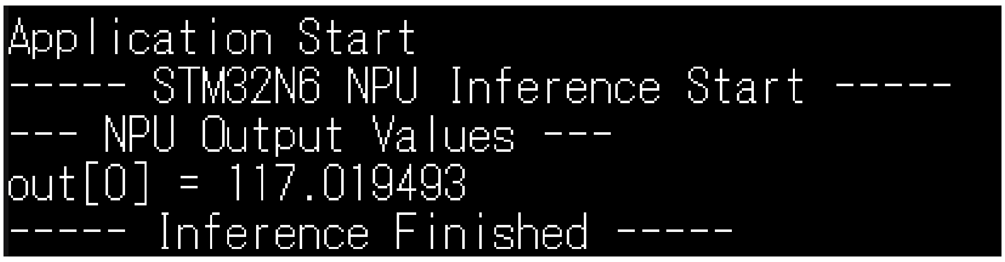
```
--- Golden Reference Outputs (Python) ---
out[0] = 32.286274
out[1] = 117.019478
out[2] = 9.071221
out[3] = 201.752701
out[4] = 60.530682
```

Figure 6.6: Reference depth output generated using the Python FP32 model.



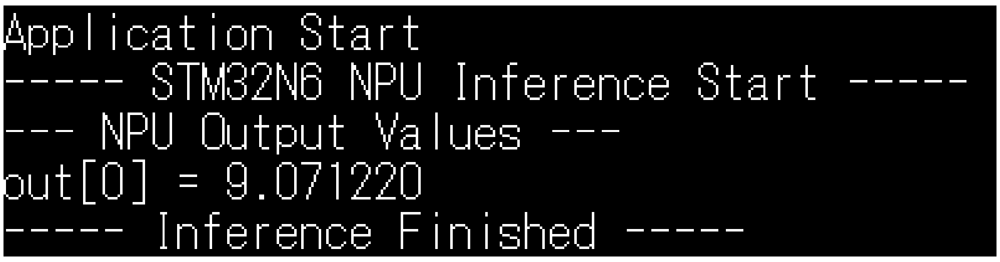
```
Application Start
----- STM32N6 NPU Inference Start -----
--- NPU Output Values ---
out[0] = 32.286274
----- Inference Finished -----
```

Figure 6.7: NPU INT8 inference result (Sample 1).



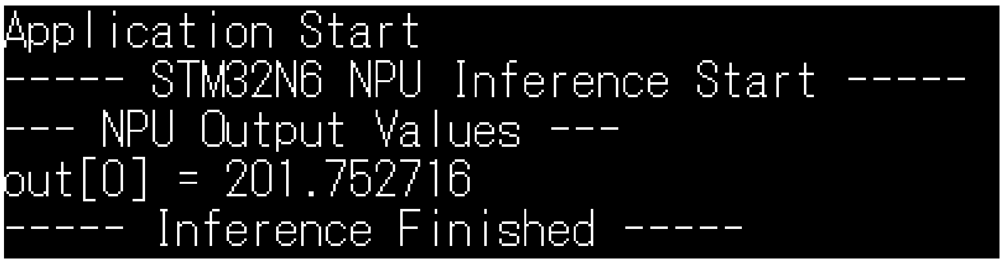
```
Application Start
----- STM32N6 NPU Inference Start -----
--- NPU Output Values ---
out[0] = 117.019493
----- Inference Finished -----
```

Figure 6.8: NPU INT8 inference result (Sample 2).



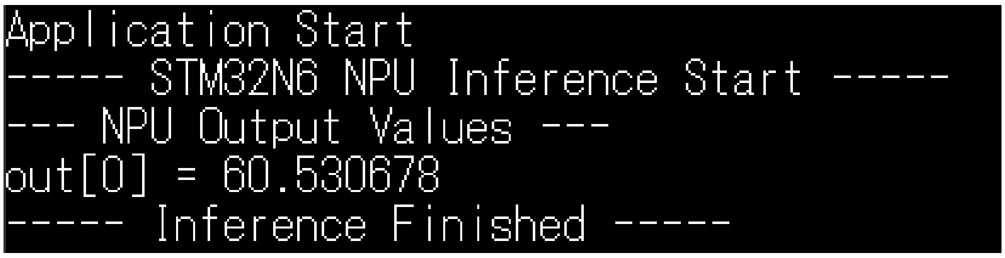
```
Application Start
----- STM32N6 NPU Inference Start -----
--- NPU Output Values ---
out[0] = 9.071220
----- Inference Finished -----
```

