Jaypee Institute of Information Technology ,

Noida Sec - 62

Hardware Accelerator

Supervisor: Dr. Gaurav Verma

Members:

Vidhi Agarwal - 22802006

M S Avantika Lakshmi - 22802004

ABSTRACT

In recent years, personal safety has become a significant concern, especially for vulnerable individuals in unfamiliar or potentially dangerous situations. Advances in wearable technology present an opportunity to provide real-time safety solutions that are easily accessible and effective. This project, titled **SafetyPulse**, introduces a **smartwatch-based personal safety application** designed to enhance user security through **live location tracking, instant emergency alerts, and seamless integration with mapping services.**

SafetyPulse is developed with the primary goal of providing users with a reliable method to **communicate their location and send distress signals during emergencies.** The core functionality of the app leverages GPS technology to continuously monitor and update the user's location. In the event of an emergency, the user can activate an SOS alert, which sends an instant message containing the real-time location coordinates to predefined contacts. The integration with Google Maps enables recipients to view the sender's exact location, facilitating swift responses from friends, family, or emergency services.

The application is developed on the Android platform using Android Studio and Kotlin, and it includes a range of features specifically tailored to improve user safety. Key functionalities of SafetyPulse include GPS-based location tracking, emergency SMS with location sharing, and an intuitive user interface designed for quick access and ease of use. To optimize the app's responsiveness, special attention has been given to performance efficiency and battery management, ensuring the app remains operational during critical moments without significantly impacting device battery life.

Beyond technical innovation, SafetyPulse addresses the growing need for accessible personal safety solutions. By focusing on simplicity, reliability, and effective communication, this project aims to empower individuals with a tool that enhances their sense of security. SafetyPulse has the

potential to benefit diverse user groups, from commuters and travelers to outdoor enthusiasts and those concerned about their personal safety.

The project is structured to demonstrate both the technical aspects of building a location-tracking and alert-based application and the real-world implications of providing a dependable safety tool. This report details the design, development process, challenges encountered, and testing results, along with insights for potential future enhancements. The successful implementation of SafetyPulse underscores the importance of leveraging technology for personal safety and opens avenues for further innovation in the field of wearable safety devices.

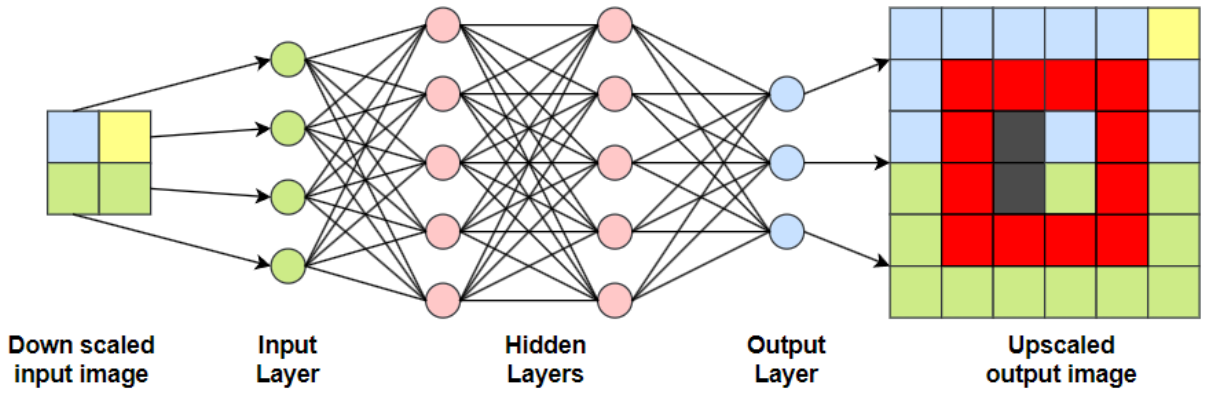
Table of Contents

Introduction

1.1 Overview of Deep Learning and YOLO

1.1.1 Evolution of Deep Learning in Computer Vision

Deep Learning is a subset of Artificial Intelligence (AI) that imitates how the human brain works to create systems capable of learning and performing complex tasks. Its core strength lies in neural networks—specifically, deep neural networks with multiple layers of artificial neurons. These layers allow the model to learn hierarchical patterns in data, extracting increasingly abstract features as information moves through the network. In simple terms, Deep Learning enables a computer to simulate a network of neurons, performing countless simple operations repeatedly. Over time, these networks adjust their internal connections based on the data, making them highly effective for tasks like object detection.The following diagram describes more or less that.



Deep learning has fundamentally transformed computer vision over the past decade, shifting paradigms from traditional handcrafted feature extraction to end-to-end trainable neural

networks. Convolutional Neural Networks (CNNs) in particular have become the cornerstone architecture for visual tasks, systematically outperforming conventional methods by automatically learning hierarchical representations directly from raw pixel data.

In the field of object detection, progress has been marked by distinct generations of approaches:

Region-based methods: Starting with R-CNN (2014), which proposed a selective search for region proposals followed by CNN classification

Fast R-CNN and Faster R-CNN: Improving efficiency through shared feature computation and region proposal networks

Single-shot detectors: Including SSD (Single Shot MultiBox Detector) and YOLO, which eliminated the separate region proposal step entirely

1.1.2 The YOLO Architecture and Its Innovations

YOLO (You Only Look Once), introduced by Joseph Redmon et al. in 2016, represented a breakthrough in real-time object detection by framing the problem as a single regression task. Its key innovations include:

Grid-based prediction: The image is divided into an S×S grid, with each cell responsible for predicting objects centered within it.

Simultaneous bounding box and class prediction: Each grid cell predicts B bounding boxes with confidence scores and C conditional class probabilities.

End-to-end optimization: The entire network is trained jointly to optimize detection performance directly.

Unified architecture: A single neural network predicts bounding boxes and class probabilities simultaneously across the entire image.

The original YOLO architecture uses a 24-layer CNN for feature extraction followed by fully connected layers for the final detection. This structure allows it to process images at

approximately 45 frames per second on high-end GPUs, with a smaller variant reaching up to 155 FPS with some reduction in accuracy.

1.1.3. YOLO Versions and Their Improvements

YOLO has evolved significantly through multiple iterations:

YOLOv1 (2016): The original implementation demonstrated the viability of single-shot detection but struggled with small objects and precise localization.

YOLOv2/YOLO9000 (2017): Added batch normalization, anchor boxes, dimension clusters, and multi-scale training while introducing a hierarchical classification system capable of detecting over 9,000 object categories.

YOLOv3 (2018): Enhanced the backbone with a deeper feature extractor (Darknet-53) and implemented feature pyramid networks for better multi-scale object detection, significantly improving performance on small objects.

YOLOv4 (2020): Incorporated numerous architectural improvements including CSPDarknet53 backbone, PANet feature aggregation, and introduced advanced training techniques like Mosaic data augmentation and DropBlock regularization.

YOLOv5 (2020): Optimized for production deployment with significant speed improvements while maintaining competitive accuracy.

YOLO-Tiny variants: Specifically designed for resource-constrained environments, these lightweight versions sacrifice some accuracy for dramatically reduced computational requirements, making them suitable for embedded systems like the PYNQ-Z2.

1.1.4 Network Architecture Details

The core YOLO architecture consists of:

Backbone network: Responsible for feature extraction (e.g., Darknet variants)

Neck: Feature aggregation layers that combine features from different scales (e.g., FPN, PAN)

Head: The final layers that transform feature maps into detection predictions

For a Tiny-YOLO variant suitable for FPGA implementation, the architecture is typically streamlined to include:
Fewer convolutional layers (13-19 layers compared to 53+ in full models)
Reduced channel dimensions throughout the network
Fewer anchor boxes for prediction
Simplified feature pyramid structures

1.1.5 Deep Learning on FPGAs: Challenges and Approaches

Implementing deep neural networks like YOLO on FPGAs presents distinct challenges:

Memory constraints: FPGA on-chip memory is limited compared to GPU VRAM
Computational precision: Reduced precision (INT8, INT16) is often necessary
Dataflow optimization: Careful scheduling of operations to maximize parallelism
Resource utilization: Balancing DSP blocks, LUTs, and memory resources

Common approaches to address these challenges include:

Model quantization: Converting floating-point operations to fixed-point
Network pruning: Removing redundant connections or filters
Layer fusion: Combining consecutive operations to reduce memory transfers
Tiling strategies: Processing subsets of data to fit within on-chip memory
Pipeline parallelism: Creating deeply pipelined architectures to process multiple inputs simultaneously

For the PYNQ-Z2 implementation specifically, the Tiny-YOLO variant requires careful optimization to fit within the available resources of the Zynq-7020 SoC while maintaining acceptable detection performance. This typically involves 8-bit quantization, strategic layer

allocation between the ARM processor and FPGA fabric, and custom acceleration for computation-intensive operations like convolutions.

1.1.6 Domain-Specific Applications of YOLO

The speed-accuracy tradeoff offered by YOLO has enabled numerous real-world applications:

Autonomous driving: Detecting vehicles, pedestrians, and road signs in real-time
Surveillance systems: Identifying and tracking people or objects of interest
Manufacturing quality control: Detecting defects on production lines
Agricultural monitoring: Identifying crops, weeds, or livestock
Retail analytics: Tracking customer movements and product interactions
Healthcare: Assisting in medical image analysis when rapid results are needed

For embedded applications like those targeted by this project, YOLO's efficiency and adaptability make it particularly valuable for creating intelligent edge devices that can operate independently of cloud connectivity, with lower latency and improved privacy.

1.2. Introduction to FPGA and PYNQ-Z2

1.2.1. FPGA Technology Overview

Field-Programmable Gate Arrays (FPGAs) are semiconductor devices containing an array of programmable logic blocks and reconfigurable interconnects. Unlike fixed processors (CPUs/GPUs), FPGAs can be reprogrammed to implement custom digital circuits after manufacturing, offering significant advantages for specialized computing tasks:

- **Parallelism:** Ability to perform multiple operations simultaneously
- **Low latency:** Direct hardware implementation reduces processing delays
- **Power efficiency:** Custom circuits can be more energy-efficient than general-purpose

processors

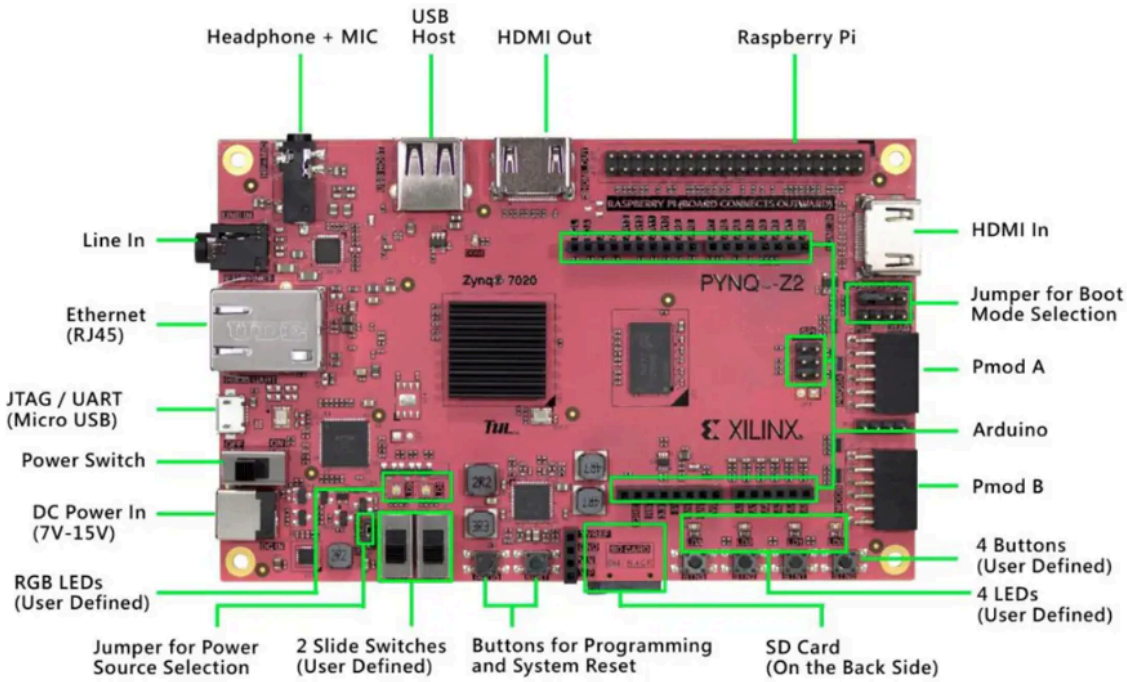
- **Flexibility:** Can be reconfigured for different applications without changing hardware

These characteristics make FPGAs increasingly attractive for deep learning inference, where specific operations like convolutions can benefit from custom hardware acceleration.

1.2.2 The PYNQ-Z2 Platform

The PYNQ-Z2 board is an affordable development platform based on the Xilinx Zynq-7000 SoC architecture. It combines the software programmability of an ARM processor with the hardware configurability of an FPGA. Key specifications relevant to this project include:

- **Processor:** Dual-core ARM Cortex-A9 (processing system, PS)
- **FPGA:** Artix-7 FPGA fabric (programmable logic, PL)
- **Memory:** 512MB DDR3 (shared between PS and PL)
- **Storage:** MicroSD card slot for booting and data storage
- **Video I/O:** HDMI input and output ports
- **Connectivity:** Ethernet, USB, and various GPIO pins



The board's resource constraints are important considerations for this project:

- Limited on-chip memory (Block RAM)
- Approximately 13,300 logic slices
- 220 DSP slices for efficient arithmetic operations
- 630 KB of Block RAM

1.2.3. The PYNQ Framework

PYNQ (Python Productivity for Zynq) is a key component of this project. This open-source framework allows developers to program Zynq devices using Python rather than traditional hardware description languages like VHDL or Verilog. Key features include:

1. **Jupyter Notebook Interface**: Interactive development environment running on the board
2. **Overlays**: Pre-built FPGA designs that can be loaded at runtime

3. **Python APIs**: Allow software to interact with hardware accelerators
4. **Hardware Libraries**: Ready-to-use IP blocks for common functions

As used in the reference project, PYNQ enables a high-level approach to deploying deep learning on FPGA:

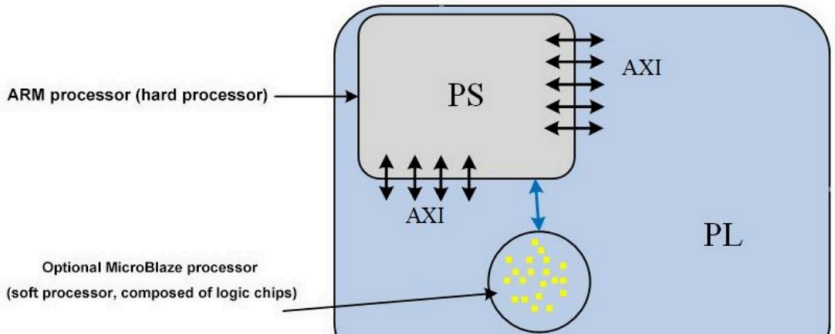
```
from pynq import Overlay
overlay = Overlay("yolo_overlay.bit")
accelerator = overlay.yolo_engine
```

1.2.4. Hardware-Software Co-Design Approach

The PYNQ-Z2 platform facilitates hardware-software co-design, which is central to the project implementation:

1. **Processing System (PS)**: The ARM cores handle high-level tasks such as:
    - Loading and preprocessing images
    - Post-processing detection results
    - Managing user interface and data flow
    - Controlling hardware accelerators
2. **Programmable Logic (PL)**: The FPGA fabric implements:
    - Compute-intensive CNN layers
    - Memory-efficient dataflow for the YOLO network
    - Custom optimized processing elements

The AXI (Advanced eXtensible Interface) protocol facilitates communication between these components, allowing for efficient data transfer between software and hardware domains.



