VIETNAM NATIONAL UNIVERISTY - HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY (HCMUT)
FACULTY OF COMPUTER SCIENCE & ENGINEERING

CAPSTONE PROJECT

ACCELERATION OF GENERATIVE ADVERSARIAL NETWORKS FOR IMAGE GENERATION ON THE SOC-FPGA PLATFORM

MAJOR: COMPUTER ENGINEERING
COMMITTEE: CE-CC02

Supervisors:

Assoc. Prof. Dr. Pham Quoc Cuong - HCMUT

Reviewer:

Assoc. Prof. Dr. Tran Ngoc Thinh - HCMUT

Authors:

Le Ngoc Minh Thu - 2053476

Huynh Trung Nhat - 2053294

Do Huu Thanh Thien - 2053453

HO CHI MINH CITY - June-2024

This thesis is dedicated for our parents and our instructors at HCMUT.

CONTENTS

List of Figures vii

List of Tables xi

Acknowledgment xiii

Abstract xv

1 Introduction 1
- 1.1 Introduction . . . 1
- 1.2 Research objective . . . 3
- 1.3 Research scope . . . 3
- 1.4 Research subject . . . 3
- 1.5 Outline . . . 4

2 Background and Related work 5
- 2.1 Background . . . 5
  - 2.1.1 Field Programmable Gate Array and System on Chip . . . 5
  - 2.1.2 Kria KV260 platform . . . 7
  - 2.1.3 ZCU106 platform . . . 9
  - 2.1.4 PYNQ framework . . . 10
  - 2.1.5 Deep Learning and Deep Neural Network . . . 11
  - 2.1.6 Generative Adversarial Networks . . . 13
  - 2.1.7 Deconvolution Neural Network . . . 19
- 2.2 Related work . . . 22
  - 2.2.1 Generative Adversarial Networks Applications . . . 22
  - 2.2.2 Generative Adversarial Networks on FPGA . . . 23

3 Overall Architecture 27
- 3.1 General architecture . . . 27
- 3.2 Dataflow of Deconvolution Multi-kernel Processor . . . 29
- 3.3 Optimized GAN Execution Research and Survey. . . . 32
  - 3.3.1 Introduction . . . 32
  - 3.3.2 Overflow Handling in Fixed-Point Computation . . . 32
  - 3.3.3 Deconvolution Methodologies . . . 33
  - 3.3.4 Comparative Analysis and Conclusion . . . 34

3.3.5 Final Conclusion 34

4 Implementation **35**
4.1 System implementation 35
4.1.1 Processing system 35
4.1.2 PS-PL communication 37
4.1.3 Acceleration core - TOP 37
4.2 Deconvolution Multi-kernel Processor Implementation 38
4.2.1 Signal Descriptions 38
4.2.2 Functional Descriptions 40
4.3 Core Overlap Processor Implementation 49
4.4 Tilling and Gather Processor Implementation 55
4.4.1 Detailed Overview 56
4.4.2 Integration of Till Gather Cores and Buffers 57
4.4.3 Operational Details of Tilling Gather Core 59
4.5 Parameters of the acceleration core 60
4.6 Register Bank 61
4.7 Software Implementation 63
4.7.1 Supporting Software 63
4.7.2 Application Software 73

5 Experimental Results **77**
5.1 Performance Model 77
5.2 Experimental Setup 79
5.3 Synthesis Results 85
5.4 Simulation Results 86
5.5 Performance Validation and Analysis 87
5.5.1 DMA Transferring Time 87
5.5.2 Execution time and Speed Up 87
5.5.3 Image Generation Quality 93
5.6 State-of-the-art comparisons 95
5.7 Conclusion 96

6 Conclusion **99**

Bibliography **101**

LIST OF FIGURES

2.1 Basic FPGA Architecture 6
2.2 Basic MPSoC Zynq Architecture 7
2.3 Kria KV260 Vision AI Starter Kit [1] 8
2.4 ZCU106 Evaluation Kit [2] 10
2.5 PYNQ Framework [1] 11
2.6 An example of AI field 12
2.7 An example of an artificial neural network 13
2.8 An example of deep neural networks 14
2.9 Basic network architecture of GAN 15
2.10 Generator Network basic flow 15
2.11 Discriminator Network basic flow 16
2.12 MNIST dataset [source] 16
2.13 CELEB-A dataset [source] 17
2.14 Generator architecture for two datasets 18
2.15 Discriminator architecture for two datasets 18
2.16 Comparison between conventional convolution and deconvolution 20
2.17 Deconvolution computation illustration (stride = 2) 21

3.1 General architecture of Generator Network accelerator 28
3.2 Dataflow structure of Deconvolution Multi-kernel Processor . . . 30
3.3 Memory structure of Feature BRAMs and Weight BRAMs 31
3.4 Dataflow of input feature and output feature 32

4.1 System implementation on Xilinx SoC 36
4.2 System implementation on Xilinx SoC 36
4.3 Central Direct Access Memory IP on ZYNQ UltraScale+ Processing System 38
4.4 Input Interface of DCMKP 39
4.5 Output Interface of DCMKP 39
4.6 General design of Deconvolution Multi-kernel Processor 41
4.7 Deconvolution Multi-kernel Processor Implementation 42
4.8 Row-overlap accumulating processor block diagram 44

4.9 Block diagram for adding each pixel of registers and input data . . 45
4.10 Block diagram of INIT state 46
4.11 Block diagram of WRITE state 46
4.12 Block diagram of WAIT state 47
4.13 Block diagram of READ state 48
4.14 Column-overlap accumulator FSM 48
4.15 Block diagram to illustrate the Core Overlap Processor Function . 50
4.16 Core Overlap Processor Implementation 50
4.17 Row overlap process state machine 51
4.18 Column overlap process state machine 52
4.19 Core Overlap Processor Timeline Diagram 53
4.20 Data flow from Core Overlap Processor to Tilling and Gather Processor 55
4.21 Tilling and Gather machine implementation 57
4.22 Tilling Gather Buffer implementation 58
4.23 Tilling Gather Core Implementation 59
4.24 MNIST and Celeb-A datasets were trained from scratch for 60 epochs 64
4.25 Visualization of evaluation techniques for MNIST dataset training (2 dimensions - 1 channel) 66
4.26 Scatter plots of generated and truth image in different dimensions for Celeb-A dataset (4 dimensions - 3 channels) 67
4.27 Histogram of distance comparison between generated image and truth image to its center for Celeb-A dataset 68
4.28 Comparison of low-resolution and high-resolution images from the DIV2K dataset. 69
4.29 Modified SRGAN Generator architecture incorporating deconvolution layers. [3] 69
4.30 SRGAN Discriminator architecture. [3] 70
4.31 Evaluating the impact of GAN and content losses on image quality. 71
4.32 Assessment of GAN and perceptual losses on image generation. . 71
4.33 Integration of GAN, content, and perceptual losses and their effect on the output. 72
4.34 Application Software Operation Flow Chart 76

5.1 Kria KV260 board diagram 80
5.2 ZCU106 board diagram 81
5.3 Block design for DMA transferring time experiment 83
5.4 A flow diagram illustrates the data transfer and processing between software and hardware components. 84
5.5 Detailed Power Consumption report 86

5.6 Deconvolution multiplication simulation on Vivado 87
5.7 Execution time of Deconvolution Neural Networks for 2 datasets between multiple platforms 91
5.8 Data throughput and inference time of acceleration core correlation 92
5.9 Comparison of output image between two datasets on single layer 93
5.10 Comparison of MNIST images: the left side shows images generated by the FPGA, and the right side shows the generated images on software. 94
5.11 Comparison of Celeb-A images: the left side shows images generated by the FPGA, and the right side shows the generated images on software. 95

LIST OF TABLES

2.1 XCK26 specification 9
2.2 ZCU106 specification 9
2.3 A summary of the parameters in the deconvolution layer in GAN . 20

3.1 Comparison of GAN model performance with and without overflow handling using MNIST and Celeb-A datasets. 33
3.2 Detailed comparison of computational operations required for zero-padding vs. direct implementation of deconvolution across various parameters. 34

4.1 Address Mapping of PL memories 38
4.2 List of I/O pins of DCMKP 40
4.3 Register Bank description 61
4.4 SSIM and PSNR evaluation results for DIV2K datasets. 73

5.1 Terms definition in the performance model 78
5.2 Hardware resources on Kria KV260 and XCU106 boards 85
5.3 GOPs and GOPs/DSPs 86
5.4 DMA transferring time of 2 methods. 88
5.5 Performance comparison across different computing platforms. The table showcases processing times (in seconds) for various deconvolution layers. 89
5.6 DMA Execution time for the Deconvolution Neural Networks (not including Batch Normalization and Activation Function 90
5.7 Execution time for the Deconvolution Neural Networks between different platforms (not including Batch Normalization and Activation Function 91
5.8 FID scores for FPGA generated-image and software training 94
5.9 Comparison of FPGA implementations 96

ACKNOWLEDGMENT

First, we would like to express our sincere thanks to *Associate Professor Pham Quoc Cuong* for his enthusiastic support during my university studies and report writing. Thank you, Computer Engineering K19 alumni *Mr Pham Dinh Trung* for wholeheartedly following our research progress, supporting us by giving advice and experience in solving complex problems and developing regular directions for the research report. We would like to thank *Mr Huynh Phuc Nghi* in CE-lab for spending their time to exchange knowledge with us in my study field.

Besides, we would like to express my gratitude and respect to all lecturers in the Faculty of Computer Science and Engineering and the Ho Chi Minh City University of Technology. They have helped us gain a solid knowledge background for our report research.