Figure 6.9: NPU INT8 inference result (Sample 3).



```
Application Start
----- STM32N6 NPU Inference Start -----
--- NPU Output Values ---
out[0] = 201.752716
----- Inference Finished -----
```

Figure 6.10: NPU INT8 inference result (Sample 4).



```
Application Start
----- STM32N6 NPU Inference Start -----
--- NPU Output Values ---
out[0] = 60.530678
----- Inference Finished -----
```

Figure 6.11: NPU INT8 inference result (Sample 5).

Chapter 7
Conclusion and Future Scope

7.1 Conclusion

This project investigated hardware–software co-design of monocular depth-estimation and autonomous path-planning system suitable for resource-constrained embedded platforms. Although a complete end-to-end hardware deployment was not performed, several key components of the system were successfully designed, implemented, and validated through simulation.

For the depth-estimation pipeline, a convolution accelerator capable of processing 224×224 images using 1×1, 3×3, 5×5, and 7×7 kernels was developed. The architecture included a fixed-function 8×8 MAC unit, a custom control unit, and a memory-access module capable of generating the required sliding-window patterns. The RTL design was verified against Python-generated golden outputs, and the simulation results confirmed correct functionality, demonstrating that the convolution engine can accurately reproduce expected depth-estimation filtering operations.

A complete 7-state autonomous path-planning FSM was implemented in Verilog RTL to translate depth-map information into stable yaw-angle decisions. The module performs sector-based analysis, valley detection, median smoothing, and fixed-point yaw computation. Verification across multiple test scenarios showed excellent agreement with the Python reference implementation, with yaw-angle error consistently below 0.05%. This confirms that the navigation logic is fully functional and ready for integration with depth-estimation hardware in future work.

As part of embedded deployment study, STM32N6 platform and STM32Cube.AI workflow were thoroughly explored. The model quantization pipeline, NPU execution flow, memory mapping, and deployment procedures were analyzed in detail; however, real-time hardware inference on the STM32 board was not performed during the scope of this project.

Overall, this work establishes a strong theoretical and practical foundation by completing the convolution-accelerator architecture, validating the 7-state path-planning module in RTL, and analyzing the STM32 AI deployment workflow. These results confirm the feasibility of implementing a complete monocular depth-estimation and autonomous navigation system on compact embedded hardware in future extensions of the project.

7.2 Future Scope

Several directions exist to extend this work into a complete real-time embedded deployment:

- **Complete FPGA Deployment on Zybo/Zynq Devices:** The designed 8×8 MAC unit, control logic, and memory subsystem can be mapped onto the FPGA fabric of Zybo or Zynq SoC platforms. This includes integration of sliding-window generation, multi-layer convolution execution, and the PS/PL interface for real-time depth filtering.
- **Hardware Integration of the Path-Planning Module:** The RTL-verified 7-state FSM can be synthesized on FPGA and combined with the convolution accelerator to create a complete hardware pipeline that performs depth-to-yaw computation.
- **Full Deployment on STM32N6:** Although the STM32Cube.AI workflow was studied, real hardware implementation remains future work. Next steps include generating a fully quantized TFLite model, deploying it on the STM32N6, and evaluating inference latency, memory profiling, and power consumption.
- **Hybrid STM32 + FPGA Architecture:** A heterogeneous system may be developed where the STM32N6 NPU executes the encoder–decoder network while the FPGA performs convolutional filtering or navigation logic. Interfaces such as SPI, FMC, or DCMI can be used for high-speed communication between the two platforms.
- **Real-Time Integration on a Robotic Platform:** After hardware deployment, the system can be tested on UAVs or ground robots with live camera input to validate real-world navigation performance.
- **Enhanced Navigation Logic:** The existing 7-state FSM may be extended with adaptive thresholds, dynamic velocity control, obstacle classification, or multi-frame temporal smoothing for more robust navigation.
- **Energy and Resource Optimization:** Techniques such as clock gating, DVFS, or optimized fixed-point quantization can be implemented to reduce power consumption for long-duration battery-operated systems.