1.2.5 Quantization for FPGA Implementation

A critical aspect of the YOLO on PYNQ-Z2 implementation is quantization - converting floating-point operations to fixed-point or integer operations. The project employs 8-bit quantization to:

- Reduce memory requirements
- Simplify arithmetic operations
- Enable more efficient use of FPGA resources
- Increase throughput

This approach trades a small reduction in accuracy for significant improvements in speed and resource utilization, making real-time object detection feasible on the resource-constrained PYNQ-Z2 platform.

1.2.6. Design Considerations for the Project

For the YOLO implementation on PYNQ-Z2, several design considerations guided the approach:

1. **Model selection:** Using Tiny-YOLO variant to fit within resource constraints
2. **Memory management:** Carefully orchestrating data movement between external DDR and on-chip BRAM
3. **Computation scheduling:** Optimizing the execution sequence to maximize resource utilization
4. **Interface design:** Creating efficient communication between software and hardware components

These considerations form the foundation for the project implementation, balancing performance

requirements against the platform's limitations.

1.3. Project Objectives and Scope

1.3.1 Primary Objectives

This project aims to implement and optimize the YOLO object detection algorithm on the PYNQ-Z2 platform with several specific goals:

1. **Real-time object detection:** Achieve inference speeds suitable for real-time applications (target: 5+ frames per second) on the PYNQ-Z2 board
2. **Efficient resource utilization:** Maximize the use of available FPGA resources while staying within the constraints of the Zynq-7020 device
3. **Practical accuracy:** Maintain acceptable detection accuracy for common objects despite model optimizations
4. **Demonstration system:** Create a functional end-to-end system capable of processing video input and displaying detection results

1.3.2. Technical Approach

To accomplish these objectives, the project employs a structured technical approach:

1. **Model adaptation:**
    - Using Tiny-YOLO architecture to fit within FPGA resources
    - Implementing 8-bit quantization to reduce computational requirements
    - Preserving critical features for maintaining detection accuracy
2. **Acceleration strategy:**
    - Offloading convolutional layers to FPGA fabric
    - Implementing custom dataflow for efficient data movement
    - Utilizing DSP slices for arithmetic operations
    - Creating specialized hardware for activation functions
3. **System integration:**
    - Developing a software pipeline for preprocessing and post-processing
    - Creating hardware-software interfaces using the PYNQ framework
    - Implementing efficient memory management between PS and PL

1.3.3 Scope Definition

1.3.4. Scope Limitations

To maintain feasibility within the project constraints, certain aspects are intentionally excluded:

- Training of custom YOLO models (using pre-trained weights instead)
- Ultra-high resolution image processing
- Multi-camera or stereo vision capabilities
- Advanced tracking of detected objects across frames
- Custom dataset preparation or annotation

1.3.5. Expected Outcomes

Upon completion, the project will deliver:

1. A functional implementation of Tiny-YOLO on the PYNQ-Z2 platform
2. Documented hardware design and software components
3. Performance benchmarks showing inference speed and accuracy
4. A demonstration system capable of processing images/video and displaying detection results
5. Analysis of resource utilization and optimization strategies

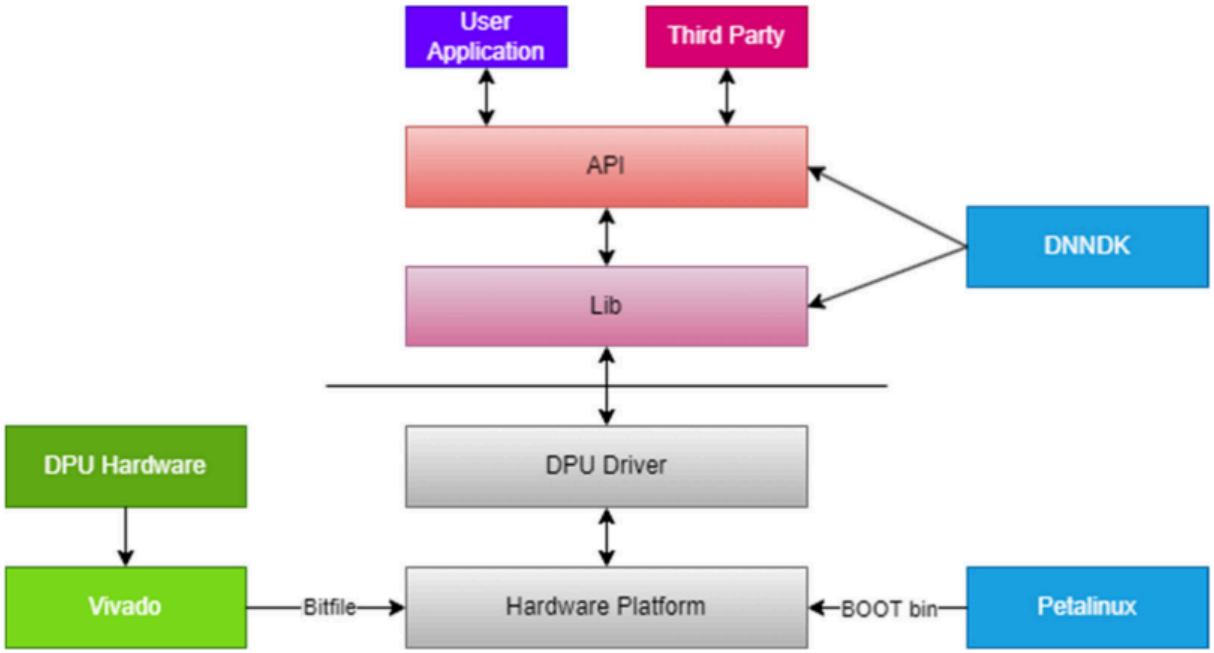
1.3.6. Significance and Applications

This implementation demonstrates the viability of deploying sophisticated deep learning models on affordable embedded platforms, opening possibilities for numerous applications:

- Edge computing devices: Autonomous systems that can perform visual recognition locally
- IoT nodes: Smart cameras and sensors with on-device intelligence
- Educational platforms: Teaching hardware-software co-design for AI applications
- Prototyping: Rapid development of computer vision applications before custom ASIC design

The project showcases how modern deep learning can be adapted to resource-constrained environments while maintaining practical functionality, bridging the gap between laboratory AI research and real-world deployed systems.

1.4. Development Flow Overview



The implementation of YOLO on the PYNQ-Z2 platform follows a structured development methodology that addresses both hardware and software aspects of the system. This systematic approach ensures efficient resource utilization while maintaining the detection capabilities of the neural network.

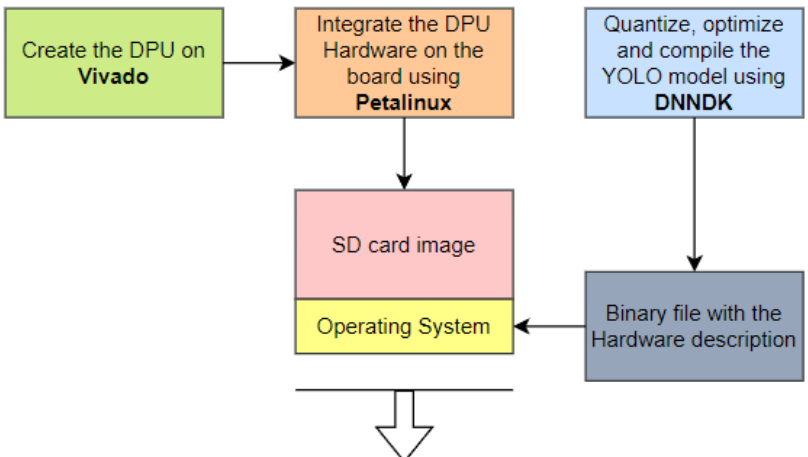
1.4.1. System Architecture Overview

The first diagram illustrates the complete system architecture for implementing YOLO on PYNQ-Z2:

This architecture consists of several key components:

1. **DPU Hardware**: The Deep Processing Unit (DPU) is a specialized accelerator for neural network inference, implemented on the FPGA fabric of the PYNQ-Z2.
2. **Vivado**: Xilinx's development environment used to design and synthesize the DPU hardware, generating a bitfile that configures the FPGA.
3. **Hardware Platform**: The foundational layer representing the physical PYNQ-Z2 board, configured through the bootable binary file.
4. **DPU Driver**: Software that enables communication between higher-level applications and the DPU hardware accelerator.
5. **Libraries (Lib)**: Software libraries that provide APIs for neural network operations, optimized for the DPU.
6. **API Layer**: Application Programming Interface that allows user applications to access DPU functionality through standard function calls.
7. **User Application**: The high-level software implementing the YOLO object detection algorithm, using the DPU for acceleration.
8. **Third Party**: Additional software components or tools that may interact with the system.
9. **DNNDK**: Deep Neural Network Development Kit, Xilinx's toolkit for optimizing neural networks for their DPU accelerators.
10. **Petalinux**: The Linux distribution customized for Xilinx platforms, providing the operating system environment.

The bidirectional arrows indicate data flow and communication between components, showing how software layers interact with hardware elements.



This workflow consists of the following steps:

1. **Create the DPU on Vivado**:
    - Design and configure the DPU accelerator using Xilinx Vivado
    - Synthesize and implement the design
    - Generate the bitstream file for FPGA configuration
2. **Integrate the DPU Hardware on the board using Petalinux**:
    - Configure Petalinux for the PYNQ-Z2 platform
    - Integrate the DPU driver and necessary kernel modules
    - Create a hardware platform that incorporates the DPU
3. **Quantize, optimize and compile the YOLO model using DNNDK**:
    - Convert the floating-point YOLO model to fixed-point representation
    - Optimize the network for the DPU architecture
    - Compile the model into a format executable on the DPU
4. **Generate Binary file with the Hardware description**:
    - Create a binary description of the hardware configuration
    - Package the compiled neural network with hardware information
5. **Create SD card image with Operating System**:
    - Combine Petalinux operating system with DPU drivers
    - Include compiled YOLO model and application software
    - Generate a bootable SD card image
6. **Deploy YOLO on PYNQ-Z2**:
    - Write the SD card image to physical media
    - Boot the PYNQ-Z2 from the SD card
    - Execute the YOLO application

1.5. Significance of Real-Time Object Detection

1.5.1. The Need for Real-Time Processing

Real-time object detection represents a critical capability in modern computer vision systems where timely information extraction can directly impact system effectiveness. Unlike offline processing, real-time detection requires:

- Processing speeds sufficient to analyze video frames as they arrive
- Low-latency response for time-critical applications
- Continuous operation under varying conditions
- Resource efficiency for deployment on embedded platforms

For the YOLO on PYNQ-Z2 implementation, "real-time" is defined as processing at least 5 frames per second with acceptable accuracy, sufficient for many practical applications while realistic for the platform's constraints.

1.5.2. Applications Enabled by FPGA-Based Detection

The implementation of YOLO on the PYNQ-Z2 platform opens possibilities for numerous practical applications:

1. **Smart Surveillance**:
    - Intrusion detection in restricted areas
    - Anomaly detection in public spaces
    - Traffic monitoring and analysis
    - Crowd behavior assessment
2. **Industrial Automation**:
    - Quality control on production lines
    - Safety monitoring in workspaces
    - Tool and part identification
    - Process verification
3. **Robotics**:

    - Obstacle avoidance
    - Object manipulation
    - Navigation in dynamic environments
    - Human-robot interaction
4. **Smart Infrastructure**:
    - Parking management systems
    - Building occupancy analysis
    - Energy optimization based on usage patterns
    - Safety compliance monitoring
5. **Assistive Technologies**:
    - Navigation assistance for the visually impaired
    - Smart home systems for elder care
    - Activity monitoring for healthcare

1.5.3. FPGA Advantages for Edge-Based Detection

The PYNQ-Z2 FPGA platform offers distinct advantages for real-time object detection at the edge:

1. **Low and Deterministic Latency**: Unlike CPUs or GPUs that may experience variable processing times due to operating system interference, FPGAs provide consistent, predictable performance critical for real-time systems.
2. **Energy Efficiency**: For battery-powered or energy-constrained applications, the FPGA implementation offers significantly better performance-per-watt compared to general-purpose processors.
3. **Parallel Processing**: The inherent parallelism of FPGA fabric allows simultaneous computation of different neural network layers, enabling higher throughput.
4. **Customizability**: The hardware can be optimized specifically for YOLO's computational patterns rather than using a one-size-fits-all processor architecture.
5. **Security and Privacy**: On-device processing eliminates the need to transmit potentially

sensitive video data to cloud servers, addressing privacy concerns.

1.5.4. Challenges and Considerations

Implementing real-time object detection on embedded FPGA platforms like PYNQ-Z2 presents several challenges:

1. **Resource Constraints**: Balancing model complexity against available FPGA resources requires careful optimization.
2. **Accuracy vs. Speed Tradeoff**: Real-time performance often necessitates compromises in detection accuracy, requiring application-specific tuning.
3. **Environmental Adaptation**: Real-world deployment faces challenges from varying lighting conditions, camera movements, and environmental factors.
4. **Power Management**: While more efficient than general-purpose processors, power consumption remains a consideration for portable applications.
5. **Development Complexity**: The hardware-software co-design approach introduces additional complexity compared to software-only implementations.

1.5.5. Impact on Edge Computing Paradigm

This project exemplifies the broader trend toward edge computing for AI applications, demonstrating how sophisticated deep learning models can function effectively without cloud connectivity. Key benefits include:

1. **Reduced Bandwidth Requirements**: Only detection results rather than raw video need to be transmitted.
2. **Lower Communication Latency**: Eliminating round-trip communication to cloud servers significantly reduces response time.
3. **Operation in Disconnected Environments**: Applications can function in areas with limited or intermittent connectivity.
4. **Scalability**: Distributed processing across multiple edge devices avoids central server

bottlenecks.

The YOLO on PYNQ-Z2 implementation serves as a proof-of-concept for deploying state-of-the-art computer vision algorithms on accessible hardware platforms, potentially democratizing access to advanced AI capabilities. By enabling real-time object detection in resource-constrained environments, this approach bridges the gap between laboratory AI research and practical field deployments across diverse domains.

With the introduction complete, subsequent chapters will delve into the technical details of the YOLO algorithm, its adaptation for the PYNQ-Z2 platform, hardware-software co-design strategies, and performance evaluation metrics.

1.6 Literature Review of Edge Computing for Object Detection

Redmon et al. (2016) introduced YOLO (You Only Look Once), a revolutionary approach to object detection that framed the problem as a single regression task. Unlike previous region proposal-based methods, YOLO divided images into a grid and simultaneously predicted

bounding boxes and class probabilities for each cell. This unified approach resulted in significantly faster detection speeds, achieving up to 45 FPS on high-end GPUs. The authors demonstrated how their architecture sacrificed some localization accuracy compared to two-stage detectors but gained substantial speed advantages, making it particularly suitable for real-time applications. This seminal work established the foundation for subsequent research on deploying object detection systems on edge devices, as its unified architecture was more amenable to optimization than more complex two-stage detectors.

Huang et al. (2017) conducted a comprehensive comparative study of various object detection systems, focusing on the speed/accuracy trade-offs. Their paper, "Speed/accuracy trade-offs for modern convolutional object detectors," evaluated several architectures including Faster R-CNN, R-FCN, and SSD with various feature extractors. The research provided crucial benchmarks that informed subsequent edge deployments by identifying which model families offered the best balance for resource-constrained environments. The authors found that while Faster R-CNN with ResNet or Inception architectures achieved the highest accuracy, SSD models with MobileNet provided significantly better efficiency with reasonable accuracy drops, making them more suitable for edge deployment. This paper established important baseline metrics that researchers use when optimizing models for platforms like the PYNQ-Z2 or Movidius NCS.

Howard et al. (2017) introduced MobileNets, a family of lightweight convolutional neural networks specifically designed for mobile and embedded vision applications. Their paper, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," detailed the use of depthwise separable convolutions to dramatically reduce computational costs and model size while maintaining reasonable accuracy. The authors provided a comprehensive exploration of the accuracy/latency trade-off space through width multipliers and resolution multipliers, allowing developers to select the appropriate configuration for their specific hardware constraints. MobileNets became a foundation for many edge computing implementations of object detection, including those on FPGA platforms like the PYNQ-Z2, as they reduced computational requirements while maintaining acceptable detection performance.