Moreover, we are humbly grateful for having the chance to work with outstanding colleagues and seniors at Uniquify Vietnam, Truechip Solutions and Bosch Engineering and Solutions Vietnam. They have taught us professional knowledge, study and work skills during our internship. Stepping out from the practical industry environment, we gain confidence and enthusiasm with the industry domain we pursue.

We thank our families for giving us solid spiritual support and sympathy so that we can devote all our energy to the report. Thank you, a small group of Computer Engineering close friends for their dedication and mutual support during a 4-year journey. We hope that, finally, each of us will achieve our worthy desires.

And lastly, thank you three of us have endlessly tried hard for this Computer Engineering project although we have encountered multiple obstacles and misunderstandings.

Ho Chi Minh City, May 01^{st}, 2024
Le Ngoc Minh Thu
Do Huu Thanh Thien
Huynh Trung Nhat

ABSTRACT

In these days, AI technology has successfully been applied in various industries such as agriculture, medical, automation, transportation and security. AI is now a potential trend due to the rapid development of hardware platforms. Powerful hardware can significantly speed up the complex computations of AI algorithms. There are many hardware platforms that contribute to the AI training performance including multi-core Central Processing Unit (CPU), and Graphic Processing Unit (GPU). However, FPGA is an efficient choice for AI applications because it strikes a compromise between processing speed and flexibility to custom different algorithmic circuits. Generative Adversarial Networks is a type of machine learning model that is used to generate new data samples based on the learning distribution and has been used in a variety of applications. Despite its ability to produce realistic data, the inference time of this model is significant because the networks are typically deep and complex, and the training process is iterative on a large dataset. Therefore, meeting high performance in terms of computation and memory requirements is a challenging problem. In this report, we propose a novel architecture for accelerating the Generative Adversarial Networks model on a FPGA-based SoC platform. This architecture is highly pipelined, parallel and scalable for many FPGA devices. In the final phase, an application for generating synthetic images using multiple datasets will be implemented using GAN accelerated by FPGA.

1

INTRODUCTION

In this section we briefly introduce our thesis. This section includes an introduction to the context of this project, the research objective, the research scope, the research subject and the contents of this thesis.

1.1. INTRODUCTION

AI technology has developed rapidly in recent years and brought multiple benefits to industrial business and daily life. There are six significant sub-fields, each with its focus and applications: Machine learning, Neural network, Deep learning, Natural language processing, Cognitive computing, and Computer vision. Deep learning is a subset of machine learning that uses artificial neural networks to learn from data. Neural networks are inspired by the human brain, and they are made up of interconnected nodes that learn to process information in a way that is similar to the way the human brain does. Generative Adversarial Networks is one of deep learning algorithms that consist of two neural networks: a generator and a discriminator. The generator is responsible for creating new data, while the discriminator is responsible for distinguishing between real and generated data. The two networks are trained together in a game-like setting, where the generator tries to fool the discriminator into thinking that its output is real, and the discriminator tries to become better at distinguishing between real and generated data. GANs have been used in a variety of applications including image generation, text-to-speech conversion, text-to-image conversion, photo editing, image resolution, and vice versa. In comparison with other generative models, GANs are relatively easy to train and converge faster. The training process of GAN is usually done on GPUs and typically trained offline. However, the inference time, which is the time it takes to gener-

1

ate new data, can be prohibitively long for real-time applications such as virtual reality and augmented reality. Additionally, GANs are used to generate a large amount of data which takes a lot of time to process complex computations. Therefore, edge devices such as embedded CPUs or micro-controllers inefficiently operate computationally intensive tasks. The reason is that these platforms have insufficient memory for parameter storage (for microcontrollers) and poor parallel computing utilization.

There are many hardware platforms to run inference of generative adversarial networks, such as multi-core CPU, GPU and FPGA. However, FPGA is proven to have the highest power efficiency in resource-limited edge computing applications in [4]. Although FPGA operates at a lower frequency than GPU and CPU, it can be configured for a particular purpose so that it can utilize its resources most optimally. Therefore, much research has been dedicated to accelerating complex algorithms on FPGA to improve the inference. FPGA is one of the most efficient platforms for implementing edge devices. However, FPGA is poor for running software applications. Thus, SoC-FPGA platform is a good choice since it integrates FPGA and microprocessors. Due to this integration, application parts require sequential tasks that can be run on a microprocessor. In contrast, parts with parallel tasks can be offloaded on FPGA. ASIC is superior to FPGA in processing speed as well as power consumption. However, it cannot be reconfigured, so ASIC lacks flexibility compared to FPGA. With FPGA, we can harness their parallel nature to simultaneously process multiple input pixels by deploying multiple processing elements. Additionally, memory access can be optimized by redesigning the memory structure to store the forest's parameters. Compared to CPUs, accessing memory in an FPGA is more efficient. For specific use cases or datasets that require tailored designs, FPGAs demonstrate their flexibility by allowing re-synthesized with new designs optimized for those specific scenarios. Hence, this approach reduces the effort needed to implement an architecture that performs well across all possible cases. Moreover, when carefully designed with specialized hardware knowledge, FPGA can deliver high performance and energy efficiency.

This thesis studies deconvolution algorithm inside Generative Adversarial Networks and designs an architecture on FPGA to accelerate Generator network for image generation task. The whole design including four main deconvolution computation cores which can speed up the performance of computing four times at the same time. The computing cores supports for processing with multi-kernels data. The pipelining technique is implemented to enhance its performance. The Overlap Processor and Tilling-Gather Buffer are also designed in the pipelining approach to maximize the operating frequency and data throughput of the system. The whole system is a combination of compo-

nents designed with Verilog and Xilinx IP to optimize performance and flexibility. The system will be deployed and evaluated on ZCU106 board. The hardware architecture also supports multiple datasets configurations to increase the offloading tasks to hardware.

1.2. RESEARCH OBJECTIVE

These are the objectives of this thesis:

- First is to design an architecture to accelerate deconvolution network with multi-kernels so that it can operate on multiple platforms. This architecture is designed to be compatible with major applications using Generative Adversarial Networks.
- Second is to implement the design on ZCU106. Multiple configurations of the system are also supported to process, such as the number of input features, the size of each channel feature and kernels and stride parameters.
- Third is to evaluate the system performance. Additionally, two different datasets are used to evaluate the performance of the accelerator compared with Intel.
- Apply the design to solve Image Super-Resolution task.

1.3. RESEARCH SCOPE

This thesis implements a generator network accelerator on Zynq UltraScale+ MPSoC. The communication between the microprocessor and accelerator is performed via AXI bus and it is out of scope for this thesis. The design relies on Xilinx IPs, such as Block Memory Generator and CDMA, to achieve optimal performance on the Xilinx platform. After implementing the acceleration core, a software application is written to validate the functionality, execution time, accuracy and inference time between PS and PL. The targets of this architecture are the majority of FPGA devices. Therefore, device-specific detail implementations will not be covered here.

1.4. RESEARCH SUBJECT

Based on the research scope and research objective, these are the research subjects of this thesis:

- Generative Adversarial Networks algorithm and applications.

- Data transfer between Processing System (PS) and Programmable Logic (PL).
- Architecture of deconvolution for multiple kernels acceleration core on FPGA.
- Application of deconvolution for multiple kernels accelerator.

1.5. OUTLINE

The rest of the thesis is organized as follows.

- **Chapter 1 - Introduction:** Present the situation to implement and accelerate the deconvolution network. Then, define the objectives and scopes of the project. Brief descriptions of chapters are also described here.
- **Chapter 2 - Background and Related works:** Analyze theories needed to understand this thesis, including FPGA, SoC and Generative Adversarial Networks. Then, related works are analyzed to find out the novel ideas to apply to this report.
- **Chapter 3 - Proposed Architecture:** Abstract architecture is described in a top-down approach and general design of components.
- **Chapter 4 - Implementation:** Detail implementations of the system and components.
- **Chapter 5 - Evaluation:** Evaluate performance and resource usage of the system
- **Chapter 6 - Conclusion:** Summary of the current system's benefits and drawbacks and how to enhance the design in the future.

2

BACKGROUND AND RELATED WORK

This section discusses some background theories related to the thesis, including hardware and software concepts. Then, some related works are investigated to examine their advantages and disadvantages to determine which ideas can be applied to this thesis.

2.1. BACKGROUND

2.1.1. FIELD PROGRAMMABLE GATE ARRAY AND SYSTEM ON CHIP

In the contemporary technology landscape, microprocessors and micro controllers represent the norm for carrying out tasks. However, they may not always be optimized for specialized assignments, such as the execution of complex algorithms like Deep Learning or Random Forest. These hardware platforms may become uneconomical in terms of chip area and power consumption due to these complex algorithms computationally intensive execution patterns. To address this challenge, Application Specific Integrated Circuits (ASICs), Graphic Processing Units (GPUs), and Field Programmable Gate Arrays (FPGAs) offer more suitable alternatives. While ASICs may be costly and inflexible for other hardware implementations, FPGA technology allows for greater flexibility in reconfiguring hardware implementation. Notable industry leaders, such as AMD-Xilinx and Intel-Altera, offer FPGA technology. Moreover, compared to GPUs - which are easy to program, faster than CPUs, and offer many well-tested tools - FPGA technology can deliver the same or even better speed if designed carefully and consume less power than CPUs and GPUs.

FPGA technology is comprised of Configurable Logic Blocks (CLBs), which

implement logic gates, flip-flops, and programmable interconnects for connecting CLBs, IOs, and other components such as DSPs, BRAMs, and URAMs. To configure the hardware, hardware description language (Verilog, VHDL) synthesis is completed to generate a bitstream file. Modern FPGAs also support dynamic partial reconfiguration, which allows for the modification of a part of the design's netlist at runtime while other parts continue to operate. In Figure 2.1, CLB is for Configurable Logic Blocks; IC is for Interconnect, and IO is for Input Output [5].