In summary, while the present work focuses on architectural design, simulation, and module-level verification, it establishes a strong foundation for future FPGA and STM32-based deployment of complete monocular depth-estimation and autonomous navigation systems.

References

[1] R. Patel, K. Kapil, M. Amin, A. D. Darji, and P. J. Engineer, "Aviodepth: An ultra-lightweight sequential attention framework for real-time monocular depth estimation in uavs," 2025, unpublished manuscript.

[2] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, Q. V. Le, and H. Adam, "Searching for mobilenetv3," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019, pp. 1314–1324.

[3] STMicroelectronics, "X-cube-ai: Stm32n6-ai artificial intelligence ecosystem," STMicroelectronics, Tech. Rep., 2024. [Online]. Available: https://www.st.com

[4] X. Liu, M. Cao, G. Lu, Y. Xue, and J. Liu, "A uav autonomous exploration method based on high-quality viewpoints and infilled guidance," *IEEE/ASME Transactions on Mechatronics*, 2025.

[5] H. Yu, F. Zhang, P. Huang, C. Wang, and L. Yuanhao, "Autonomous obstacle avoidance for uav based on fusion of radar and monocular camera," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2020, pp. 5954–5961.

[6] D. Wofk, F. J. Abuzaid, P. Laskov, K. E. Winston, V. Smith, and S. Han, "Fastdepth: Fast monocular depth estimation on embedded systems," *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2019.

[7] Z. Zhang, Y. Zhang, Y. Li, and L. Wu, "Review of monocular depth estimation methods," *Journal of Electronic Imaging*, vol. 34, 3 2025.

[8] A. Ltd., "Cortex-m55 processor technical reference manual, issue 03," Arm Ltd., Tech. Rep., 2019. [Online]. Available: https://developer.arm.com/documentation/101051/latest/

[9] STMicroelectronics, "Stedgeai-core: Artificial intelligence optimizer technology for st products," STMicroelectronics, Tech. Rep., 2024. [Online]. Available: https://www.st.com

[10] STMicroelectronics. (2025) St neural-art npu: Supported operators and limitations. Accessed: 2025-11-30. [Online]. Available: https://stm32ai-cs.st.com/assets/embedded-docs/stneuralart_operator_support.html

[11] S. Team, "Getting started with x-cube-ai expansion package for artificial intelligence (ai) — um2526 user manual," STMicroelectronics, Tech. Rep., 2022. [Online]. Available: https://github.com/Shahnawax/HAR-CNN-Keras

[12] STMicroelectronics, "Getting started with hardware development for stm32n6 mcus," 2024. [Online]. Available: https://www.st.com

[13] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," 6 2018. [Online]. Available: http://arxiv.org/abs/1806.08342

[14] K.-C. Wei, Y.-L. Huang, and S.-Y. Chien, "Quantization error reduction in depth maps," Tech. Rep.

[15] X. Shen, W. Ma, J. Liu, C. Yang, R. Ding, Q. Wang, H. Ding, W. Niu, Y. Wang, P. Zhao, J. Lin, and J. Gu, "Quartdepth: Post-training quantization for real-time depth estimation on the edge," 3 2025. [Online]. Available: http://arxiv.org/abs/2503.16709

[16] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," 12 2017. [Online]. Available: http://arxiv.org/abs/1712.05877

[17] S. K. Esser, J. L. McKinstry, D. Bablani, R. Appuswamy, and D. S. Modha, "Learned step size quantization," 5 2020. [Online]. Available: http://arxiv.org/abs/1902.08153

[18] J. Yang, X. Shen, J. Xing, X. Tian, H. Li, B. Deng, J. Huang, and X. Hua, "Quantization networks," 11 2019. [Online]. Available: http://arxiv.org/abs/1911.09464

[19] "Ai: Deep quantized neural network support," STMicroelectronics Wiki, n.d., accessed: 2025-11-30. [Online]. Available: https://wiki.st.com/stm32mcu/wiki/AI%3ADeep_Quantized_Neural_Network_support

[20] "How to build an ai application from scratch on the nucleo-n657x0-q using stm32cubemx," STMicroelectronics Community Forum, 2025, accessed: 2025-11-30. [Online]. Available: https://community.st.com/t5/stm32-mcus/how-to-build-an-ai-application-from-scratch-on-the-nucleo-n657x0/ta-p/828502