Blott et al. (2018) published "FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks," which focused specifically on FPGA implementations of deep neural networks. Their research demonstrated methods for creating highly optimized dataflow architectures for neural network inference on FPGAs, achieving significant throughput improvements compared to CPU implementations. The authors detailed quantization strategies, hardware-aware neural architecture design, and specialized compilation techniques that transformed neural networks into optimized FPGA implementations. Their framework showed particular promise for platforms like the PYNQ-Z2, as it enabled efficient

hardware-software co-design tailored to specific model architectures. This work provided key insights into the optimization strategies required for implementing detection models like YOLO on FPGA platforms.

Wang et al. (2019) published "Design and Implementation of Real-time Object Detection System Based on PYNQ," which directly addressed FPGA-based acceleration of object detection models. Their research implemented a simplified detection network on the PYNQ platform, demonstrating the viability of the FPGA-based approach for real-time applications. The authors detailed their hardware-software partitioning strategy, allocating computationally intensive convolutional layers to custom hardware accelerators while handling other operations in software. Their implementation achieved significant speedups compared to CPU-only execution while maintaining lower power consumption than GPU alternatives. This work provided direct comparison points to Andre Araujo's YOLO on PYNQ-Z2 implementation, showing how FPGA platforms could efficiently accelerate object detection workloads.

Véstias et al. (2020) published "Moving Convolutional Neural Networks to Embedded Systems: The AlexNet and VGG-16 Case," which focused on optimizing CNN implementations for FPGA platforms including the PYNQ-Z2. The paper provided a detailed exploration of hardware optimizations including loop unrolling, pipelining, and on-chip memory management strategies specific to FPGA architectures. The authors demonstrated how careful consideration of the memory hierarchy and computational parallelism could yield significant performance improvements even on modest FPGA platforms. Their implementation achieved efficient throughput for standard CNN models, providing valuable insights for object detection deployment. This work complemented Andre Araujo's implementation by detailing optimization strategies applicable to the convolutional backbone networks used in detection systems like YOLO.

1.7 Research Gaps That Could Be Addressed by the YOLO on PYNQ-Z2 Implementation

Hardware-Specific Optimization Techniques

While papers like Véstias et al. (2020) and Blott et al. (2018) discuss general FPGA optimizations, there's limited research on optimization techniques specifically tailored for the Zynq architecture used in the PYNQ-Z2. Your implementation could fill this gap by demonstrating YOLO-specific optimizations that leverage the unique heterogeneous architecture of the Zynq platform, which combines ARM processors with programmable logic.

Scalability Analysis for Low-Cost FPGAs

Most FPGA-based object detection research focuses on high-end FPGA platforms. There's insufficient research on how detection models can scale down to more affordable platforms like the PYNQ-Z2. Your work could address this gap by providing detailed analysis of scaling techniques and performance trade-offs on budget-constrained hardware, making FPGA acceleration more accessible.

Hardware-Software Co-Design Methodology

Wang et al. (2019) touched on hardware-software partitioning, but there remains a lack of comprehensive methodologies for determining optimal partitioning specifically for YOLO-based detectors. Your implementation could fill this gap by presenting a systematic approach to partitioning YOLO's computational graph between programmable logic and processing system components.

Educational Framework for Learning FPGA-Based AI

The existing literature lacks accessible frameworks for AI researchers to learn FPGA implementation. The PYNQ platform's Python-based interface offers unique educational advantages, and your model could address this gap by providing a well-documented pathway for ML practitioners to understand hardware acceleration without requiring extensive hardware description language knowledge.

Quantitative Power-Performance Analysis

While multiple papers address either performance or power consumption, comprehensive quantitative analysis comparing CPU, GPU, dedicated accelerators (like Movidius), and FPGA implementations of YOLO variants is limited. Your implementation could fill this gap by providing detailed benchmarks across these platforms with consistent metrics and test conditions.

Real-Time Adaptation Strategies

There's limited research on how FPGA-based detectors can adapt to changing conditions in real-time. Your model could explore dynamic reconfiguration capabilities of the PYNQ-Z2 to adjust precision, model complexity, or processing pipeline based on runtime requirements—an area not well explored in the literature.

Practical Implementation Guidelines

While theoretical papers abound, there's a scarcity of practical, step-by-step implementation guides for deploying state-of-the-art detectors on accessible FPGA platforms. Your work on the PYNQ-Z2 could fill this gap by providing concrete implementation details, addressing real-world challenges not covered in idealized research papers.

2. Requirements

2.1. Hardware Requirements

The physical components required for this project are as follows:

- **PYNQ-Z2 Board**: The central component for implementing the YOLOv3 object detector.
- **Ethernet Cable (RJ45)**: Used for network connectivity.
- **Ethernet to USB Adapter**: Necessary if your computer lacks an Ethernet port.
- **Micro-USB Cable**: For power supply and data transfer.
- **External Power Regulator (7 to 15 VDC, 12V recommended)**: Provides stable power to the PYNQ-Z2 board.
- **Micro SD Card (Minimum 8GB)**: Stores the operating system and project files.
- **USB Flash Drive (Pen Drive)**: For transferring files to and from the PYNQ-Z2 board

Estimated Cost:

Component	Specification	Approx. Cost (INR)
PYNQ-Z2 Board	Xilinx Zynq-7000 SoC (XC7Z020), 512MB DDR3 RAM, HDMI in/out, GPIO headers	₹30,000 – ₹40,000

Ethernet Cable (RJ45)	CAT5e or CAT6, 1m	₹100 – ₹300
Ethernet to USB Adapter	USB 3.0 compatible, for systems without an RJ45 port	₹600 – ₹1,200
Micro-USB Cable	USB-A to Micro-USB, for power and UART	₹100 – ₹250
Power Adapter	12V DC, 2A, center-positive, 2.1mm plug	₹500 – ₹1,200
Micro SD Card	Minimum 16GB Class 10 or UHS-I	₹350 – ₹700
USB Flash Drive (Pen Drive)	8GB or more, for file transfer and backups	₹300 – ₹600
Multimeter (Optional)	To verify power polarity	₹500 – ₹1,000

2.2. Software Requirements

To set up and run the YOLOv3 model on the PYNQ-Z2 board, the following software components are necessary:

- **PYNQ Image:** An operating system image tailored for the PYNQ-Z2 board.
- **WinRAR:** For decompressing downloaded files.
- **Etcher or Win32DiskImager**: Tools used to flash the PYNQ image onto the micro SD card.
- **Anaconda (2020.02):** A distribution of Python and R for scientific computing.
- **Python 3.9**: The programming language version compatible with the project.
- **DNNDK v3.1:** Xilinx's Deep Neural Network Development Kit for model optimization.
- **OpenCV (Version 4.5.5.62)**: An open-source computer vision library.
- **PyQt5 (Version 5.12)**: A set of Python bindings for the Qt application framework
- **VirtualBox** – A virtualization tool used to run an Ubuntu VM (where needed for model optimization).
- **PuTTY** – SSH client for direct terminal communication with the PYNQ board.
- **MobaXterm** – Can also be used for SSH and remote file transfers.

- **AOMEI Partition Assistant** – Helps in partitioning SD cards or managing external storage.
- **MiniTool Partition Wizard** – Alternative to Windows Disk Manager for resizing internal partitions.

To develop applications on the PYNQ-Z2 or access specific functionalities, you need to access various Xilinx tools. Let's introduce some of them.

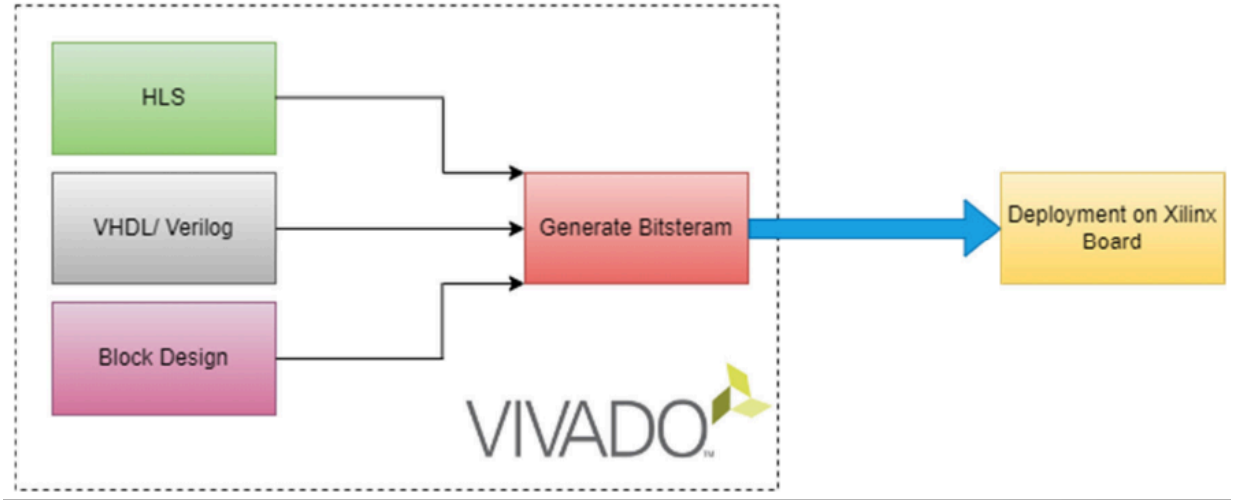
Vivado

Vivado is a suite containing various tools that allow to develop Hardware to be used on the PL of Xilinx FPGA's.

You can essentially create hardware in 3 ways:

- ➔ **High-Level Synthesis (HLS)** - Vivado HLS is a tool that allows to create hardware using high level programming languages like C, C++ or OpenCL. This allows the developer to express desired functionalities at a more abstract level, making the hardware development process easier and more accessible.
- ➔ **Hardware Description Languages (HDL)** - The traditional approach on hardware development consists of writing code in Hardware description languages like VHDL or Verilog. Vivado offers a Integrated Development Environment (IDE), to write, test, verify and debug these projects.
- ➔ **Block Design** - For complex projects, Vivado offers a system design tool based on graphics that allows to connect functional blocks pre-projected and customize the behavior of the system. These blocks are called Intellectual Property (IP) and were designed by Xilinx and

other creators.



Vivado also allows the user to perform simulation and validation on the Hardware before it's physical implementation. After the creation and development of the desired functions in Hardware, Vivado is able to generate a Bitstream, which is a binary file that has the information to configure the FPGA with the implemented logic.

Petalinux

Petalinux is an open-source development platform from Xilinx used to create, customize and implement Linux OS (Operational Systems) on Xilinx devices. So, it's possible to configure libraries and packages on the board's OS, define the Kernel, I/O's, drivers, etc. In addition to Software, this is the platform that links the Hardware created on Vivado to the board. After all configurations and after defining the binary file relative to de desired Hardware, we are able to copy specific files to a micro SD card and later start the board with it.

It's also important to mention that Petalinux is a Linux distribution and is normally used on a Virtual Machine so it's possible to make tests and configurations without worrying about damaging the native Operating System. I will guide you trough that process on the specific

chapter.

DNNDK

We have DNNDK which stands for Deep Neural Network Development Kit. This is a set of tools developed by Xilinx to accelerate the development and implementation of Deep Neural Networks (DNN's) on Xilinx FPGA devices. This development kit was specially designed to take advantage of the power of parallel processing and reconfigurable flexibility of the FPGA's to accelerate machine learning tasks and DNN inference.

This tool will be essential on the development of a object detector because it will be responsible to compress the YOLO model to fit the PYNQ-Z2 requirements. Also, the resulting file will communicate directly with the DPU to accelerate the Neural Network and make a very quick inference.

3. Project Overview

3.1. Understanding the Deep Processing Unit

The Deep Processing Unit (DPU) is a specialized hardware accelerator designed by Xilinx specifically for neural network inference. For our YOLO implementation on PYNQ-Z2, understanding the DPU architecture and capabilities is essential as it forms the foundation of our hardware acceleration strategy.

3.1.1 DPU Architecture Overview

The DPU is a configurable computation engine that accelerates convolutional neural networks (CNNs) by implementing dedicated hardware for operations such as convolution, pooling, and activation functions. Key characteristics include:

- **Scalable Architecture**: The DPU can be configured with different numbers of processing elements based on available FPGA resources
- **Instruction Set Architecture**: Uses specialized instructions optimized for CNN operations
- **Memory Subsystem**: Includes data buffers and memory interfaces optimized for CNN dataflow
- **Quantization Support**: Operates on reduced precision (typically INT8) to maximize computational efficiency

The DPU used in this project is designed to efficiently execute the YOLO algorithm's computational patterns, particularly the convolution operations that dominate the network's computational requirements.

3.1.2. DPU Components

The DPU consists of several fundamental components:

1. **Convolution Engine**: Hardware modules that perform the multiply-accumulate operations central to convolutional layers
2. **Control Unit**: Manages the execution flow and scheduling of operations
3. **Memory Interface**: Handles data movement between external DDR memory and internal buffers
4. **On-chip Memory**: Buffers that store intermediate results to minimize external memory access
5. **Activation Units**: Implement non-linear functions like ReLU

3.1.3. DPU and YOLO Integration

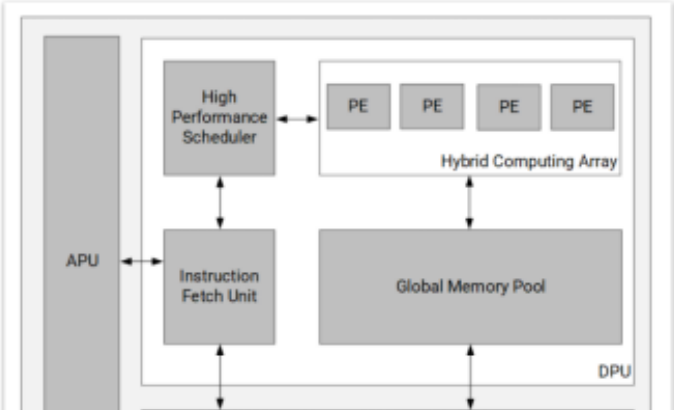
For the YOLO algorithm, the DPU is particularly valuable as it accelerates:

- Multiple convolutional layers that form the backbone of the network
- Feature extraction operations across different scales
- Processing of high-dimensional feature maps

The DPU effectively handles the majority of YOLO's computational workload, while the ARM processor handles control flow, preprocessing, and post-processing operations.

3.2. Creating the DPU 3.0 Architecture

It includes a set of highly optimized instructions, and supports most convolutional neural networks, such as VGG, ResNet, GoogLeNet, YOLO, SSD, MobileNet, FPN, and others. The DPU requires instructions to implement a neural network and accessible memory locations for input images as well as temporary and output data. A program running on the application processing unit (APU) is also required to service interrupts and coordinate data transfers.



The DPU has the following features:

• One AXI slave interface for accessing configuration and status registers.

• One AXI master interface for accessing instructions.

• Supports configurable AXI master interface with 64 or 128 bits for accessing data depending on the target device.