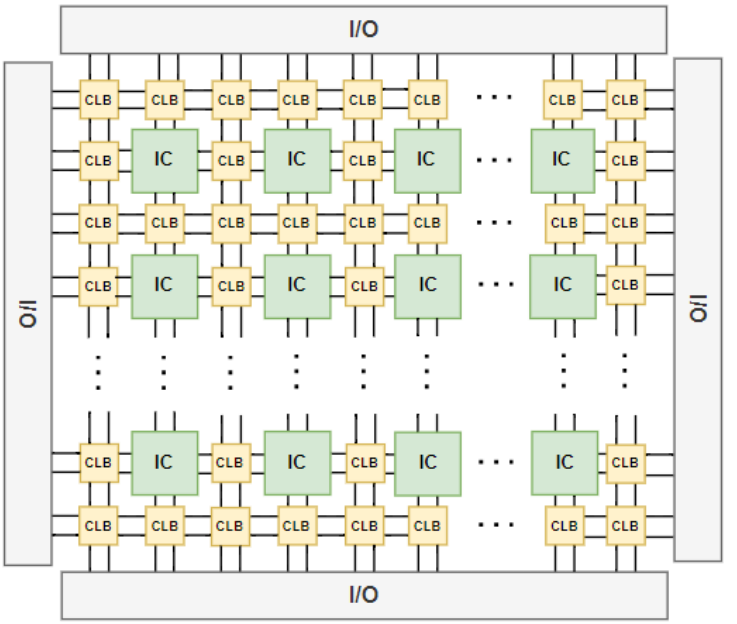


Figure 2.1: Basic FPGA Architecture

FPGAs are integrated circuits that consist of CLBs, programmable interconnects, and IOs for communication with external devices. FPGA technology is a powerful tool for designing and implementing specialized hardware solutions, and its architecture is widely applied by notable companies such as AMD-Xilinx and Intel-Altera.

As hardware systems become more complex, the integration of numerous components, such as memory, microprocessors, and programmable logic, is required. System-on-Chip (SoC) has been designed to combine hardware resources into a single chip, and for Xilinx SoCs, it is referred to as the Multi-Processor System on Chip (MPSoC)[6]. It combines programmable logic and

a fixed processor within a chip, connected by Advanced eXtensible Interfaces (AXI). The co-design workflow, which consists of hardware and software development, is widely applied, and a basic MPSoC is divided into two regions: Programmable Logic (PL) and Processor System (PS). PL is an FPGA that is strong for processing parallel tasks, while PS is efficient for processing sequential, general-purpose tasks, and it is an ARM Cortex-A9[6].

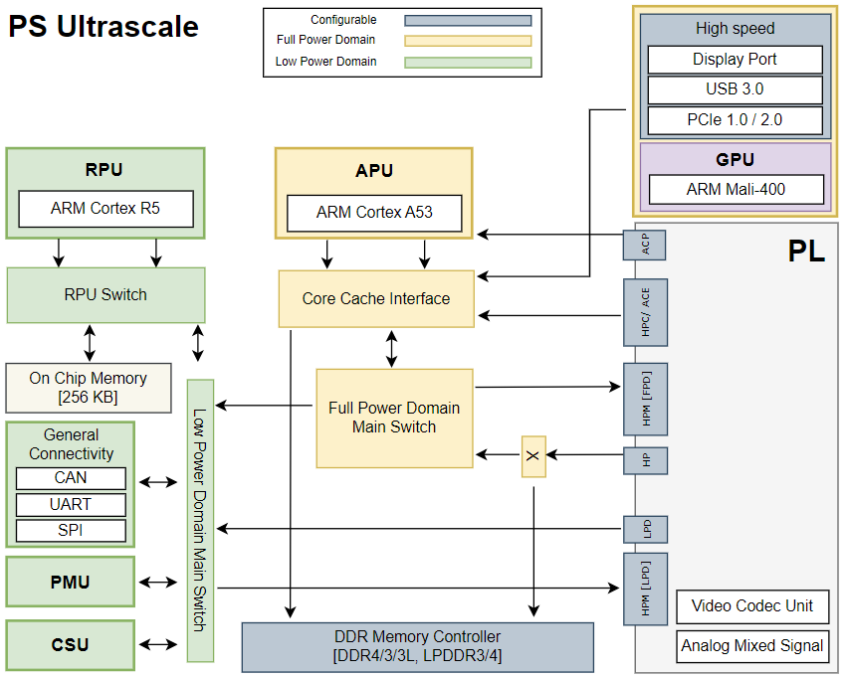


Figure 2.2: Basic MPSoC Zynq Architecture

2.1.2. KRIA KV260 PLATFORM

Kria KV260 [1] (Figure 2.3) is a board kit for developing AI vision applications. Equipped with advanced functionalities and features, the KV260 is a game-changer for professionals seeking to optimize their development process and augment productivity. One of the main advantages of the KV260 is its ability to support complex systems and edge devices, making it an indispensable tool that can assist you in achieving your objectives seamlessly and efficiently. This board kit has built-in hardware components supporting various applications such as smart city, machine vision, security cameras, and industrial applications. Additionally, the KV260 boasts a user-friendly interface and intuitive soft-

ware that simplifies the development process and reduces the time required for debugging and testing. This board kit includes the following:

- K26 SoM
- 8 interfaces support camera connectivity
- MIPI sensor interfaces
- ISP
- HDMI, DisplayPort Outputs
- 1Gb Ethernet
- USB 3.0/2.0
- microSD card slot

The strength of this kit comes from the built-in K26 System-on-Module (SoM). The K26 System-on-Module is a highly capable and versatile device, renowned for its impressive processing power and efficient power consumption, which has made it a popular choice among developers and engineers. Moreover, the K26 System-on-Module is designed to offer a wide range of connectivity options, making it easy to integrate into a diverse array of systems and applications. This SoM consists of XCK26 SoC Zynq Ultrascale+ MPSoC, 4GB 64-bit DDR4, 16GB eMMC, Trusted Platform Module (TPM), 512Mb QSPI,... XCK26 is designed to fit in the acceleration of vision AI applications. This report focuses on the developed acceleration core on the programmable logic part of XCK26. Detailed resource information is listed in Table 2.1.

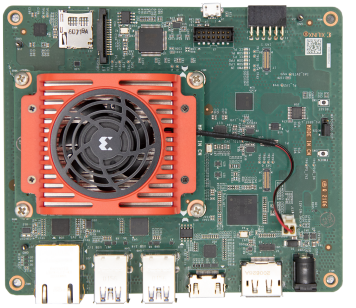


Figure 2.3: Kria KV260 Vision AI Starter Kit [1]

Table 2.1: XCK26 specification

* **Resources**	**Information**
MPSoC Chip	XCK26
Available IOBs	186
LUT Elements	117,120
Flip-Flops	234,240
Block RAMs	144
Ultra RAMs	64
DSPs	1248

2.1.3. ZCU106 PLATFORM

The ZCU106 Evaluation Kit [2], developed by Xilinx, serves as a robust platform for designing and evaluating applications based on the Zynq UltraScale+ MPSoC. This kit is particularly useful for various domains, including video conferencing, surveillance, advanced driver-assisted systems (ADAS), and streaming applications. Similar to Kria KV260 platform, this evaluation kit is a flexible platform to debug and validate the hardware acceleration application with a huge of hardware resources.

Notably, it supports 4KP60 video encoding and decoding using H.264/H.265 codecs. The ZCU106 boasts high-speed interfaces (HDMI, PCIe, USB3, DisplayPort), memory options, and FPGA fabric for custom designs.

Detailed resource information is listed in Table 2.2

Table 2.2: ZCU106 specification

* **Resources**	**Information**
MPSoC Chip	XCZU7EV
Available IOBs	260
LUT Elements	230,400
Flip-Flops	460,800
Block RAMs	312
Ultra RAMs	96
DSPs	1728

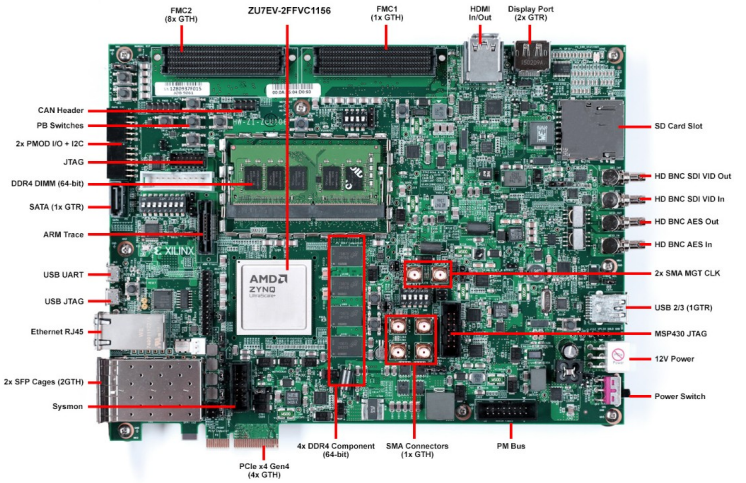


Figure 2.4: ZCU106 Evaluation Kit [2]

2.1.4. PYNQ FRAMEWORK

Xilinx provides a comprehensive range of IPs and platforms that can be conveniently accessed through programmable logic parts. However, this process requires a certain level of expertise in Hardware Language Description (HDL) and hardware design. Fortunately, PYNQ - which stands for Python Productivity for ZYNQ -[7]framework offers a solution for developers who seek to quickly develop subordinate tasks like IO processing and video streaming by utilizing Xilinx platforms. PYNQ framework is compatible with a number of Xilinx platforms, including Zynq-7000 SoC series, Zynq UltraScale+, Zynq RFSoC, Alveo accelerator boards, and AWS-F1.