• Some highlights of DPU functionality include:

• Configurable hardware architecture core includes: B512, B800, B1024, B1152, B1600, B2304, B3136, and B4096

• Maximum of three homogeneous cores

• Convolution and deconvolution

• Depthwise convolution

• Max pooling

• Average pooling

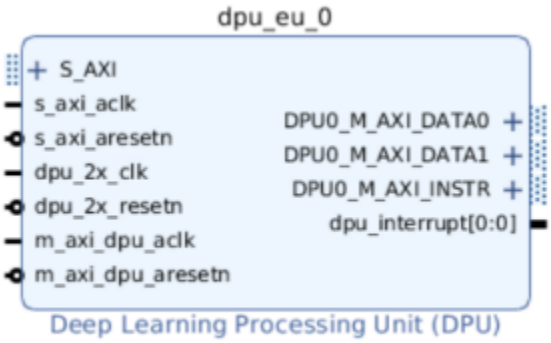
• ReLU, ReLU6, and Leaky ReLU

• Concat

• Element Wise-Sum

• Dilation

• Reorg



DPU IO ports

- Fully connected layer
- Softmax (Not Support at Zynq-7000)
- Batch Normalization
- Split

Table 1: DPU Signal Description

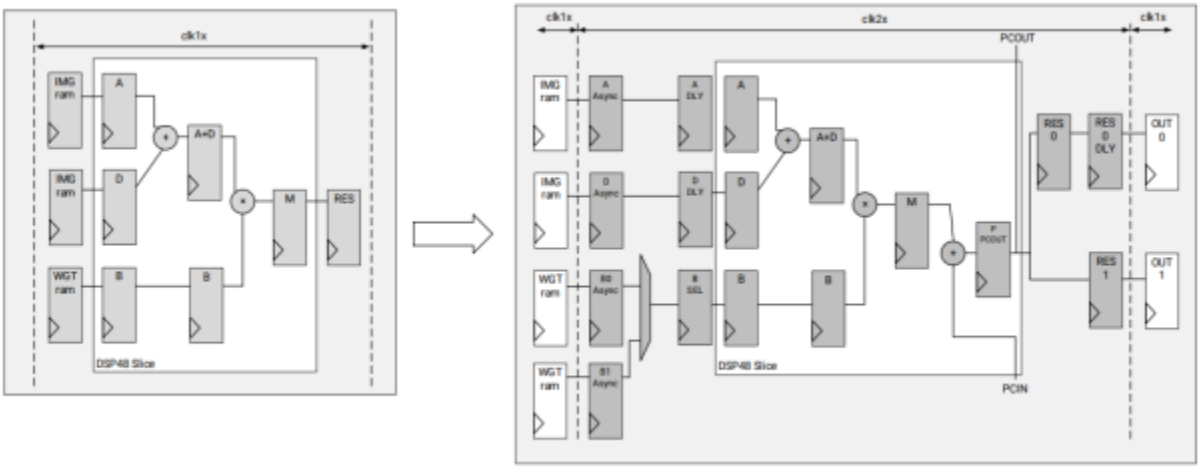
Signal Name	Interface Type	Width	I/O	Description
S_AXI	Memory mapped AXI slave interface	32	I/O	32-bit memory mapped AXI interface for registers.
s_axi_aclk	Clock	1	I	AXI clock input for S_AXI
s_axi_aresetn	Reset	1	I	Active-Low reset for S_AXI
dpu_2x_clk	Clock	1	I	Input clock used for DSP blocks in the DPU. The frequency is twice that of m_axi_dpu_aclk.
dpu_2x_resetn	Reset	1	I	Active-Low reset for DSP blocks
m_axi_dpu_aclk	Clock	1	I	Input clock used for DPU general logic.
m_axi_dpu_aresetn	Reset	1	I	Active-Low reset for DPU general logic
DPUx_M_AXI_INSTR	Memory mapped AXI master interface	32	I/O	32-bit memory mapped AXI interface for DPU instructions.
DPUx_M_AXI_DATA0	Memory mapped AXI master interface	128	I/O	128-bit memory mapped AXI interface for DPU data.
DPUx_M_AXI_DATA1	Memory mapped AXI master interface	128	I/O	128-bit memory mapped AXI interface for DPU data.
dpu_interrupt	Interrupt	1~3	O	Active-High interrupt output from DPU. The data width is determined by the number of DPU cores.
SFM_M_AXI (optional)	Memory mapped AXI master interface	128	I/O	128-bit memory mapped AXI interface for softmax data.
sfm_interrupt (optional)	Interrupt	1	O	Active-High interrupt output from softmax module.
dpu_2x_clk_ce (optional)	Clock enable	1	O	Clock enable signal for controlling the input DPU 2x clock when DPU 2x clock gating is enabled.

Table 7: Deep Neural Network Features and Parameters Supported by the DPU

Features	Description	
Convolution	Kernel Sizes	W: 1–16 H: 1–16
	Strides	W: 1–4 H:1–4
	Padding_w	1: kernel_w - 1
	Padding_h	1: kernel_h - 1
	Input Size	Arbitrary
	Input Channel	1–256 * channel_parallel
	Output Channel	1–256 * channel_parallel
	Activation	ReLU, LeakyReLU, or ReLU6
	Dilation	dilation * input_channel <= 256 * channel_parallel && stride_w == 1 && stride_h == 1
Depthwise Convolution	Kernel Sizes	W: 1–16 H: 1–16
	Strides	W: 1–4 H:1–4
	Padding_w	1: kernel_w - 1
	Padding_h	1: kernel_h - 1
	Input Size	Arbitrary
	Input Channel	1–256 * channel_parallel
	Stride_w	stride_w * output_channel <= 256 * channel_parallel
	Stride_h	Arbitrary
	Padding_w	1: kernel_w - 1
	Padding_h	1: kernel_h - 1
	Input Size	Arbitrary
	Input Channel	1–256 * channel_parallel
	Output Channel	1–256 * channel_parallel
	Activation	ReLU or LeakyReLU
Max Pooling	Kernel Sizes	W: 1–8 H: 1–8
	Strides	W: 1–4 H:1–4
	Padding	W: 1–4 H: 1–4
Elementwise-sum	Input channel	1–256 * channel_parallel
	Input size	Arbitrary
Concat	Output channel	1–256 * channel_parallel
Reorg	Strides	stride * stride * input_channel <= 256 * channel_parallel

3.2.1. DPU with Enhanced Usage of DSP

• DSP Double Data Rate (DDR) technique is used to improve the performance achieved with the device.
• Therefore, two input clocks for the DPU are needed, one for general logic, and the other for DSP slices.



Difference between DPU without DSP DDR and DPU Enhanced Usage

3.2.2. DPU Convolution Architecture

Table 8: Parallelism for Different Convolution Architectures

Convolution Architecture	Pixel Parallelism (PP)	Input Channel Parallelism (ICP)	Output Channel Parallelism (OCP)	Peak Ops (operations/per clock)
B512	4	8	8	512
B800	4	10	10	800
B1024	8	8	8	1024
B1152	4	12	12	1150
B1600	8	10	10	1600
B2304	8	12	12	2304
B3136	8	14	14	3136
B4096	8	16	16	4096

In each clock cycle, the convolution array performs a multiplication and an accumulation, which are counted as two operations. Thus, the peak number of operations per cycle is equal to PP*ICP*OCP*2.

3.2.3 RAM Usage

Table 9: Number of BRAM36K Blocks in Different Architectures for Each DPU Core

DPU Architecture	Low RAM Usage	High RAM Usage
B512 (4x8x8)	73.5	89.5
B800 (4x10x10)	91.5	109.5
B1024 (8x8x8)	105.5	137.5
B1152 (4x12x12)	123	145
B1600 (8x10x10)	127.5	163.5
B2304 (8x12x12)	167	211
B3136 (8x14x14)	210	262
B4096 (8x16x16)	257	317.5

3.2.4. Channel Augmentation

Table 10: Extra LUTs of DPU with Channel Augmentation

DPU Architecture	Extra LUTs with Channel Augmentation
B1024 (8x8x8)	2670
B1152 (4x12x12)	2189
B4096 (8x16x16)	1550
B512 (4x8x8)	1475
B800 (4x10x10)	2796

• Channel augmentation is an optional feature for improving the efficiency of the DPU when handling input channels much lower than the available channel parallelism.

• When the number of input channels is larger than the channel parallelism, then enabling channel augmentation will not make a difference

• In summary, the channel augmentation can improve the total efficiency for most CNNs, but it will cost extra logic resources.

3.2.5. DSP Usage

Table 11: Resources for Different DSP Usage

High DSP Usage					Low DSP Usage				
Arch	LUT	Register	BRAM	DSP	Arch	LUT	Register	BRAM	DSP
B512	20055	28849	69.5	98	B512	21171	33572	69.5	66
B800	21490	34561	87	142	B800	22900	33752	87	102
B1024	24349	46241	101.5	194	B1024	26341	49823	101.5	130
B1152	23527	46906	117.5	194	B1152	25250	49588	117.5	146
B1600	26728	56267	123	282	B1600	29270	60739	123	202
B2304	39562	67481	161.5	386	B2304	32684	72850	161.5	290
B3136	32190	79867	203.5	506	B3136	35797	86132	203.5	394
B4096	37266	92630	249.5	642	B4096	41412	99791	249.5	514

• This allows you to select whether DSP48E slices will be used for accumulation in the DPU

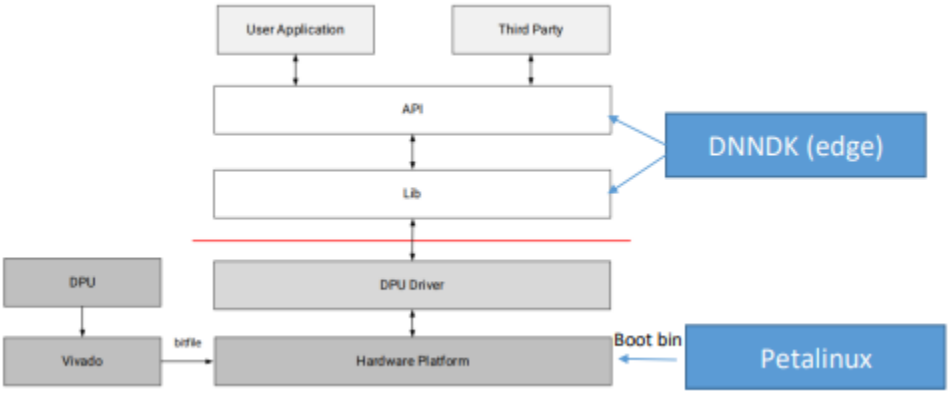
convolution module.

• In low DSP usage mode, the DPU IP will use DSP slices only for multiplication in the convolution.

• In high DSP usage mode, the DSP slice will be used for both multiplication and accumulation.

• Thus, the high DSP usage consumes more DSP slices and less LUTs

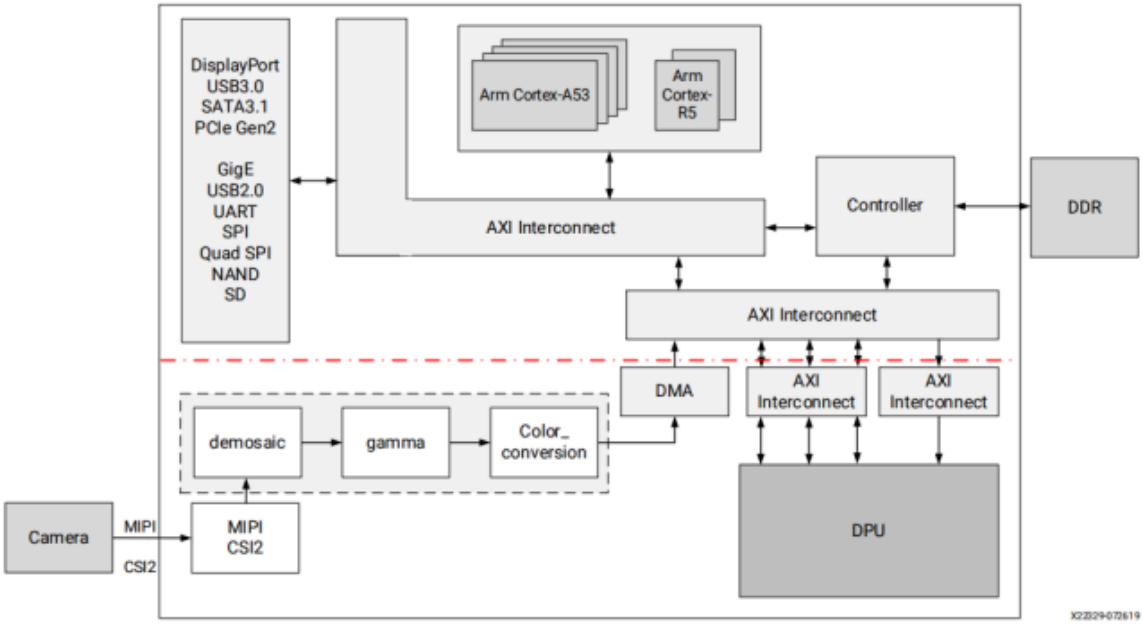
3.2.6 DPU Development Flow



• The DPU requires a device driver which is included in the Xilinx Deep Neural Network Development Kit (DNNDK) toolchain.

• Vivado to generate the bitstream. Then, download the bitstream to the target board and install the DPU driver.

3.2.7 Example System with DPU

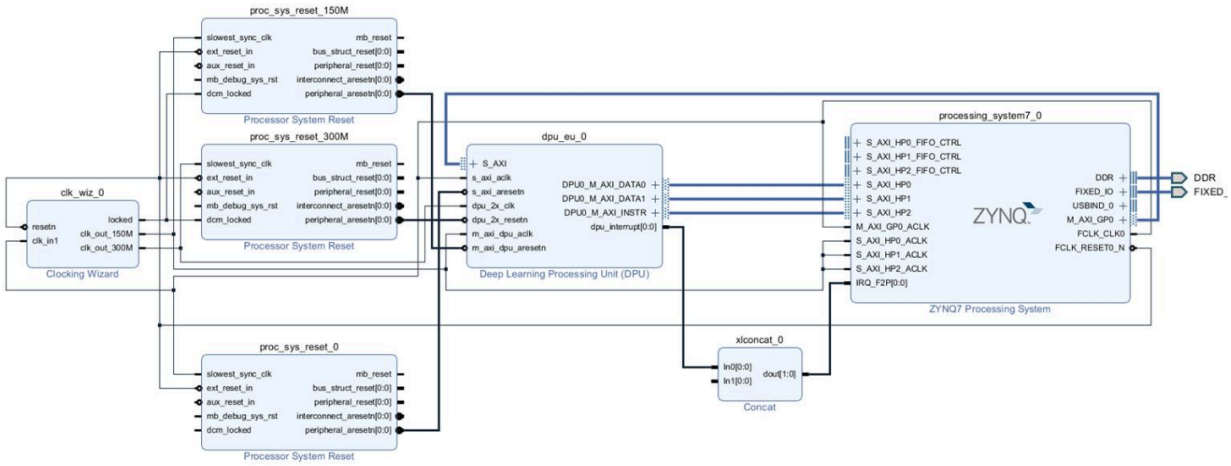


3.2.8. Final Layout

3.2.8.1. Prerequisites Installation

1. **Install Vivado 2020.1:**
    - Download and install Vivado Design Suite 2020.1
    - Ensure that the Zynq-7000 device family is selected during installation
    - Install required licenses for IP cores

2. **Install Vitis AI Library:**
    - Download Vitis AI 1.3 from Xilinx website
    - Follow installation instructions for your operating system
    - Verify DPU IP is available in the installation



3. **Set up PYNQ-Z2 Support:**
    - Download PYNQ-Z2 board files
    - Copy to `Vivado/2020.1/data/boards/board_files/` directory
    - Restart Vivado to recognize the board

3.2.8.2. Creating the Base Vivado Project

1. **Launch Vivado and Create New Project:**
    - Open Vivado 2020.1
    - Click "Create New Project"
    - Select "Next" on the welcome page
2. **Project Configuration:**
    - Specify project name (e.g., "dpu_pynq_z2")
    - Choose project location
    - Select "RTL Project" and check "Do not specify sources at this time"

    - Click "Next"
3. **Board Selection:**
    - In the "Default Part" window, select "Boards" tab
    - Filter or scroll to find "PYNQ-Z2"
    - Select the PYNQ-Z2 board and click "Next"
    - Review project summary and click "Finish"

3.2.8.3. Creating the Block Design

1. **Create Block Design:**
    - In the Flow Navigator, click "Create Block Design"
    - Name the design "system" and click "OK"
2. **Add Zynq Processing System:**
    - Right-click in the block design canvas and select "Add IP"
    - Search for "ZYNQ7" and double-click "ZYNQ7 Processing System"
3. **Configure Zynq PS:**
    - Double-click the Zynq PS block to open configuration
    - Click "Presets" and apply "PYNQ-Z2" preset
    - In the PS-PL Configuration section:
        - Enable HP0 interface (S AXI HP0 Interface)
        - Enable General Purpose AXI Master Port (M AXI GP0)
    - **In Clock Configuration:**
        - Set PL Fabric Clocks > PL CLK 0 to 100 MHz
        - Enable FCLK_CLK0 and set to 100 MHz
    - Click "OK" to apply settings
4. **Run Block Automation:**
    - Click "Run Block Automation" in the green banner
    - Accept default options and click "OK"