The PYNQ software framework is a must-have for developers who want to streamline their design process. With its all-in-one solution for creation, execution, and testing, PYNQ eliminates the need for separate software installations, maintenance, and debugging efforts, ultimately saving developers valuable time. The structure of PYNQ is represented below. The framework employs Python language and is equipped with several Python packages that can be used to extend software applications running on processing system. It is a significant advantage, providing developers with flexibility and the ability to leverage powerful machine learning libraries for prototyping deployments on edge devices, such as robots or autonomous vehicles, with PYNQ. Additionally, PYNQ contains several overlays, which are hardware libraries that can be used to configure programmable logic for accelerating software applications. For ex-

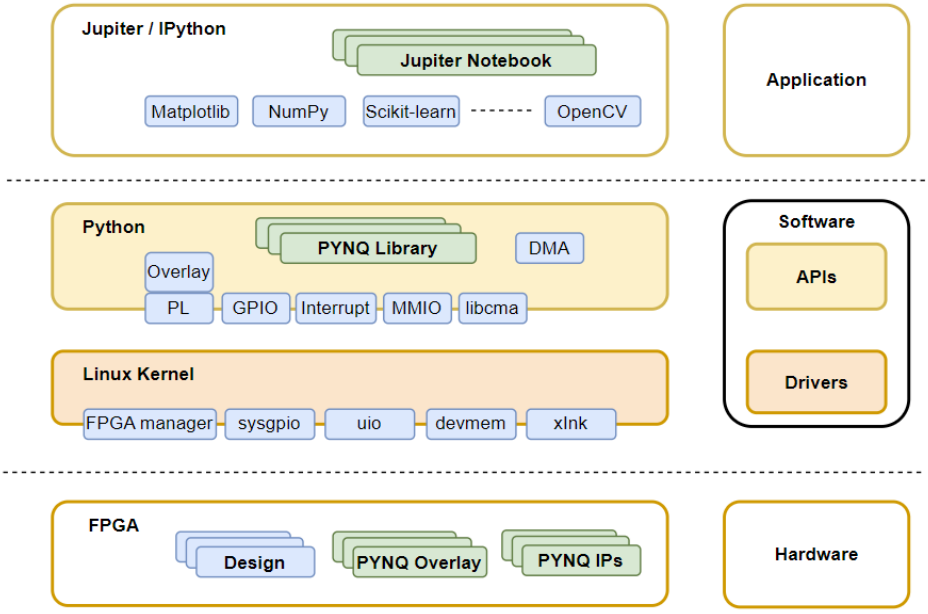


Figure 2.5: PYNQ Framework [1]

ample, users can use a Baseoverlay to create an HDMI connection controlled by HDMI IP in programmable logic without requiring any knowledge of hardware design or HDMI.

PYNQ allows users to program in the Jupyter Notebook environment, which is an effective way to organize code. This feature is particularly useful for users who wish to develop complex applications and keep track of their code.

2.1.5. DEEP LEARNING AND DEEP NEURAL NETWORK

DEEP LEARNING

Deep learning is a subset of machine learning, which is essentially a neural network with three or more layers. These neural networks attempt to simulate the behavior of the human brain-albeit far from matching its ability—allowing it to "learn" from a large amount of data. While a neural network with a single layer can still make approximate predictions, additional hidden layers can help to optimize and refine for accuracy.

Deep learning drives many artificial intelligence (AI) applications and services that improve automation, performing analytical and physical tasks without human intervention. Deep learning technology lies behind everyday prod-

ucts and services (such as digital assistants, voice-enabled TV remotes, and credit card fraud detection) as well as emerging technologies (such as self-driving cars).

2

DEEP NEURAL NETWORK

A deep neural network (DNNs) [8] is a type of artificial neural network with multiple layers. Data flows from input to output layers. Neurons apply activation functions to inputs. Training adjusts connection weights to minimize errors. DNNs can learn complex features from data. Convolution Neural Networks (CNNs) [9] handle images, Recurrent Neural Networks (RNNs) [10] work with sequences. They excel in image recognition, speech, and more. Challenges include data and computational demands. Specialized hardware accelerates them. Ethical concerns include bias and privacy. DNNs are a cornerstone of modern AI.

To gain profound insights into the concept of deep neural networks, it is essential to trace the revolution in the Figure 2.6. Prior to the emergence of deep networks, foundational components like ML and ANN had to be established.

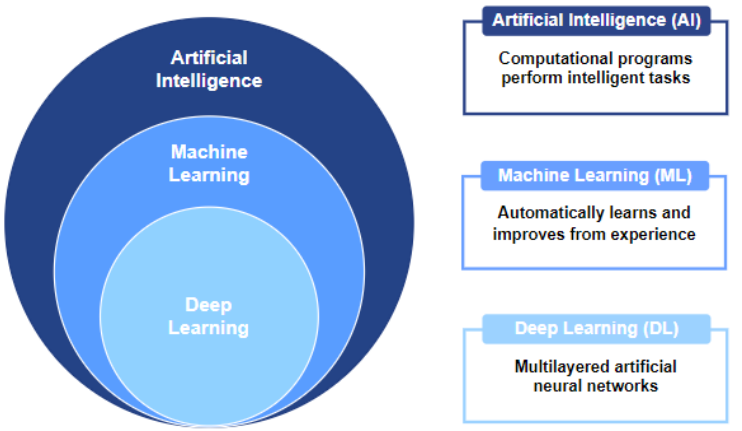


Figure 2.6: An example of AI field

Initially, the foundation of machine learning needed to be established. ML relies on statistical models, such as linear regression models. These models are trained using a dataset, allowing us to update and determine appropriate weights for predictions. Subsequently, these models are employed for prediction tasks. Within this framework, data preprocessing becomes crucial, involving the selection of relevant input features.

The evolution of creating models through the learning process gave rise to Artificial Neural Networks (ANNs). ANNs make use of a hidden layer to capture and assess the significance of each input feature about the output (Figure 2.7). This hidden layer retains information about the importance of inputs, eliminating the need for preprocessing data.

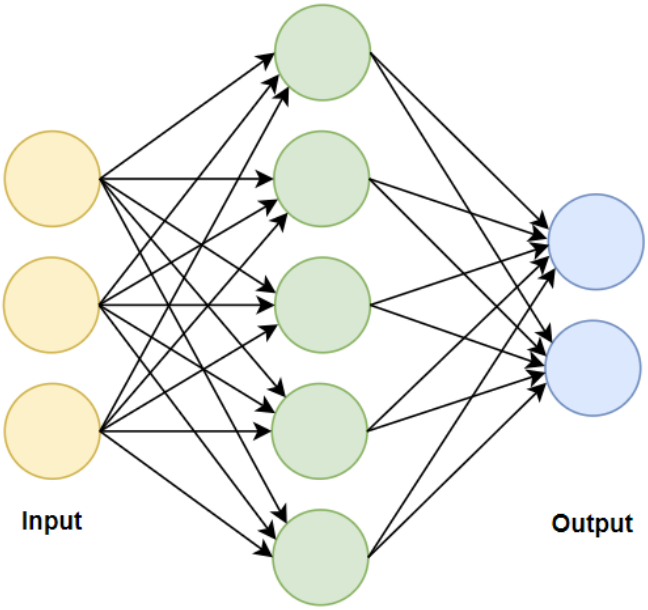


Figure 2.7: An example of an artificial neural network

Deep neural networks (DNNs) leverage the fundamental elements of ANNs. In Figure 2.8, DNNs encompass multiple hidden layers positioned between the input and output layers, hence the term "deep" neural networks. These DNNs empower models to autonomously derive and retain generalizations within these hidden layers.

2.1.6. GENERATIVE ADVERSARIAL NETWORKS

GENERATIVE MODELS

Image generation is the process of creating visual content, such as pictures, illustrations, or graphics, typically using computer algorithms and models. This field has seen significant advancements in recent years, thanks to the rise of deep learning techniques and generative models, particularly Generative Adversarial Networks (GANs) and Variational Autoencoders (VAEs) [11].

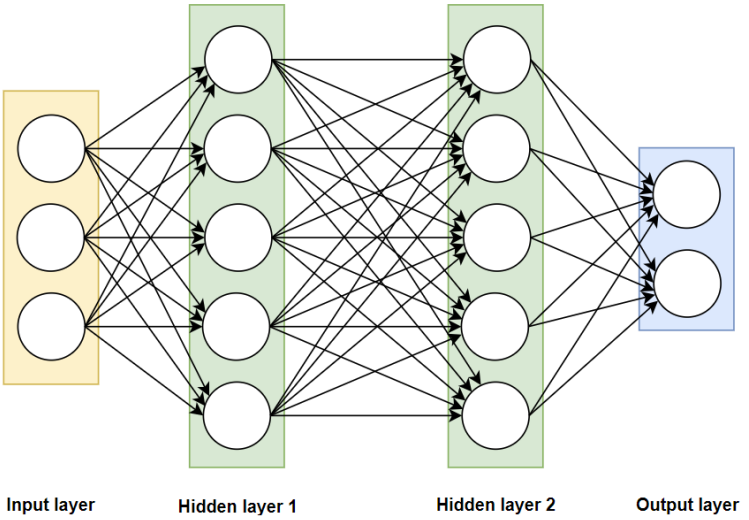


Figure 2.8: An example of deep neural networks

GENERATIVE ADVERSARIAL NETWORKS

A generative adversarial network (GAN) is a class of machine learning framework and a prominent framework for approaching generative AI. GAN is first introduced by Ian Goodfellow and his colleagues in [12]. Generative adversarial networks (GANs) have emerged as a powerful framework for generating realistic and high-quality synthetic data, making them highly valuable in various domains such as image synthesis, natural language processing, and drug discovery. GAN consists of two main components:

- The **generator** learns to generate plausible data. The generated instances become negative training examples for the discriminator.
- The **discriminator** learns to distinguish the generator's fake data from real data. The discriminator penalizes the generator for producing implausible results.