3.2.8.4. Adding and Configuring the DPU IP

1. **Add DPU IP Core:**
    - Right-click on the canvas and select "Add IP"
    - Search for "DPU" and double-click "DPU" from Vitis AI Library
2. **Configure the DPU IP:**
    - Double-click the DPU block to open configuration
    - Set Architecture Parameters:
        - Architecture: B4096
        - Clock: 1.4× (if timing allows) or 1.0× (safe)
        - RAM Usage: High
        - DSP Usage: High
        - AXI Data Width: 128 bits
        - DPU Convolution: Enabled
        - Depth-wise Convolution: Enabled
        - Average Pooling: Enabled
        - ReLU Type: ReLU + LeakyReLU
        - Channel Augmentation: Enabled
    - Set AXI Interface Parameters:
        - S_AXI_CLOCK_AUTO_FREQ: Enabled
        - M_AXI_CACHE_VALUE: 2
    - Set Advanced Parameters:
        - Architecture Version: Compatible
        - MISC_CTRL: Disable softmax
    - Click "OK" to apply settings
3. **Add Clock Wizard:**
    - Add IP: "Clocking Wizard"
    - Configure:
        - Input frequency: 100 MHz from Zynq PS

        - Output clocks: 200 MHz for DPU core clock
    - Click "OK"
4. **Add Processor System Reset Blocks:**
    - Add IP: "Processor System Reset"
    - Add two instances: one for each clock domain

3.2.8.5. System Integration

1. **Connect Clock and Reset:**
    - Connect clk_wiz_0 input to Zynq PS FCLK_CLK0
    - Connect PS reset to rst_clk_wiz_0 ext_reset_in
    - Connect clk_wiz_0 outputs to DPU clock inputs and reset blocks
    - Connect reset blocks' peripheral_aresetn to corresponding DPU reset inputs
2. **Connect AXI Interfaces:**
    - Add "AXI Interconnect" IP for connecting PS and DPU
    - Configure AXI Interconnect:
        - One slave interface
        - One master interface
    - Connect:
        - S00_AXI to Zynq PS M_AXI_GP0
        - M00_AXI to DPU S_AXI
3. **Connect Memory Interfaces:**
    - Add "AXI SmartConnect" for high-performance memory access
    - Configure for one slave and one master port
    - Connect:
        - S00_AXI to DPU M_AXI
        - M00_AXI to Zynq PS S_AXI_HP0
4. **Run Connection Automation:**
    - Click "Run Connection Automation" in the green banner

    - Select all interfaces and click "OK"
    - Verify all clock, reset and AXI connections
5. **Validate Design:**
    - Click "Validate Design" button (✓) in the toolbar
    - Fix any reported errors or warnings
    - Ensure all interfaces are properly connected

3.2.8.6. Generate Output Products

1. **Create HDL Wrapper:**
    - Right-click on the block design file in Sources
    - Select "Create HDL Wrapper"
    - Choose "Let Vivado manage wrapper and auto-update"
    - Click "OK"
2. **Generate Block Design:**
    - Right-click on the block design
    - Select "Generate Output Products"
    - Use default settings and click "Generate"
    - Wait for completion (this may take several minutes)

3.2.8.7. Implementation and Bitstream Generation

1. **Synthesis:**
    - Click "Run Synthesis" in the Flow Navigator
    - Use default settings and click "OK"
    - Wait for synthesis to complete
2. **Implementation:**
    - Click "Run Implementation" after synthesis completes
    - Use default settings and click "OK"
    - Monitor for timing and resource utilization issues

3. **Bitstream Generation:**
    - Click "Generate Bitstream" after implementation
    - Use default settings and click "OK"
    - Wait for bitstream generation to complete
4. **Export Hardware:**
    - Click "File" > "Export" > "Export Hardware"
    - Select "Include bitstream"
    - Specify output location
    - Click "OK"

3.2.8.8. Post-Implementation Analysis

1. **Resource Utilization:**
    - Open the implemented design
    - View the "Report Utilization" summary
    - Verify that usage stays within PYNQ-Z2 resources:
        - DSP: Less than 220 (100%)
        - BRAM: Less than 280 (100%)
        - LUT: Less than 53,200 (100%)
        - FF: Less than 106,400 (100%)
2. **Timing Analysis:**
    - Check "Report Timing Summary"
    - Ensure all timing constraints are met (positive slack)
    - Address any timing violations if present

3.3 Integrating DPU into SD Card Image

After finishing the Vivado project, it's time to implement the designed Hardware (the DPU) on a SD card image. To do that we will use a Xilinx software called Petalinux. It allows to

create a SD card image with the Linux Operating System and the packages and libraries we want, and it also holds the DPU hardware information to be sent to the Zynq-7020 chip.

This image will be flashed on an actual Micro SD card and later to be inserted on the PYNQ-Z2. To use Petalinux we will resort to a Virtual Machine (VM) because it allows to make configurations without damaging the native OS. In other words, by using VM we can have a different OS running on your computer. In this case we have a Windows OS on the computer (That is called the Host machine) and we will need to run a Ubuntu OS on as a program on the host computer (this is the Guest Machine).

3.3.1 Virtual Machine Set up

For this project we are going to use the Virtual Machine Software VirtualBox.

To install VirtualBox follow this steps:

1. Go to VirtualBox website.
2. Click the link that says "Windows hosts".
3. After the download is complete, run the execution file.
4. Proceed with the installation.

Now we have the VirtualBox icon on the Desktop and we are able to create multiple virtual machines on your computer.

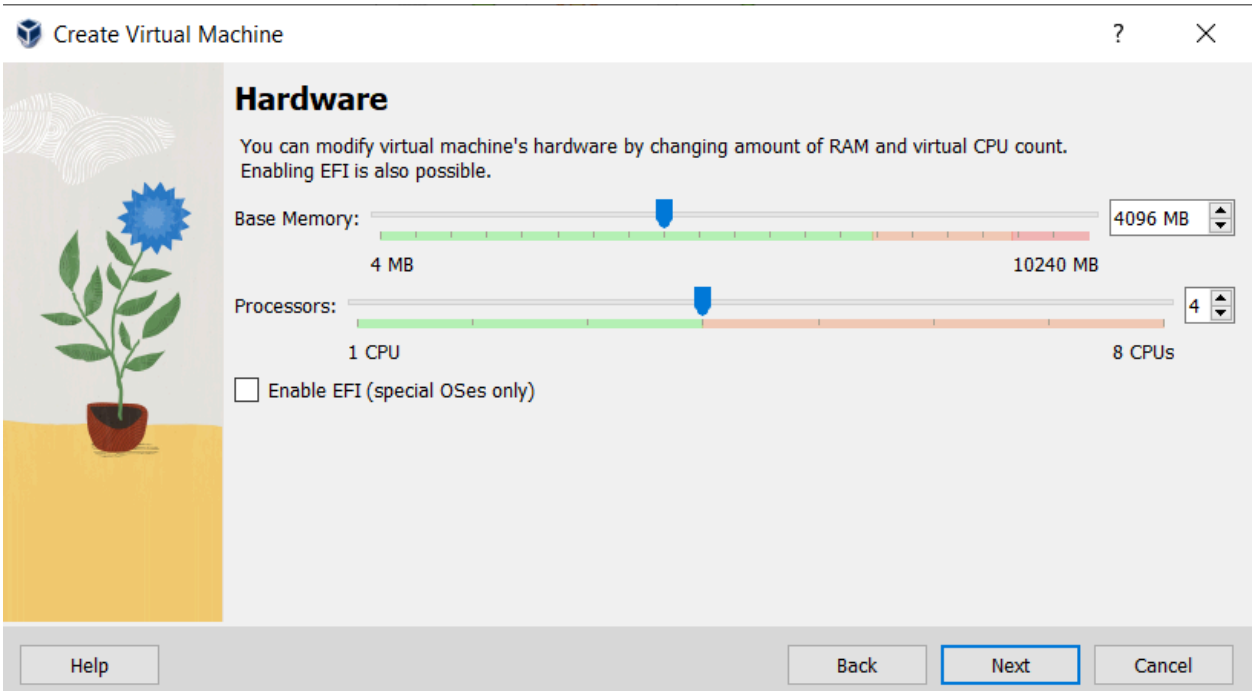
We now need to create the virtual machine running the Ubuntu OS. To do that, first we need to download the Ubuntu 18.04.2 disk image from the old releases website. You want to select the "ubuntu-18.04.2-desktop-amd64.iso".

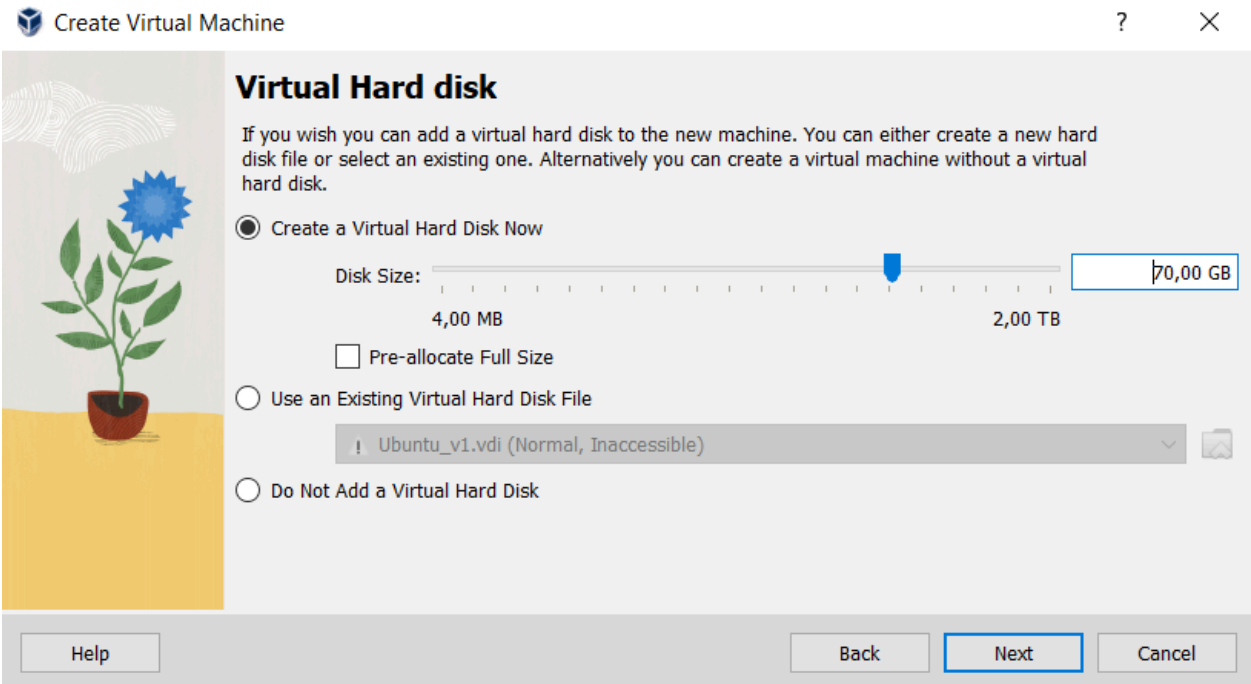
Next step, we are going to give a name for the Virtual Machine, for example, "Petalinux" because that's what we will be working with. Then we select the folder where the files will be

stored (you can leave the default option). Before proceeding, we have to select the .iso file we just downloaded so the system knows it is Ubuntu 18.04.Before proceeding, we have to select the .iso file we just downloaded so the system knows it is Ubuntu 18.04.

Next, we want to set the username and the password.Next we need to select the resources.

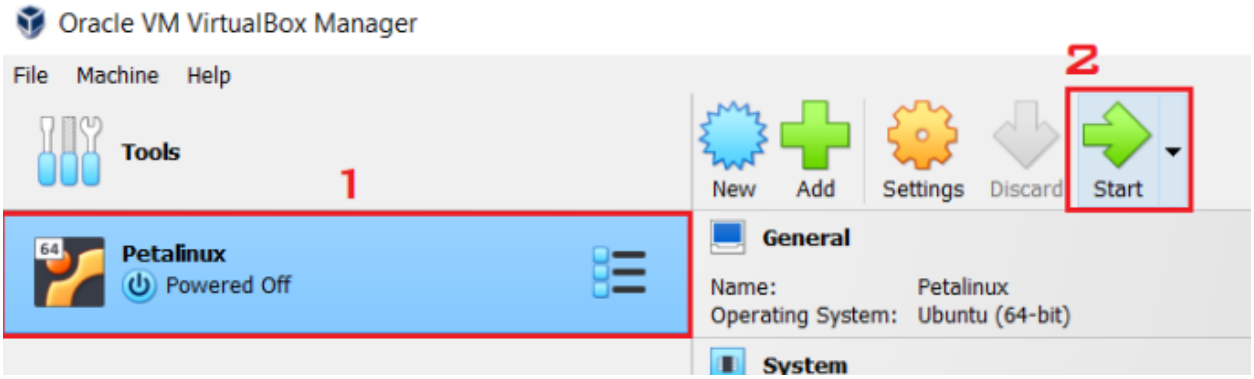
Next we need to allocate some disk space for the Ubuntu Desktop on the host PC. It should be fine to select 60GB but to be on the safe side we allocated 70GB.





Now the window shows the summary and you can check every setting. Then we only need to press "Finish" to effectively create the Virtual Machine.

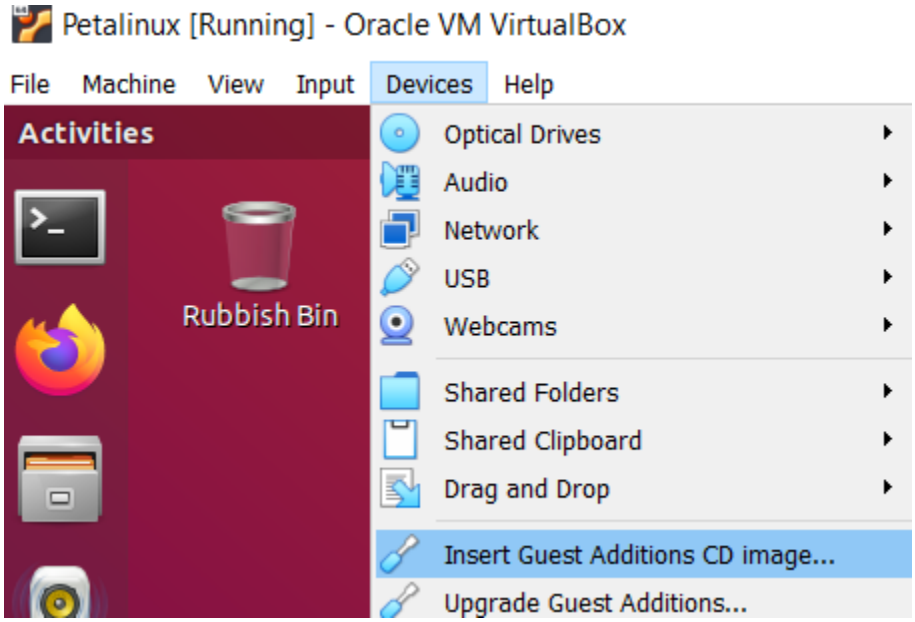
After installation, every time you need to use the Ubuntu VM you just open VirtualBox, select Petalinux and click on "Start".



3.3.2 Ubuntu Setup

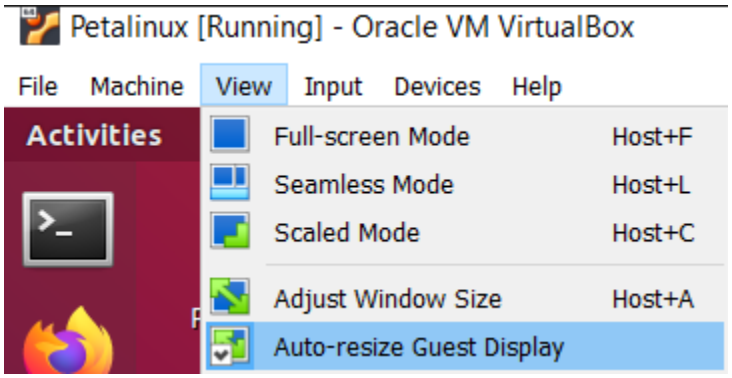
Ubuntu is now available on a virtual machine on your computer. But be aware when there is a warning recommending the OS update, refuse it.The Ubuntu version should not be changed. Click "Don't upgrade" and then "Cancel".

The next step is to install "Guest Additions" on the Operating System. This package is from VirtualBox and it enables some very important features like the ability to use the same clipboard and to maximize the window size. The clipboard will be shared by the Host OS (Windows) and the Guest OS (Ubuntu), enabling you to copy and paste text, images and files between them. To install guest additions open Ubuntu, log in and then click on "Devices" on the top bar, and after that click the screwdriver icon.



We need to authorize the program to run on the Terminal. It will install all the necessary stuff and after that restart the VM. After restarting the VM, enable the maximization of the Ubuntu

screen by clicking on "view" on the top bar and then "Auto-resize Guest Display".



On the Terminal, first access the root directory that contains some important files of the OS. To do that run the command *su root*.

It will ask for user password and then press ENTER. Then to access the "suddoers" file and open it: *nano /etc/sudoers*

The file will be open and now to move the cursor below "User privilege specification" and insert username as well as all the permissions. We can do it like this:

 # User privilege specification

root ALL=(ALL:ALL) ALL

<user> ALL=(ALL:ALL) ALL

After defining the user as admin, save the modifications to the file by following this steps:

1. CNTRL + O (To save content);
2. ENTER (To confirm save);
3. CNTRL + X (To leave editing);
4. Write "exit" on the terminal to leave the root directory and get back to normal;

3.3.3 Petalinux Installation

Before starting, download Petalinux 2019.2 from the website (Version compatible with the Vivado files).Download the file and meanwhile we will complete other requirements.

First to update the Operating System using these two commands:

sudo apt update

sudo apt upgrad

sudo apt install htop vim openssh-server iproute2 gcc g++ net-tools libncurses5-dev zlib1g:i386 libssl-dev flex bison libselinux1 xterm autoconf libtool texinfo zlib1g-dev gcc-multilib build-essential screen pax gawk python3 python3-pexpect python3-pip python3-git python3-jinja2 xz-utils debianutils iputils-ping libegl1-mesa libsdl1.2-dev pylint3 cpio chrpath socat python -y

pip3 install --upgrade pip

sudo mkdir -p /opt/petalinux/

sudo chmod 755 /opt/petalinux/

sudo chown -R <user> /opt/petalinux/

Next, we wait for the Petalinux download. After that

cd Downloads

sudo chmod 755 petalinux-v2019.2-final-installer.run

./petalinux-v2019.2-final-installer.run /opt/petalinux/

chsh -s /bin/bash

Now you RESTART the Virtual Machine. and follow the instructions below:

sudo dpkg-reconfigure dash

Now, every time to start Petalinux, do this command:

source /opt/petalinux/settings.sh

3.3.4. Petalinux Project

To start, we need to create a folder to store all the necessary project files. Create a folder on the Desktop directory with the "mkdir" command on the terminal or using the Ubuntu interface.

After that I need you to go to that newly created directory using the "cd" command.

cd Desktop/dpu_integration

source /opt/petalinux/settings.sh

Now we create the Petalinux project on which we will call "pynqz2_dpu" based on the Zynq template:

petalinux-create --type project --template zynq --name pynqz2_dpu

Next, we need to go to the Petalinux project directory. It's here where all the important stuff will be done.

cd pynqz2_dpu/

Now, I need you to copy and paste the Vivado project with all the files to the Ubuntu VM. As we installed Guest Additions earlier, you can do it directly. We will be needing the ".xsa" file. After you have the Vivado project on the VM (you can place it on the Desktop for convenience), we can now base the Petalinux project according to the DPU Hardware. On the final part, you have to place the path to the ".xsa" file.

petalinux-config --get-hw-description=/home/andre/Desktop/VivadoDPU/pynqz2_dpu

You will be presented on a nice little menu. Here is how you can work on this configuration interface:

- Here you can navigate the options using the up and down arrows;
- To change between "select", "exit", "save" and "load" you can use the left and right arrows;
- To select sub-menus you can press ENTER;
- To enable things you press "y" and to disable things you press "n";

Now I need you to access this settings and disable the option:

DTG Settings->Kernel Bootargs->disable generate boot args automatically

On "user set kernel bootargs", paste the following:

console=ttyPS0,115200 root=/dev/mmcblk0p2 rw earlyprintk quiet rootfstype=ext4 rootwait cma=256M

Next thing you need to change the image to EXT4 type because that is the Linux type. To do that you go to this menus and put "y" to enable EXT4:

Image Packaging Configuration->Root filesystem type (EXT4 (SD card))

After that, you exit the configuration and save changes. Then, we need to change the content of a file. To access it you use the command gedit:

gedit project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi

On this blank file I need you to paste the following code:

```
/include/ "system-conf.dtsi"
&amba {
```

```
xlnk {
compatible = "xlnx,xlnk-1.0";
};
};
&amba{
dpu{
#address-cells = <1>;
#size-cells = <1>;
compatible = "xilinx,dpu";
base-addr = <0x4f000000>; //CHANGE THIS ACCORDING TO YOUR DESIGN
dpucore {
compatible = "xilinx,dpucore";
interrupt-parent = <&intc>;
interrupts = <0 29 4>; //CHANGE THIS ACCORDING TO YOUR DESIGN
core-num = <0x1>; //CHANGE THIS ACCORDING TO YOUR DESIGN
};
};
};
//usb device tree
/{
usb_phy0: usb_phy@0 {
compatible = "ulpi-phy";
#phy-cells = <0>;
reg = <0xe0002000 0x1000>;
view-port = <0x0170>;
drv-vbus;
};
};
&usb0 {
dr_mode = "host";
usb-phy = <&usb_phy0>;
};
```

Now to download the following file, containing some important content to the DPU.

https://www.xilinx.com/member/forms/download/design-license-xef.html?filename=zcu102-dpu-trd-2019-1-190809.zipwww.xilinx.com

We need to copy some of the files from the downloaded package, specifically some stored on a ".bsp" file. To open that specific file we will need to download a program called "7-Zip"

After the installation is complete, you can close the "Ubuntu Software" window and you can go right click on the ".bsp" file again and click on "Open With Other Application" and now select

the newly installed application: P7Zip - Desktop.

You want to access the directory that contains the folder "recipes-modules" and to do that you go here: xilinx-dpu-trd-zcu102-v2019.1/zcu102-dpu-trd-2019-1/project-spec/meta-user/

Then we need to extract the "recipes-modules" file to the home directory (or other you desire). To do it you press on the file, and then on the top bar "extract". You will be presented with the following window where you can select the destination path to /home or the /Desktop.

We want to copy the file "recipes-modules" from the home directory to the "project-spec/meta-user/" directory on the Petalinux project. You can copy by hand using the interface or you can use the following command.

cp -rv ~/recipes-modules/ project-spec/meta-user/

Next step, you need to add *echo "CONFIG_dpu"* on the last line of the file on the following folder: project-spec/meta-user/conf/user-rootfsconfig.

Then you also need to add this line *echo 'IMAGE_INSTALL_append = "dpu"'* on the file project-spec/meta-user/conf/petalinuxbsp.conf.

We need to configure the Root filesystem now. This is where we will define the packages that will be on the SD card image OS from the start. This are some important packages that will be important to do some applications later. To do the configuration, you place the following command on the Terminal:

petalinux-config -c rootfs

Then is time to configure the Kernel. The only thing we need to do here is to activate the USB driver. To access the configuration you write on the terminal:

petalinux-config -c kernel

And then you enable:

Device Drivers -> USB support -> [*]USB announce new devices

To finish kernel configuration you write this to the Terminal:

petalinux-build -c kernel -x finish

Then you clear the text and paste this:

CONFIG_USB_OTG=y

 # CONFIG_USB_OTG_FSM is not set

 # CONFIG_USB_ZERO_HNPTEST is not set

CONFIG_MEDIA_USB_SUPPORT=y

CONFIG_USB_VIDEO_CLASS=y

CONFIG_USB_VIDEO_CLASS_INPUT_EVDEV=y

CONFIG_USB_GSPCA=m

CONFIG_V4L_PLATFORM_DRIVERS=y

CONFIG_VIDEO_ADV7604=y

CONFIG_USB_HID=y

CONFIG_USB_OHCI_LITTLE_ENDIAN=y

CONFIG_USB_SUPPORT=y

CONFIG_USB_COMMON=y

CONFIG_USB_ARCH_HAS_HCD=y

CONFIG_USB=y

CONFIG_USB_ANNOUNCE_NEW_DEVICES=y

CONFIG_USB_DEFAULT_PERSIST=y

CONFIG_USB_EHCI_HCD=y

CONFIG_USB_EHCI_ROOT_HUB_TT=y

CONFIG_USB_EHCI_PCI=y

CONFIG_USB_EHCI_HCD_PLATFORM=y

CONFIG_USB_ACM=m

CONFIG_USB_PRINTER=m

CONFIG_USB_WDM=m

CONFIG_USB_TMC=m

CONFIG_USB_STORAGE=y

CONFIG_USB_CHIPIDEA=y

CONFIG_USB_CHIPIDEA_OF=y

CONFIG_USB_CHIPIDEA_PCI=y

CONFIG_USB_CHIPIDEA_HOST=y

CONFIG_USB_PHY=y

CONFIG_NOP_USB_XCEIV=y

CONFIG_AM335X_CONTROL_USB=y

CONFIG_AM335X_PHY_USB=y

CONFIG_USB_GPIO_VBUS=y

CONFIG_USB_ULPI=y

CONFIG_USB_ULPI_VIEWPORT=y

All configurations are done and we will build the project. The process can take 30 minutes with my VM resources configuration but according to your computer specs it can take less. Just paste this command on the Terminal:

petalinux-build

After the build process is complete, we need to make the Petalinux software generate the files to be written on the SD card. The next command will do that job:

petalinux-package --boot --force --fsbl images/linux/zynq_fsbl.elf --fpga images/linux/.bit --u-boot*

You can then save a bsp file with all the configurations you done using the next command. This works like a commit, so any changes you do you can save on this files. You can name the file whatever you like but here I called it "pynq-z2-dpu.bsp".

petalinux-package --bsp -p ./ -o pynq-z2-dpu.bsp

3.3.5 SD Card Image

Now we will need to pass the files we created on Petalinux to the SD car but first we will need to configure it properly. For this, we will use a program called "AOMEI Partition Assistant" and it will run on Windows before moving again to the Ubuntu VM.

Now, you will need to insert the micro SD card on the adapter and the whole unit on you SD card reader.

Once you got that sorted out, we will open the newly installed application AOMEI Partition Assistant. There will be some disks, normally the computer disk where the Windows OS is stored and also, there should appear the SD card inserted on the SD card reader.

Now that you located your SD card on the program, you should right click on the disk and "Delete all partitions". After that you need to go on the top left corner and click "Apply" and confirm so it does the job.

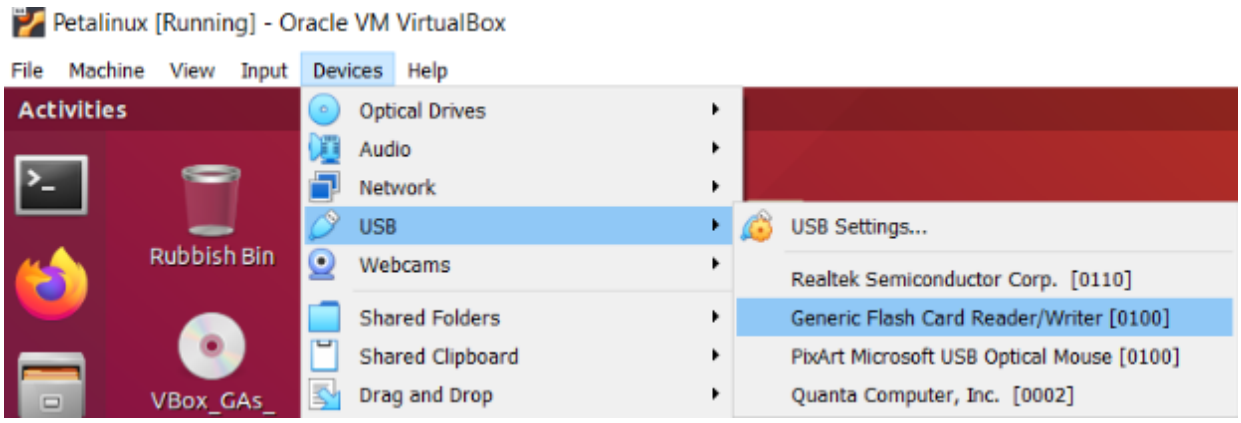
Then we need to create two partitions with different specifications:

- 128MB partition called "BOOT" with file system FAT32;

- Partition called "rootfs" with the rest of the storage space and with file system EXT4;

When the SD card is ready with two partitions (BOOT and rootfs), we can now go back to the Petalinux project.

To make it recognizable by the Ubuntu OS, you need to open the VM and on the top bar go to "devices", then "USB" and look for a Flash card reader.



After clicking the Flash card reader option, the micro SD card will be recognized by Ubuntu and will appear a little icon on the Desktop for each partition.

Generally the SD card will be here so for BOOT we can access through */media/<user>/BOOT/* and for rootfs */media/<user>/rootfs*.

Now, let's go back to the Petalinux project. We want to transfer the configuration files to the BOOT partition so the embedded OS will have the necessary information to start on the desired way. To pass those files you use these commands:

cp ./images/linux/BOOT.BIN /media/<user>/BOOT/

cp ./images/linux/image.ub /media/<user>/BOOT/

The Linux OS will be presented on the EXT4 format partition. So, we will move and uncompress the necessary folder to that partitions in a similar way to the BOOT partition:

sudo tar xzf ./images/linux/rootfs.tar.gz -C /media/<user>/rootfs/

sudo cp -r ../zynq7020_dnndk_v3.1/ /media/<user>/rootfs/home/root/*

To end the process you execute the command syncon the Terminal so you clear the system cache.

4. Methodology

In this phase of the project, the objective consists in reducing and compiling the YOLOv3 object detection model. To do that, we will be using a tool called DNNDK v3.1. This tool allows to quantize and optimize Deep Learning models and also compile, in other words, transform the quantized model in a binary file so it later can interact with the DPU. Without this reduction in size, the original YOLO model wouldn't be compatible with the PYNQ-Z2 because of it's restricted memory.

4.1 Converting YOLOv3 Model to TensorFlow Format

The first thing you should know is that the YOLOv3 model is made on a framework called Darknet.This will be addressed in this chapter, is precisely the conversion from Darknet to Tensorflow because DNNDK only supports the second one.

The first conversion will involve Darknet to Keras. Keras is a high level neural network API written in Python and that is directly related to Tensorflow, making the process easier. We will use the pre existing script to do that:

cd ~

cd Desktop/YOLO-on-PYNQ-Z2/DNNDK/model_conversion

Then to convert the YOLOv3 model we will need to use the script "convert.py" and indicate the YOLO configuration file as well as the weights.

As you can see below to get an idea, click on "weights" and initialize the download of the file. Then I need you to put that file called "yolov3.weights" on the "model_conversion" folder so

we can proceed. With everything set, you can write the next command to convert the yolo model from Darknet to Keras

After the conversion, there will be created a file with the ".h5" extension on a new "model_data" folder, meaning the YOLO is now on Darknet.