In Figure 2.9, the discriminator network is presented with randomly input samples (random noise), comprising fake examples generated by the generator and real examples from the training set. Initially, before any training commences, the discriminator easily discerns the generator's output.

Since the generator's output directly enters the discriminator for classification, we can employ the backpropagation algorithm across the entire system to adjust the generator's weights based on the discriminator's feedback.

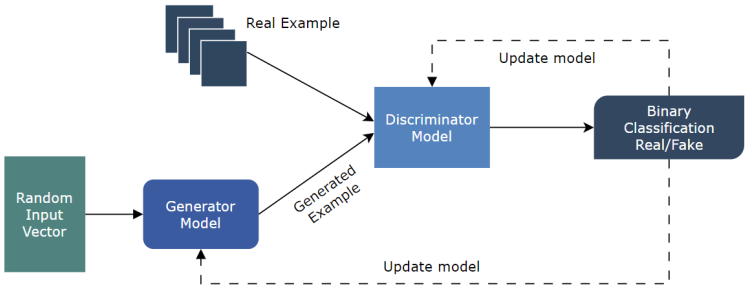


Figure 2.9: Basic network architecture of GAN

As training progresses, the generator's outputs improve in realism, making it increasingly adept at deceiving the discriminator. Ultimately, the generator's creations become so lifelike that the discriminator can no longer differentiate them from authentic examples.

GENERATOR NETWORK

The generator network plays a pivotal role in GANs, a class of machine learning frameworks designed for generative tasks. Its primary objective is to create artificial data that closely resembles real data, whether it's images, text, or any other form of information. The generator achieves this by transforming random noise or latent vectors into meaningful data points.

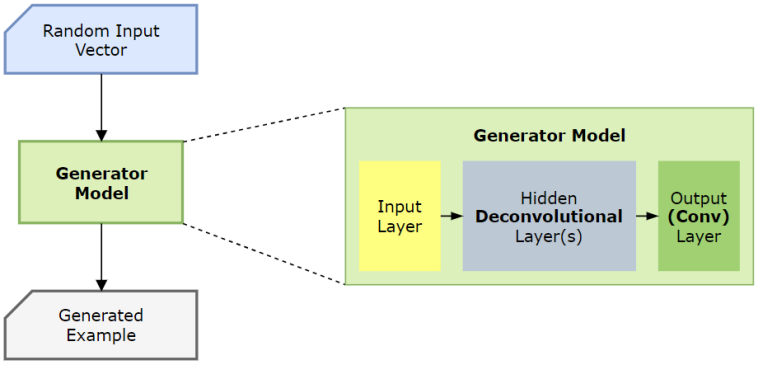


Figure 2.10: Generator Network basic flow

DISCRIMINATOR NETWORK

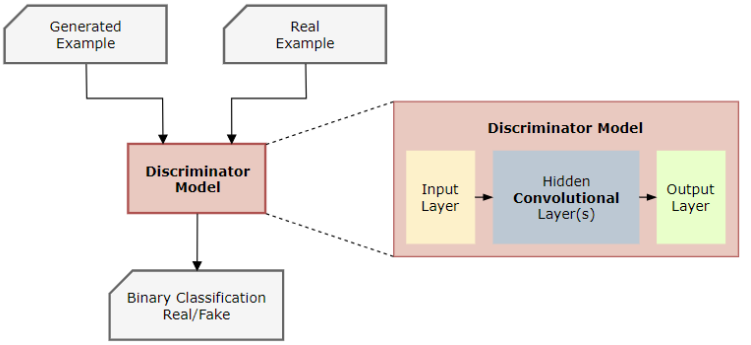


Figure 2.11: Discriminator Network basic flow

TRAINING OVERVIEW

Training GAN comes in a game between two networks, the generator G and the discriminator D. G tries to maximize the loss of D by mapping a noise vector to the input space while D concentrates on maximizing the chance to identify the real distribution of data. For this application, two different datasets MNIST and CELEB-A are used which are typical and popular for image generation applications.

Figure 2.12: MNIST dataset [source]

Figure 2.13: CELEB-A dataset [source]

LOSS FUNCTION

It is known as a cost function or objective function, which is a mathematical function that measures the difference between the predicted values generated by an AI model and the actual target value. During the training process, the model's parameters are adjusted to minimize the value of the loss function. This optimization process helps the model learn and improve.

The generator's objective is to produce images that are indistinguishable from real images. The loss function encourages the generator to produce images the discriminator classifies as real.

The loss function commonly used for the generator is the binary cross-entropy loss:

$$G_{loss} = -\log(D(G(z))) \tag{2.1}$$

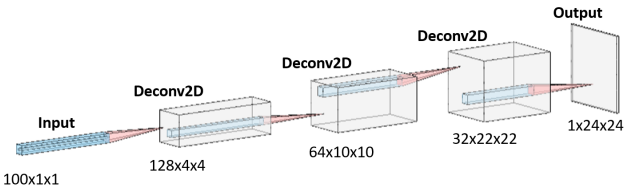
where $G(z)$ represents the generated image, $D(G(z))$ is the discriminator's output when it evaluates the generated image.

The discriminator's objective is to correctly classify real and fake images. It tries to maximize the probability of classifying real images as real and fake images as fake.

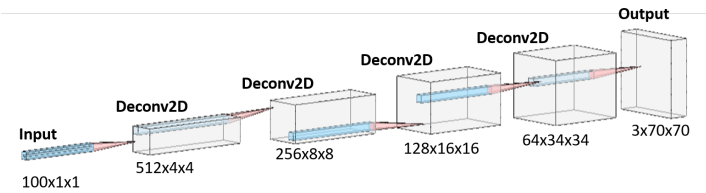
The loss function for the discriminator is also binary cross-entropy loss:

$$D_{loss} = -\log(D(x)) - \log(1 - D(G(z))) \tag{2.2}$$

Here, $D(x)$ represents the discriminator's output when it evaluates a real image (x), and $D(G(z))$ is the discriminator's output when it evaluates a generated image ($G(z)$).

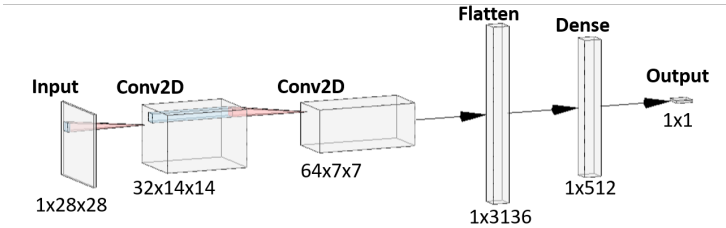


(a) MNIST generator architecture

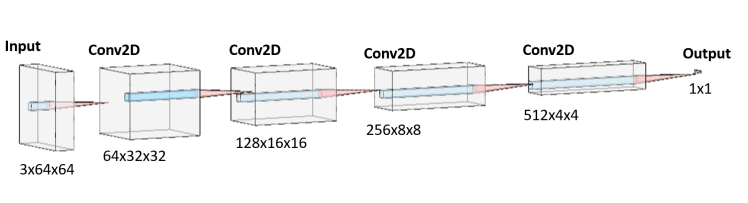


(b) Celeb-A generator architecture

Figure 2.14: Generator architecture for two datasets



(a) MNIST discriminator architecture



(b) Celeb-A discriminator architecture

Figure 2.15: Discriminator architecture for two datasets

COMMON PROBLEMS

Generative Adversarial Networks (GANs) are known to encounter various recurring issues in their training and performance. These problems have garnered significant attention within the research community, as they represent substantial obstacles to achieving optimal GAN performance. While it is important to note that complete resolutions for these challenges have not been achieved, researchers have made concerted efforts to explore potential remedies and workarounds.

One such challenge is **Mode collapse** where GANs tend to produce a limited range of outputs, lacking diversity. This problem occurs when the discriminator becomes too effective at distinguishing real data from generated data, causing the generator to focus on only a subset of the data distribution.

Another prevalent issue in the GAN landscape is **Training instability**. GANs are notorious for their sensitivity to hyperparameters and the potential for training to become unstable, leading to problems like vanishing gradients or oscillatory behavior.

Additionally, **Evaluation complexity** refers to the challenges in assessing model performance. It involves measuring how good the generated data is compared to real data. Traditional metrics like Inception Score, Frechet Inception Distance (FID), or Perceptual Path Length (PPL) can be computationally expensive, particularly when dealing with large datasets or high-dimensional data.

2.1.7. DECONVOLUTION NEURAL NETWORK

DECONVOLUTION CONCEPT

Deconvolution [13], also known as transpose convolution or up-sampling, plays a vital role in deep learning, particularly in convolutional neural networks (CNNs) and tasks related to computer vision. In essence, deconvolution's purpose is to enhance the spatial resolution of an image or feature map. To grasp the concept of deconvolution fully, it's essential to recognize its close connection with convolution as Figure 2.16, as these operations are intricately linked.