The Next step is to convert the Keras model to Tensorflow.We now need to indicate the path to the Keras model and the path to the resulting Tensorflow model.

python keras_to_tensorflow.py --input_model=/home/vidhi/Desktop/YOLO-on-PYNQ-Z2/DNNDK/model_conversion/model_data/yolo.h5 --output_model=/home/vidhi/Desktop/YOLO-on-PYNQ-Z2/DNNDK/model_conversion/model_data/yolo.pb

Now the resulting file will have the ".pb" extension and will be stored also on the "model_data" folder. This means that the yolo model is now in Tensorflow format

The last thing to do is to move or copy the Tensorflow model to the quantization folder so we can use it in the next section.

4.2. Model Quantization Techniques

Now comes the part where we will compress the YOLOv3 model. This compression is needed to fit the PYNQ-Z2 board.For the PYNQ-Z2 we will have to limit these values to 8 bit since that's the maximum width of the channel connecting the PL to the PS.

4.2.1 Install DNNDK Tools (on host machine)

Make sure DNNDK is installed (typically on an Ubuntu 16.04 machine):

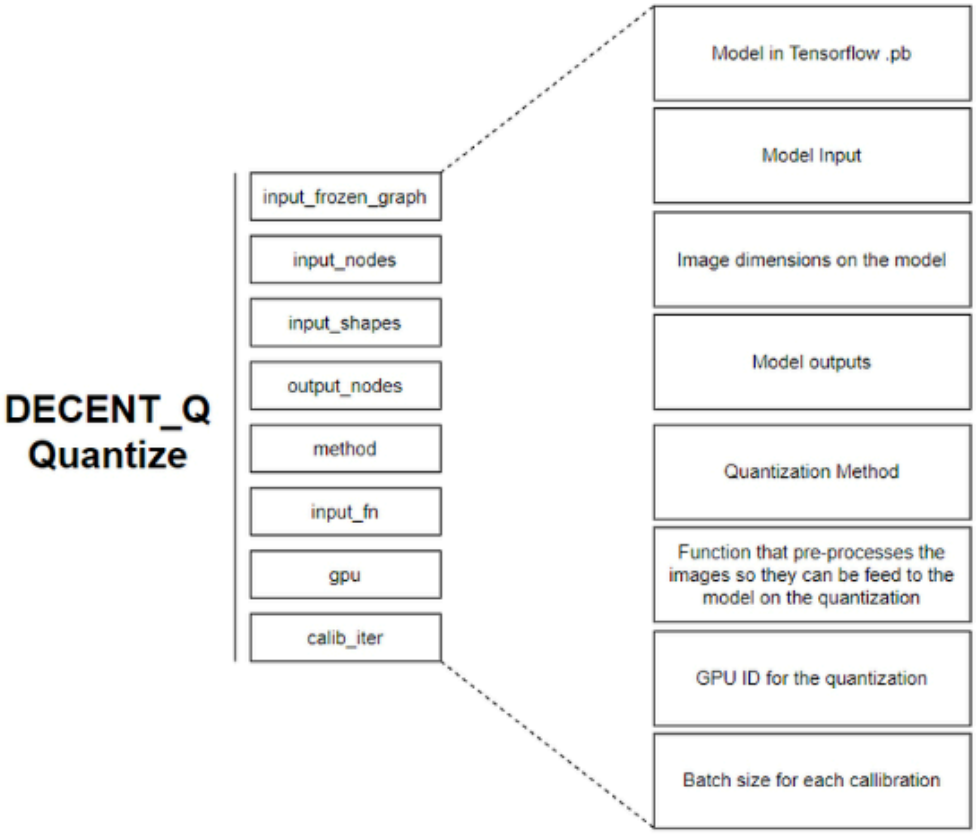
source /opt/vitis_ai/setup.sh

4.2.2 Prepare Input Images for Calibration

One of the most famous Datasets is the COCO (Common Objects in COntext). This dataset has 80 classes (80 types of possible objects) and annotations for more than 118 thousand images!

For this application this Dataset will be more than enough. The download of the COCO dataset can be done on the official website.

Store the images and annotations on a folder called "yolo_dataset". Inside this folder store the images on a folder "images" and the file called "instances_train2017.json" on a folder called "labels". The Dataset should be placed on the quantization folder of the repository.



The first parameter "input_frozen_graph" refers to the YOLOv3 model in Tensorflow. The next parameters are "input_nodes" and "output_nodes" which correspond to the Neural Network input and output names we talked about earlier. On the "input_shapes" we are just going to specify the size as 416x416 as it is very common, leaving the other values as default. The parameter "method" corresponds to the quantization method and it can be non-overflow or min-diffs being that the first might have worse results in case of outliers, the second one is the default option and the one we will use. "Input_fn" is relative to a function that does the pre-processing of the Dataset images so it can then be sent to the model on the right proportions.

```
# activate DECENT_Q Python3.6 virtual environment
#conda activate decent

# generate calibraion images and list file
#python generate_images.py

# remove existing files
rm -rf ./quantize_results


# run quantization
echo "#######################################"
echo "QUANTIZE"
echo "#######################################"
decent_q quantize \
  --input_frozen_graph ./yolo.pb \
  --input_nodes input_1 \
  --input_shapes ?,416,416,3 \
  --output_nodes "conv2d_10/BiasAdd,conv2d_13/BiasAdd" \
  --method 1 \
  --input_fn graph_input_fn.calib_input \
  --gpu 0 \
  --calib_iter 100

echo "#######################################"
echo "QUANTIZATION COMPLETED"
echo "#######################################"
```

Before executing the quantization, you will have to open the calibration function and change the path to the training dataset images according to your location.

Finally, to execute the quantization you simply run the command this commands:

cd ~

cd /home/Vidhi/Desktop/YOLO-on-PYNQ-Z2/DNNDK/quantization

./quant.sh

The next step is to compile the model with a "DPU language" so I suggest you copy the "deploy_model.pb" to the compilation directory of the repository.

cp /quantize_results/yolo.pb /home/andre/Desktop/YOLO-on-PYNQ-Z2/DNNDK/compilation

4.3 Compiling the Model with DNNDK

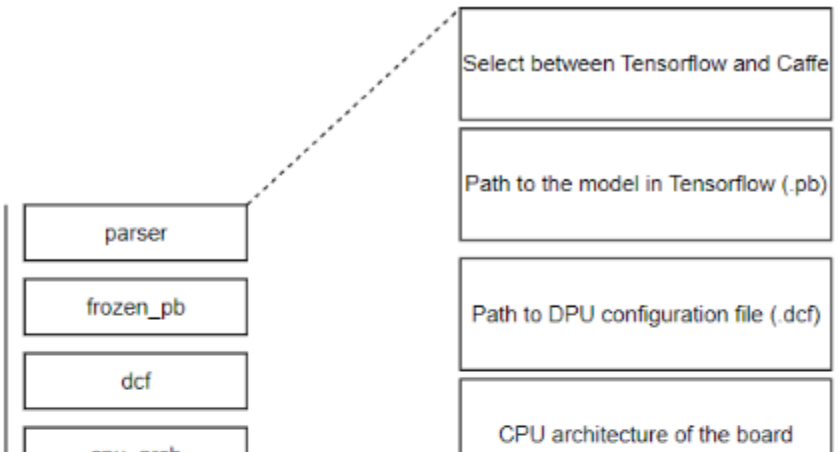
To compile the model in a binary file that the DPU can understand, there will be created a compilation script that follows a similar structure as the quantization script but with it's unique parameters. Before that we will only need a DPU description file and the YOLO model quantized and optimized in Tensorflow, which we already have.

To generate the DPU description file, use a tool called Dlet and there will only be needed a ".hwh" file from the Vivado project. This file can be hard to find but you can generally locate it on a similar path to this {Vivado_Project_location}\pynqz2_dpu\pynqz2_dpu.srcs\sources_1\bd\pynqz2_dpu\hw_handoff.

With the possession of the ".hwh" file we can proceed and generate the ".dcf" file with this command:

dlet -f {/path/to/Vivado_project}/pynqz2_dpu.hwh {/path/to/output}/pynq-z2.dcf -o

Below is the compilation script:



The parameter "parser" refers to the framework associated with the model which is Tensorflow in this case. Then in "frozen_pb" we refer the quantized YOLO model. From the two files generated on the quantization we are interested on the "deploy_model.pb". The next parameter refers to the ".dcf" file we generated earlier so the system has the idea of which DPU architecture we will be working with. The "cpu_arch" is relative to the ARM processor architecture of the PYNQ-Z2 which is arm32. The next parameter refers to the desired location of the generated files and separates them because they are very important. Then we can allow the program to generate a kernel information file on the "save_kernel" parameter. "mode" refers to debug or normal mode and choosing a normal nut should be interesting to try debugging so there are more options enabled on the API. Lastly, the "net_name" is where we name the resulting binary file.

```
# delete previous results
rm -rf ./compile
#conda activate decent
# Compile
```

```
echo "#######################################"
echo "COMPILE WITH DNNC"
echo "#######################################"
dnnc \
    --parser=tensorflow \
    --frozen_pb=./deploy_model.pb \
    --dcf=pynqz2_dpu.dcf \
    --cpu_arch=arm32 \
    --output_dir=compile \
    --save_kernel \
    --mode normal \
    --net_name=yolo

echo "#######################################"
echo "COMPILATION COMPLETED"
echo "#######################################"
```

Above is the custom compilation script used to perform this task. Execute the compilation process using the ./compile.sh command.

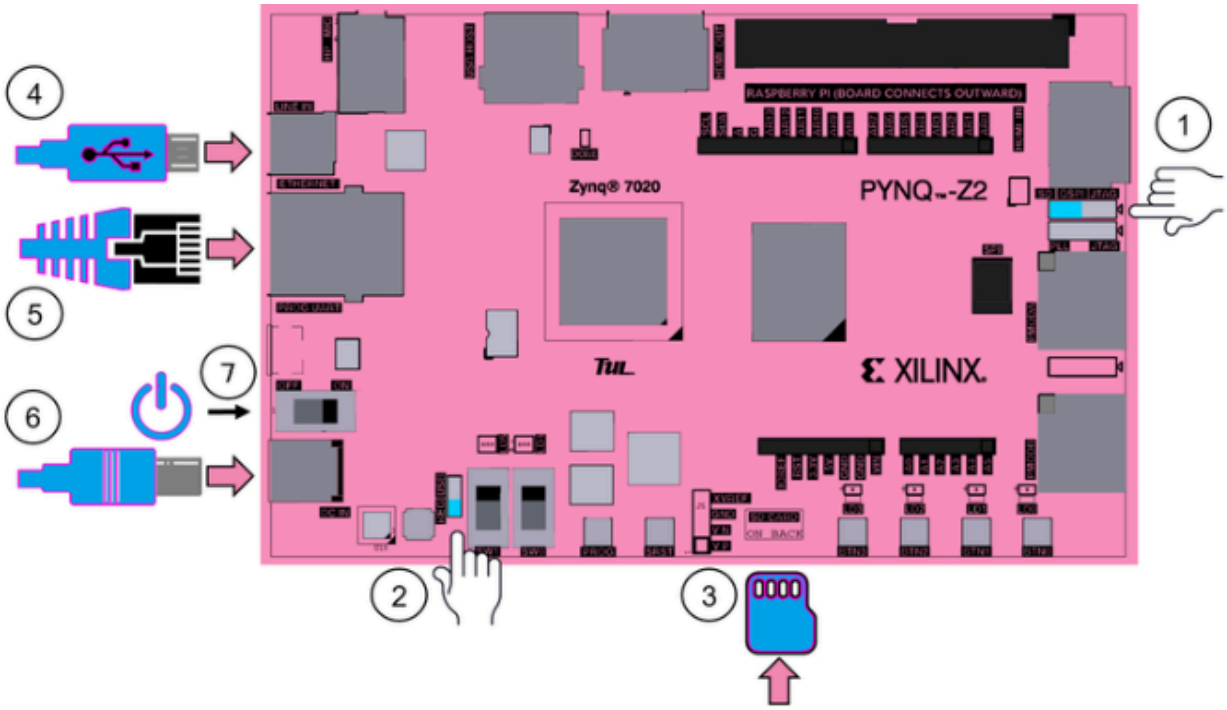
The compilation has generated a ".elf" file which is the binary file that represents the compressed Neural Network to the DPU and the remaining files are just informational files. The "yolo_kernel.info" has indeed some information's about the kernel and the image has the kernel ID and type. These files will be joined with a c++ program to make YOLO work and a Makefile to compile everything and debug the code. We will be addressing in more detail this on the next chapter. For now, you need to:

- Open "Deployment" form the repository;
- Copy the elf file and place it on the "model" folder;
- Copy the "yolo_kernel.info" and the jpg file to the "info" folder;

CHAPTER 5

DEPLOYMENT ON PYNQ-Z2

5.1 Setting Up the PYNQ-Z2 Board



1. Set the Boot jumper to the SD position. This will make the board load the Operating System on the micro SD card;
2. Set the Power jumper to REG, so the board will receive power from the power regulator and not the USB cable;
3. Insert the Micro SD card with the custom image (the one we flashed with the DPU). Remember to insert it from the back as described on the image.

4. Connect the micro USB cable between your Host computer and the board.
5. Connect the Ethernet cable (RJ45) between the Laptop and the board. Don't forget to use the adapter if your computer doesn't have the RJ45 port;
6. Connect the power adapter's coaxial port on the PYNQ-Z2 to give it the necessary power;
7. Turn on the PYNQ-Z2 and the boot sequence should correspond to a red LED turning immediately on representing that the board has power. Then the Yellow/ Green LED called Done will turn on showing that the Zynq service is ready.

5.1.1 USB connection

A USB link to the board can be used to supply power or to control the board. To establish a USB connection between the PYNQ-Z2 and the host computer we can use an application like Putty.

Before launching Putty, we have to see which port corresponds to the PYNQ-Z2. This is possible by going to the Device Manager and COM and LPT Ports. When you turn on the PYNQ-Z2 there should appear a new port there described as something similar to COM4. Then we can launch Putty by typing on the search bar it's name.

First we need to setup the communication to Serial, then write the COM port according to the one you saw at the Device Manager. Next, on the Speed domain you write 115200 which is the Baud Rate of the data transmission.

After this, a Terminal will appear and many information's regarding the Boot sequence of the PYNQ-Z2 custom image will be displayed. After that is complete, you will be asked to log in. These are the log in information's:

- **Login:** root
- **Password:** root

Now let's give the board a IP address so we can establish a SSH communication:

ifconfig eth0 192.168.2.99

On official SD card images for the PYNQ-Z2 the default IP address of the board is 192.268.2.99 and so we established the same address for simplicity.

5.1.2 Network Connection

Now to connect the board to the host computer via SSH. This is done with the RJ45 cable commonly known as Ethernet cable.

The PYNQ-Z2 already has an IP address so now we need to assign a static address to the host computer on the same range so both are connected. To do this first go to the Control Panel and then Network and Internet and after that go to the Network and Sharing Center menu. Now select the Ethernet connection and not the wireless connection.

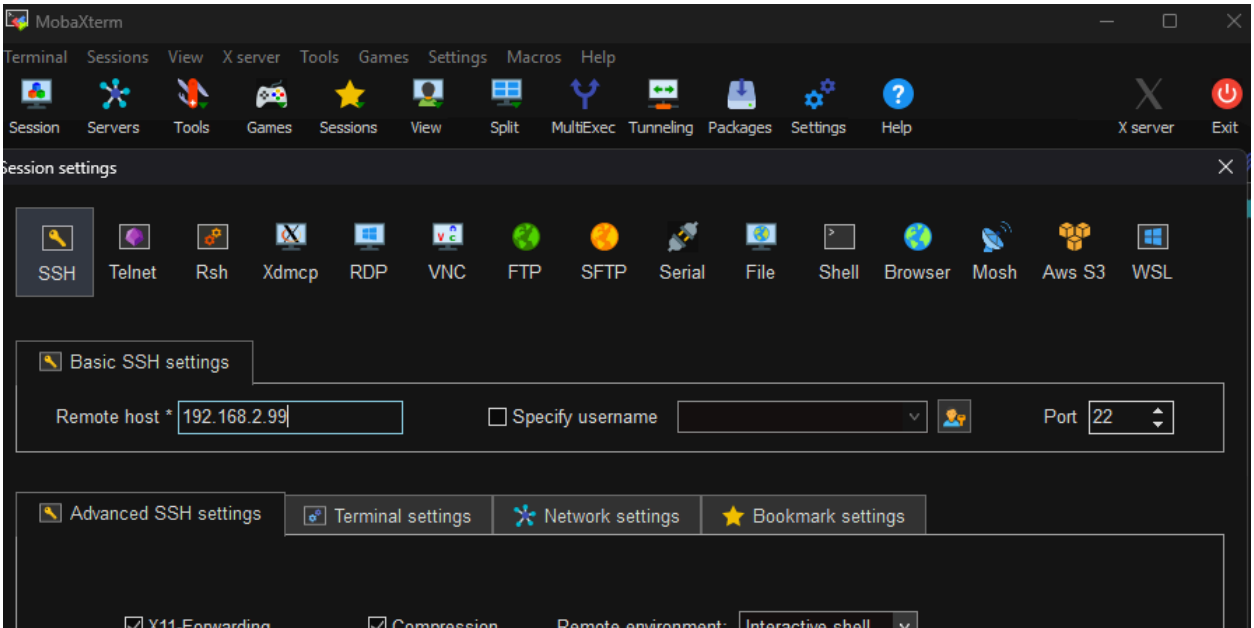
Now we need to set the Network Card of the computer with an IP on the same range as the one we gave to the PYNQ-Z2. Assign an IP between 192.168.2.1 and 192.168.2.98 for this to work.

After the configuration you have to click "OK" to save the configuration. After this there will be a SSH connection between your computer and the board.

To confirm the Network Connection we can use the PING protocol to send packets to the board's IP address. If the connection is available, the PYNQ-Z2 must confirm the reception of the packets. You can confirm this opening the Command line on Windows (CMD) and writing "ping 192.168.2.99".

5.1.3 X-11 Connection

Now that there is a safe network connection with the board, we can have total control over it using the computer. Now we will implement a Graphical interface Network protocol (GUI - Graphical User Interface). This protocol is named X11 or X Window System and it allows the development and management of graphical applications on a remote server based on UNIX Operating System using the host computer. In other words, X11 will allow a graphical interface of the PYNQ-Z2 OS and will make it easier to create Windows and interfaces.

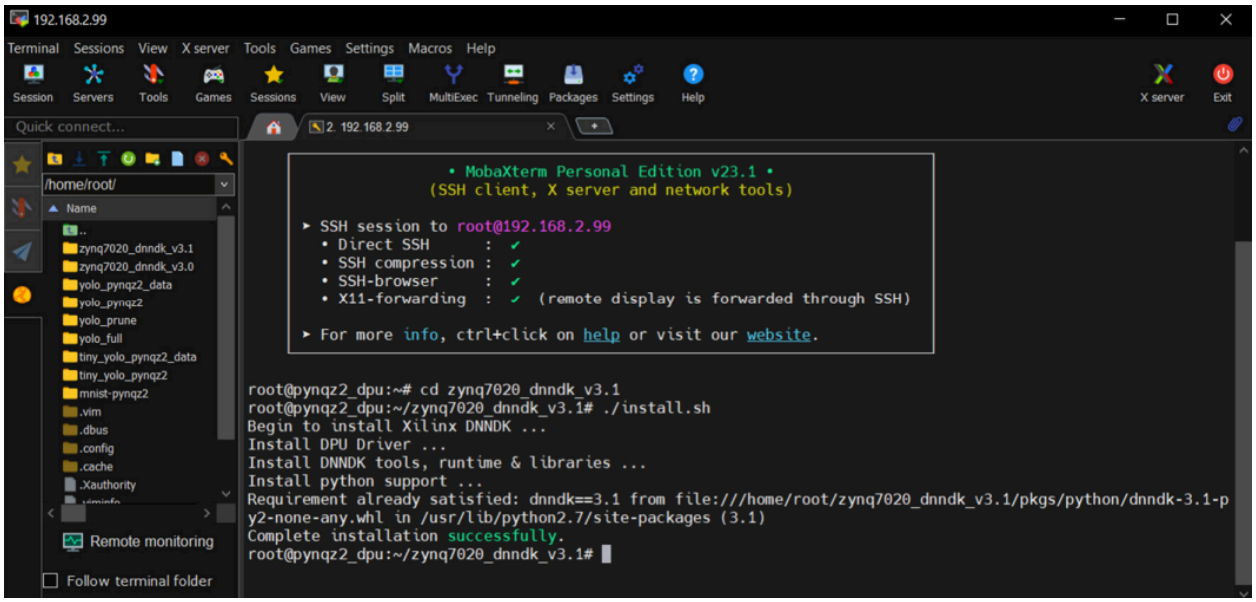


After this you will have to login on the MobaXterm Terminal. Remember that the username is "root" and the password is "root".

After the login, the first thing to do is to activate the DNNDK libraries so that later the interaction between the API and the DPU is possible. You can do it as described on the next image.

cd zynq7020_dnndk_v3.1

./install.sh



Now everything is set to execute the YOLO on the PYNQ-Z2 board.

5.2 Executing YOLOv3 on the Board

It's time to add all the things together and execute the YOLOv3 Neural Network in order to perform inference on a desired image. Previously, we've created a package containing the DPU elf file and other necessary files. We will now copy that package to the board's Operating System using the Xterm interface. The easier way to do it is to simply drag the file "yolo_pynqz2" to working area on the left side of the application.

For the deployment of the YOLOv3 on the PYNQ-Z2 we will focus on the Programs folder and on the Makefile.

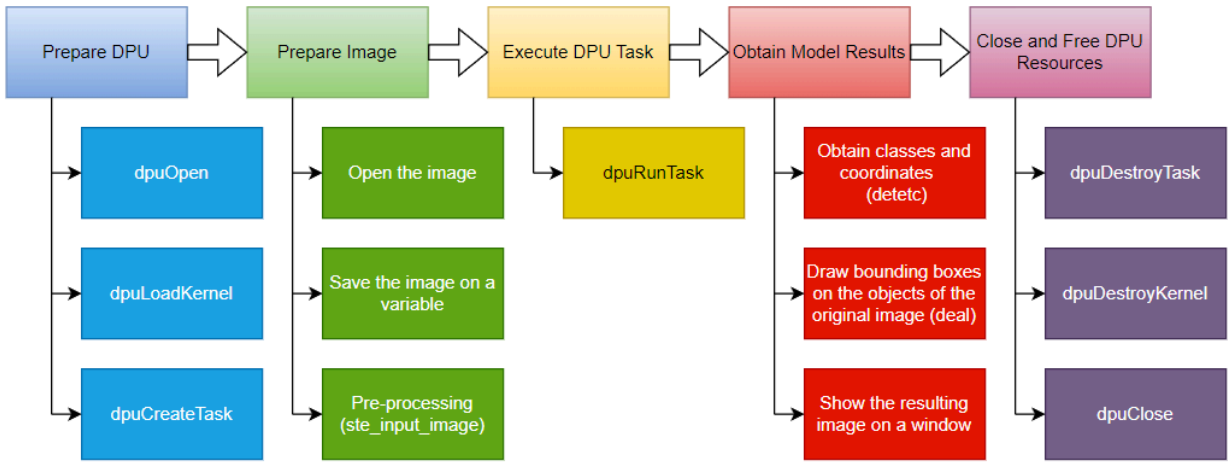
```
info
    |_yolo_kernel.info
    |_yolo_kernel_graph.jpg
model
    |_dpu_yolo.elf
objects
programs
    |_yolo_image.cpp
    |_yolo_video.cpp
dog.jpg
Makefile
```

The info folder contains those informational files that won't be used for anything. The model folder will be used to store the binary file with the YOLO instructions - the .elf file. The c++ programs with the interaction with the DPU will be a main take on this chapter as it is where every instruction is written in order to run inference on the DPU. Those files will be stored on the respective programs folder. Then there will be a test image called dog.jpg and a Makefile. The Makefile will be responsible to compile the code, include the libraries and check for errors. Finally the objects folder will store object files resulting from the compilation process. The executable file will be generated on the base location.

5.2.1 YOLO detection script

There are two c++ scripts on the programs folder. One is to perform detection's on a single given image and the other is the YOLO object detector in real time, using a USB camera.

The most important functions are the main, the detect and the deal. Analyzing the main function on the yolo_image.cpp, After invoking and executing a DPU task, for the specific Neural network, there are called some image processing functions. Essentially, the set_input_image function is responsible for the conversion and pre-processing of the image in order to respect the model requirements. The deal function has to draw bounding boxes on the respective objects detected on the image and return the edited image to the same variable. The function only knows where to draw the bounding boxes because it called the detect function to obtain the coordinates and the object class. After the processing phase, the image is shown on a window using the OpenCV library. The development flow can be described by the following diagram:



The order of events should be like this:

```
int main(const int argc, const char** argv)
{
        //Open the DPU and create a task
        dpuOpen();
        DPUKernel *kernel = dpuLoadKernel(YOLOKERNEL);
        DPUTask *task = dpuCreateTask(kernel, 0);
        int sh = dpuGetInputTensorHeight(task, INPUTNODE);
        int sw = dpuGetInputTensorWidth(task, INPUTNODE);

        //Obtain the image based on the Console input
        string name(argv[1]);
        Mat img = imread(name);

        //Pre-processing of the image
        set_input_image(task,  img,INPUTNODE);

        //Input the image to the Neural Network
        dpuRunTask(task);

        //Draw bounding boxes and classes on the image
        deal(task, img, sw, sh);

        //Show the resulting image on a window
        cv::imshow("yolo-v3", img);
```

```
        //Close DPU and free resources
        dpuDestroyTask(task);
        dpuDestroyKernel(kernel);
        dpuClose();
}
```

5.2.2 Makefile

This file has the responsibility to compile the code and include the necessary libraries. This means that it will check for errors on the code and it will call the functions needed like OpenCV and the DNNDK API.

Below is the makefile content:

```
YOLO = yolo_image

CXX      :=   g++
CC       :=   gcc
YOLOOBJ  :=   objects/yolo_image.o

# linking libraries of OpenCV
LDFLAGS  =  $(shell pkg-config --libs opencv) -lpthread -lhineon -ln2cube -ldputils

CUR_DIR =   $(shell pwd)
MODEL   =   $(CUR_DIR)/model/dpu_yolo.elf
ARCH    =     $(shell uname -m | sed -e s/arm.*/armv71/ \
        -e s/aarch64.*/aarch64/ )

CFLAGS  :=  -O2 -Wall -Wpointer-arith -std=c++11 -ffast-math

ifeq ($(ARCH),armv71)
        CFLAGS +=  -mcpu=cortex-a9 -mfloat-abi=hard -mfpu=neon
endif
ifeq ($(ARCH),aarch64)
        CFLAGS += -mcpu=cortex-a53
endif

# Define source and object file directories
SRC_DIR = programs
OBJ_DIR = objects
```

```
# List of source files
SRC_FILES := $(wildcard $(SRC_DIR)/yolo_image.cpp)

# Generate a list of object files from source files
OBJ_FILES := $(patsubst $(SRC_DIR)/%.cpp,$(OBJ_DIR)/%.o,$(SRC_FILES))

.PHONY: all clean

all: $(YOLO)

# Rule to compile .cpp files into .o files
$(OBJ_DIR)/%.o : $(SRC_DIR)/%.cpp
	$(CXX) -c $(CFLAGS) $< -o $@

# Rule to build the final executable
$(YOLO): $(OBJ_FILES)
	$(CXX) $(CFLAGS) $^ $(MODEL) -o $@ $(LDFLAGS)

clean:
	rm -f $(OBJ_DIR)/*.o $(YOLO)
```

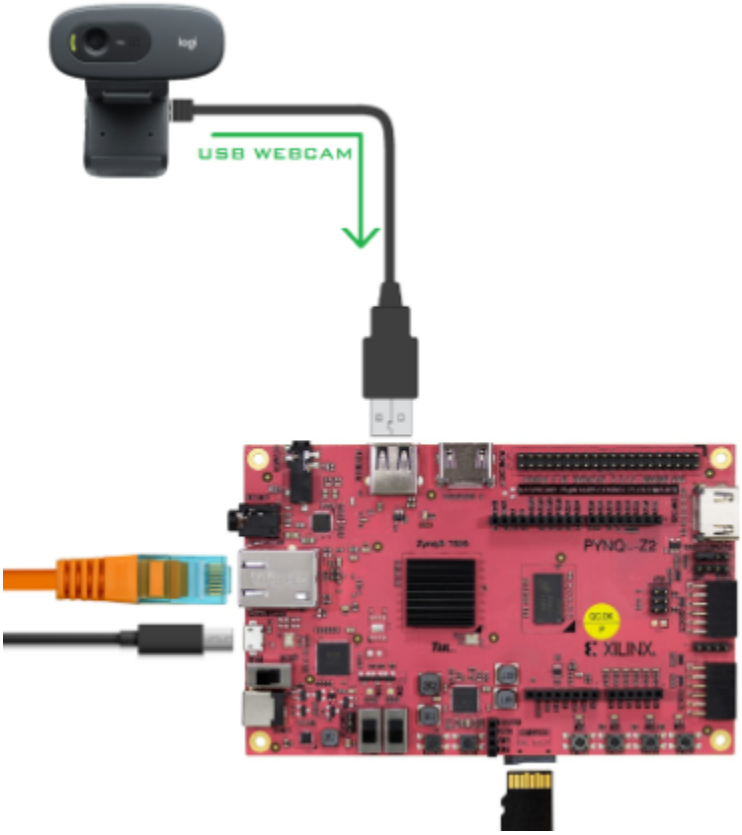
5.2.3 Run YOLOv3 on the PYNQ-Z2

First we have to compile the yolo_image script. The makefile is already set to do that so you just have to execute make. Then the object files will be placed on the objects directory and a executable file named "yolo_image" will be generated. You just execute the script and indicate which image you want to perform detection's on.

```
cd yolo_pynqz2
make
./yolo_image test.jpg
```

5.3 Implementing Real-Time Object Detection

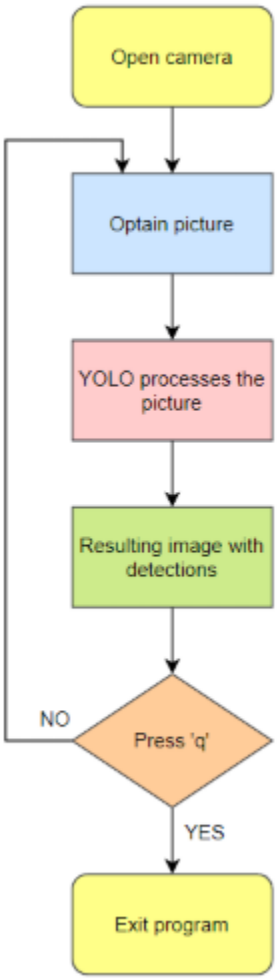
To make this real-time object detector possible, you will need to have a camera, of course. The choice here was a USB webcam, as it interfaces directly with the PS side of the board, making it simpler to deal with.



The c++ program we used to detect objects on a single image is definitely the starting point to make the real-time application. The basic structure stays the same, and we just make changes on the main function.

We have preserved the basic structure of the YOLO on the PYNQ-Z2, but we add an infinite loop. Not so infinite of course because one thing has got to give: either press 'q' or the program crashes.In any case, on the loop, a picture is taken by the camera and then gets turned into an image (goes through the pre-processing). After that, the image is processed just like on the

original program and the result will be the image with the detections on a window. This process repeats, making it "real-time".



The makefile is responsible for giving the PS all the specifications for the compilation process of the c++ program. There needs to be some minor changes on the original Makefile you have on the Deployment folder of the repository (It's the same on YOLO and on tiny YOLO). We

only need to point the name we want the executable to have, the object file name and the c++ program we want to be compiled.

Save the changes to the file and open the yolo_pynqz2 folder on the console. To compile the desired yolo_video.cpp program, you just execute this command: make.

./yolo

A window will pop up with the video captured by the camera. There is a considerable delay and the frame rate is miserable, but it gets the job done.

6. Results and Analysis

11. References