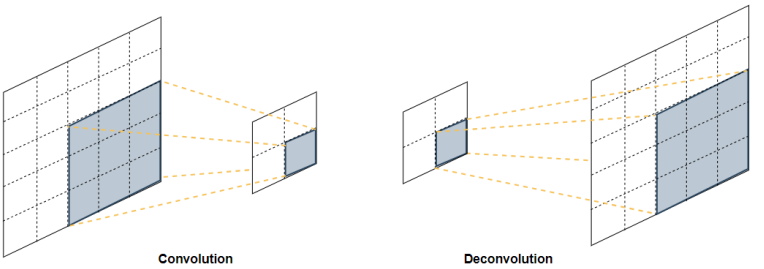


Figure 2.16: Comparison between conventional convolution and deconvolution

DECONVOLUTION ALGORITHM

Parameter	Layer	Description
H		Height of the input feature map
W		Width of the input feature map
H_O		Height of the output feature map
W_O		Width of the output feature map
N_C	Deconv	Number of channels in the input feature map
N_F		Number of filters in the output feature map
k		Height and width of the kernel
s		the stride
p		the amount of zero padding

Table 2.3: A summary of the parameters in the deconvolution layer in GAN

DeConv layer takes feature maps of size N_C x H x W and a group of coefficient matrix of shape N_F x N_C x k x k as inputs, and produces output feature maps of size N_F x H_O x W_O. The input and output size in height and width dimensions are related as follows:

$$H_O = s * (H-1) + k - 2 * p \tag{2.3}$$

$$W_O = s * (W-1) + k - 2 * p \tag{2.4}$$

Algorithm 1 Deconvolution Algorithm of the Generator.

```
procedure DECONVOLUTIONLAYER(I, K)
    Input: input feature map I of shape N_C × H × W
    Input: coefficient matrix K of shape N_F × NC × k × k
    Output: output feature map O of shape N_F × H_O × W_O
    for f = 1 to N_F do
        for c = 1 to N_C do
            O[f] += deconv(I[c], K[f, c])
        end for
    end for
    return O
end procedure
```

Algorithm 1 outlines the DeCONV layer within the generator at a conceptual level, encompassing two nested loops: the filter loop and the channel loop. In each iteration, a single channel ($I[c]$) is extracted from the input with dimensions $H \times W$ and undergoes deconvolution with the corresponding kernel matrix ($K[f,c]$) to generate the output for that particular channel. The outcomes of these individual channel operations are cumulatively aggregated to form one filter of the output ($O[f]$). This iterative process is reiterated N_F times to generate all filters constituting the output feature maps.

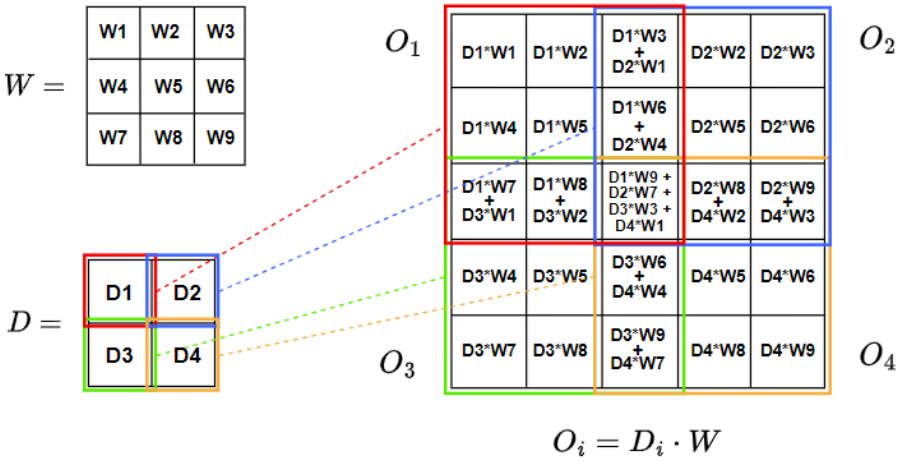


Figure 2.17: Deconvolution computation illustration (stride = 2)

In this report, we propose an FPGA-friendly method by implementing DeCONV with the following four steps: (1) multiply an individual input pixel by the $k * k$ kernel; (2) sum the results of step (1) where the outputs overlap; (3) repeat (1) and (2) for all input pixels.

Comparison: It's critical to highlight that both of these approaches yield identical results for the same input. However, when considering FPGA implementation, the software-based method has notable drawbacks, including:

- The inherent inefficiency of the zero-padding operation in FPGA, which introduces a non-uniform data access pattern when the DeCONV window slides over the zero-inserted input.
- Computational inefficiency arises from performing multiply-accumulate operations on the inserted zeros due to multiplication with zero.

In contrast, the second method is more FPGA-friendly as it avoids zero insertion, leading to improved computational efficiency. Additionally, it is more adaptable and can accommodate various layer configurations. Consequently, we propose an optimized DeCONV architecture based on this method in our work. The primary challenge in the hardware implementation of DeCONV lies in addressing the overlapping sum problem in the outputs.

2.2. RELATED WORK

2.2.1. GENERATIVE ADVERSARIAL NETWORKS APPLICATIONS

GANs have transformed super-resolution (SR), greatly improving the realistic look of zoomed-in images. This game-changing tech allows the creation of high-detail images with incredible realism, benefiting fields like photography and medical imaging. The SRGAN framework, proposed by Ledig et al. (2017) [14], is an exemplary model specifically designed to address the challenge of recovering intricate texture details in images super-resolved at large scaling factors, such as 4x. The framework incorporates a novel perceptual loss function that integrates adversarial loss and content loss components. The adversarial loss aligns the super-resolved images with the manifold of natural images, thereby ensuring the enhanced images look plausible to human perception. On the other hand, the content loss promotes perceptual similarity rather than strict pixel-wise accuracy, enabling the model to emphasize essential details and structures rather than focusing on perfect pixel reconstruction. This dual-loss strategy is instrumental in ensuring that the model captures subtle visual nuances crucial for human perception. As a result, SRGAN has set new benchmarks for perceptual quality, obtaining high Mean Opinion Score (MOS) ratings that closely approximate those of genuine high-resolution images. This

achievement is noteworthy because traditional evaluation metrics such as Peak Signal to Noise Ratio (PSNR) and Structural Similarity Index (SSIM) often fail to capture the nuances in image quality improvements achieved through perceptual loss functions. By prioritizing perceptual similarity, SRGAN has surpassed its predecessors, such as SRCNN and SRResNet, demonstrating the potential of adversarial networks in generating photo-realistic super-resolved images that meet the expectations of a broader range of visual aesthetics in non-medical applications.

The SRGAN model has made substantial strides in the medical domain, particularly in enhancing the resolution of Magnetic Resonance (MR) images. Rewa Sood et al. adapted SRGAN to address the unique challenges encountered in medical imaging [15], specifically focusing on enhancing the resolution of prostate MR images. Utilizing SRGAN on low-resolution MR images has led to reduced scanning times and improved patient comfort, while yielding notable enhancements in in-plane resolution. This application of SRGAN in medical imaging holds considerable promise, as it directly confronts the trade-off between image quality and scanning duration, which is of utmost importance in clinical settings. While SRGAN may not consistently achieve the highest PSNR or SSIM metrics compared to alternative methods like SRCNN, SRResNet, and Sparse Representation, its effectiveness is evidenced by producing images visually closest to high-resolution MR images, as indicated by MOS results. The perceptual quality attained through SRGAN proves pivotal in clinical diagnoses, where finer anatomical details significantly impact detection and treatment planning. This capability underscores the adaptability of GANs in specialized super-resolution tasks and highlights their potential to revolutionize the processing of medical images, ultimately alleviating the burden on healthcare professionals and enhancing patient outcomes.

2.2.2. GENERATIVE ADVERSARIAL NETWORKS ON FPGA

Many technological advancements have been made, resulting in significant practical achievements. While FPGA has yet to match the processing speed and power efficiency of ASIC, it offers distinct advantages, including a shorter time-to-market due to its simpler design process and the flexibility for both static and dynamic reconfiguration. In the realm of artificial intelligence (AI), numerous complex algorithms necessitate specific hyperparameters and configurations tailored to individual applications. Constructing a fixed design to accommodate every algorithm's configuration demands considerable time and effort. Hence, FPGA emerges as a suitable choice for such applications owing to its reconfigurability. Presently, there exists a dedicated community of researchers focused on leveraging FPGA for accelerating AI applications.

Besides FPGA, other hardware platforms for accelerating Generative Adversarial Networks such as Graphics Processing Unit (GPU) or multi-core Central Processing Unit (CPU). All these hardware platforms can compute in parallel and flexibly configure. There hasn't been a significant prior exploration in the realm of Generative Adversarial Networks (GANs), particularly when it comes to utilizing FPGAs, as it's a relatively new area of interest in the research community. Nevertheless, there have been several previous papers focusing on image generation algorithms using hardware acceleration. This section will delve into a similar problem domain, assessing our contributions against prior research in terms of both result quality and speed of acceleration.

In [16], Liu et al. present a novel and highly customized architecture designed to efficiently implement the DeConv method on FPGA hardware, catering specifically to the demands of hardware-accelerated image generation tasks like those found in GANs. They tackle the challenge of overlapping sums within this architecture by introducing additional hardware blocks, which effectively minimize resource usage and latency overhead. Using Verilog HDL, they craft hardware templates that can adapt to diverse DeConv layers within GANs by offering configurable parameters. Additionally, they propose a new tiling method and a memory-efficient architecture to further boost generator acceleration. By storing intermediate data on-chip, they substantially reduce the need for off-chip data transfers, thus significantly optimizing overall performance. Their experimental results demonstrate remarkable acceleration rates, averaging a 58x improvement over CPU-based implementations and a 3.6x enhancement over GPU-based implementations. In terms of power efficiency, their architecture achieves striking improvements, surpassing CPU designs by over 400 times and outperforming NVIDIA Titan X GPUs by factors ranging from 8.3 to 108.

Recent research has introduced FPGA-based accelerators tailored for DeConv networks, as discussed in [17]–[18]. Yazdanbakhsh et al. [17] introduced an FPGA accelerator for GANs, employing both MIMD and SIMD models and separating data retrieval and processing units at a granular level. Additionally, in [18], a design methodology for FPGA-based CNN acceleration targeting image super-resolution algorithms was proposed, integrating multiple Conv (Convolution) layers and a single DeConv (Deconvolution) layer with efficient parallelization techniques. However, it's worth noting that both approaches in [17] and [18] are software-based, leading to computational inefficiencies compared to the hardware-based method discussed in this work, as previously highlighted. Zhang et al. [19] presented a hardware accelerator design utilizing the Vivado HLS tool which aligns closely with our own methodology. Their approach centered on a reverse looping strategy to identify necessary input data for generating the desired output, specifically addressing the challenge of pro-

cessing overlapping sums. However, their technique necessitated the iterative calculation of input pixel positions within loop iterations.

Additionally, loop dimensions expanded to match the output image size, leading to heightened communication overhead with the host processor and increased system latency, ultimately precluding its suitability for real-time applications. In the context of FPGA-based acceleration for very deep Convolutional Neural Networks (CNNs), the challenge of data load and transfer often stands as a primary bottleneck. To mitigate this bottleneck, loop tiling techniques are commonly employed to partition the original workload into smaller segments conducive to efficient hardware execution. Referred to as blocking, this approach entails dividing the input feature maps into discrete sections, enabling their storage in on-chip buffers. This segmentation facilitates data reuse, rendering it especially advantageous for FPGA devices constrained by limited memory resources. Li et al. [20] delved into an optimized tiling approach for convolutions, albeit at the expense of accuracy. This compromise arises from the necessity of replacing boundary pixels within tiles with zeros, a measure taken to mitigate data dependencies among adjacent tiles. Consequently, while this technique addresses certain computational challenges, it may introduce inaccuracies into the results.

In a previous study [21], researchers developed a unified systolic accelerator method for deconvolution tasks. This approach segmented deconvolution into two distinct stages. Firstly, it performed vector-kernel multiplication and stored the intermediate matrices in on-chip memory. Subsequently, it processed the overlaps of these matrices. However, this strategy resulted in heightened on-chip BRAM access and unnecessary data storage, leading to increased power consumption and computation latency. Another research with a novel CNN accelerator architecture is proposed by Lin Bai et al. [22], designed to be scalable and adaptable by integrating convolution and deconvolution functionalities within a unified process element. Unlike conventional approaches, deconvolution is streamlined into a single step, obviating the need for intermediate result buffering. Moreover, the successful implementation of SegNet-Basic on the Xilinx Zynq ZC706 FPGA showcases remarkable performance achievements, with convolution and deconvolution operations achieving throughputs of 151.5 GOPS and 94.3 GOPS, respectively. These results significantly outperform existing segmentation CNN implementations, underscoring the efficacy and potential of the proposed methodology.

Earlier studies predominantly focus on converting deconvolution algorithms into convolutional layers for FPGA implementations. This approach has led to a notable gap in the development of parameterized deconvolutional FPGA implementations that can flexibly adapt to diverse datasets and configurations.

Addressing this need, our work introduces a versatile and fully pipelined FPGA-based architecture. This architecture is specifically designed to bolster the efficiency of deconvolution algorithms, with a particular emphasis on applications involving Generative Adversarial Networks (GANs). By providing a flexible and adaptable solution, our architecture aims to bridge the existing gap in FPGA-based deconvolution implementations and unlock new possibilities for advanced image processing tasks in various domains.

3

OVERALL ARCHITECTURE

The previous chapter shows mandatory theories about Generative Adversarial Networks and their limitations when running inferences on software. Based on knowledge about the algorithm and related research, this section presents an architecture for accelerating GAN on the FPGA platform. This section describes the system architecture with a top-down approach. Moreover, the general design and principle of components are also illustrated here. In addition, these designs are not specific to any FPGA devices.

3.1. GENERAL ARCHITECTURE

The general architecture consists of two main parts: PS and PL. This architecture exploits the parallel computation strength of PL to process the Deconvolution model. Figure 3.1 shows an overall design. This general architecture is inspired from [23].

The PS side is responsible for running software applications including training GAN algorithm and transferring data. Regarding the data transfer, the PL side requires the training weight parameters and input feature map under the 16-bit/8-bit format using the quantization technique. This technique is the solution to reduce the complexity of computations. Moreover, because the PL cannot store the whole statistic model, PS side plays a role in preparing data in DRAM and transferring data to PL via a communication bus - AXI.

Weights, feature input pixels, and feature output pixels are transferred between PS and PL by Direct Memory Access (DMA). Because PS may have other tasks to carry out and it costs many instructions to access memory, the transfer speed will be low if transferring data is PS's task. DMA will be the best choice for transferring data between PS and PL. PS can access the PL controller to man-

age accelerator core functionality through Register Bank, which contains a set of control and status registers.

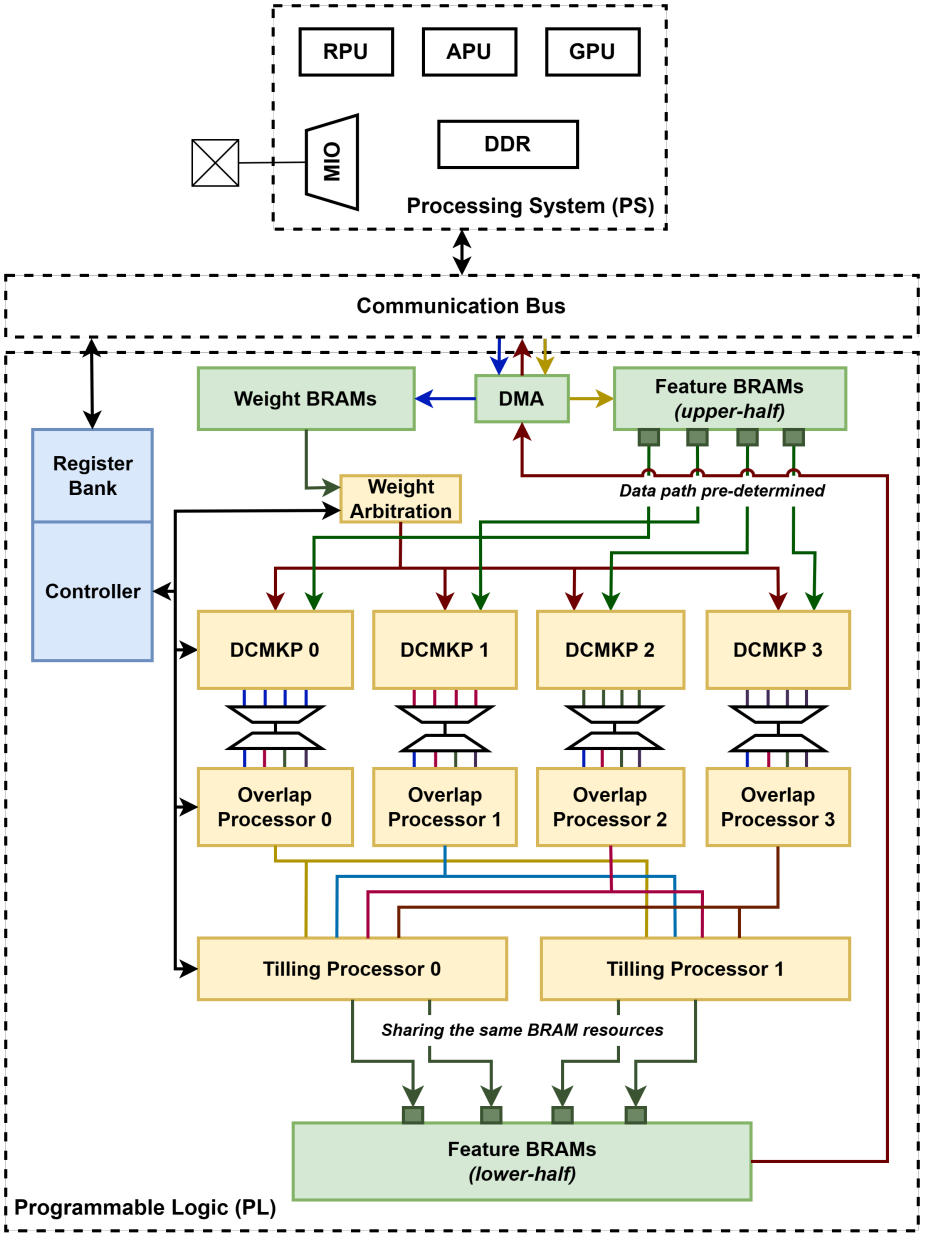


Figure 3.1: General architecture of Generator Network accelerator

As shown in Figure 3.1, when the acceleration core starts to run inference, DMA is responsible for pushing weights and feature input pixels into corresponding *Weight BRAMs* and *Feature BRAMs*. Four BRAMs containing weights are fed into the memory of DCMKPs (Deconvolution Multi-kernel Processors). The *Weight Arbitration* consists of a set of controllers and FIFOs to arrange and load weights in a defined format. *Feature BRAMs* are split into 4 specific parts with pre-defined data paths and each DCMKP has a ping-pong buffer to manage the number of pixels processing in each iteration. After all data is prepared, the PS side will activate a start signal through *Register Bank* to start the whole system to process the data. Weights and feature input pixels will be processed in sequence by DCMKPs. All DCMKPs are synchronous which means the following sets of kernels will be processed if all DCMKPs are finished with current sets of kernels. During the available kernels, DCMKPs will process each column of weights and feature inputs within a single channel. When a feature input column is fed into computing, it needs to remain until all columns of a weight channel are loaded into DCMKPs through *Weight FIFOs*. When internal processing is done, DCMKPs will request the *Weight FIFOs* to load the adjacent weight columns. If it is the last column of a weight channel and the current feature input column is not the last one, *Weight FIFOs* has a feature to loop back to the first column and local buffer of DCMKPs requests the following feature input column to continue the processing mechanism.

If the DCMKPs optimize the running time of the computation by parallel processing four quadrants of feature inputs, the *Overlap Processors* gather the output results of each quadrant to combine to a complete channel. Each *Overlap Processor* stores a set of temporary pixels with the same kernel. The column results of each *Overlap Processor* are divided into 2 parts and fed into *Tilling Processors* to wait for accumulating the channel results of each kernel to produce a complete channel out. The feature outputs are stored in the same BRAMs as feature inputs to save the hardware resources. Then, the acceleration core will send a finish signal to inform the software that the hardware design finished its task. The next subsets of kernels are transferred into the acceleration core to continue the processing.

3.2. DATAFLOW OF DECONVOLUTION MULTI-KERNEL PROCESSOR

GAN model consists of a large amount of data (feature map pixels or filters), therefore, well-managing memory resources is a challenge. Feature maps will be divided into each BRAM with the pre-formatted structure. As the multi-kernel mechanism, Figure 3.2 shows four Weight BRAMs input which stands

for four distinct weights. When the processor starts, the core will get data from BRAMs and store them in the **Input Feature Loader** (ping-pong buffer) and four **Weight FIFOs**.

Ping-pong FIFO is a type of double buffer. It uses two memory banks that improve the efficiency of data transfers. Maximizing the efficiency of two independent processes can be challenging. That's where ping-pong buffering comes in. Instead of waiting for one process to finish before starting the other, ping-pong buffering allows one process to provide output while the other is filled asynchronously. The two banks are switched as required, meaning the output goes back and forth between them, just like a ping-pong ball.

Four FIFOs store different sets of weights. This type of FIFO is specified due to the functionality of the **Deconvolution TOP**. It allows the feature that loops back the available data inside the memory by resetting the read pointer. Those buffers have the same concept of loading a whole pixel column out for each execution.

Feature/Weight Control Unit are the modules that are in charge of controlling the execution of **Weight FIFOs** and **Input Feature Loader**. They contain a set of signals to help the buffers remain stable operation.

Deconvolution TOPx is the main component of the DCMKP processor which is the central processor of deconvolution operations. Each **Deconvolution TOPx** contains multiple DSPs and mainstream traffic so the architecture is complicated. The input data of this component includes the separate **Weight FIFO** and the sharing **Input Feature Loader**. For more details, we will explain in Chapter 4.

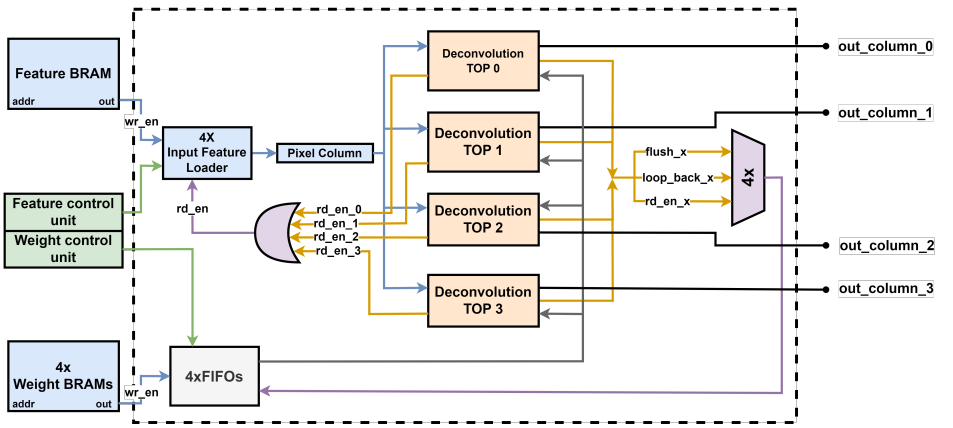


Figure 3.2: Dataflow structure of Deconvolution Multi-kernel Processor

One more thing we need to discuss is about the pixel locations when offloading data into BRAMs. Before allocating memory spaces in SoC, weights and input features are flattened and arranged in the determined order (column to column, channel to channel). Pixel data width is different from Block RAM data width. Therefore, the set of data when mapping into Block RAMs will be illustrated in Figure 3.3 which leads to each address space containing up to four pixels within one Block RAM address. For the Feature BRAMs, the input features are divided into four quarters and stored in each separate BRAM.

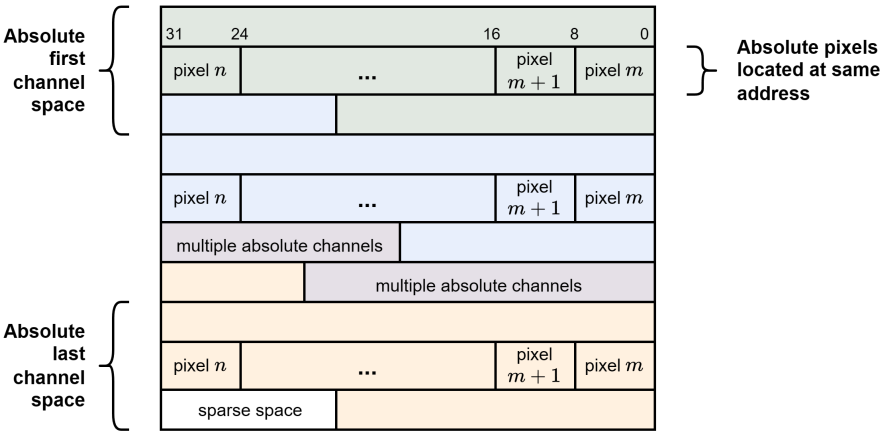


Figure 3.3: Memory structure of Feature BRAMs and Weight BRAMs

Due to the limitation of hardware resources on FPGA, the memory buffers which are Block RAMs are shared between the input feature maps and output feature maps to save the resources and avoid hardware overhead (Figure 3.4).

At first, PS side sends the package of input features to each Block RAM and fills up the upper half of the memory storage space. PL side can access the data domain by calling the corresponding base address. The amount of data PL reads out is dependent on the design requirements and configurations or distributed into multiple internal buffers to increase parallel processing. When the acceleration core finishes its process, the output feature data is written back into the same BRAM by accessing the lower half address spaces.

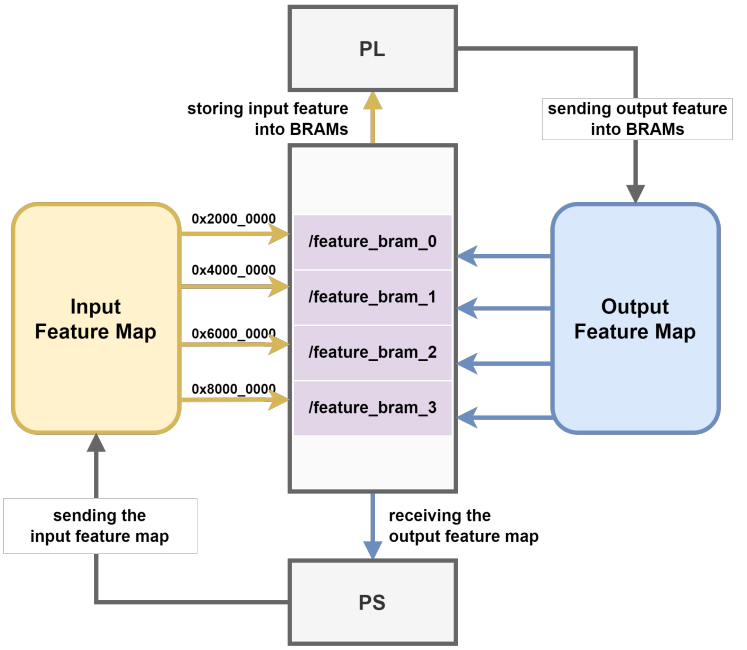


Figure 3.4: Dataflow of input feature and output feature

3.3. OPTIMIZED GAN EXECUTION RESEARCH AND SURVEY

3.3.1. INTRODUCTION

This section presents our focused research and surveys aimed at enhancing the execution efficiency of Generative Adversarial Networks (GANs). Our primary objective is to minimize resource consumption while maintaining high performance and output quality. We detail the methods evaluated and our findings in the subsequent subsections.

3.3.2. OVERFLOW HANDLING IN FIXED-POINT COMPUTATION

CONTEXT AND OBJECTIVE

The transition from 32-bit floating-point to 8-bit fixed-point computation introduces significant challenges, particularly in managing data overflow. This subsection discusses the strategies we evaluated for efficiently handling overflow in hardware implementations of GANs.

IMPLEMENTATION OF OVERFLOW HANDLING

To mitigate data overflow during calculations, we implemented checks after each addition and multiplication operation. If a computational result exceeds the highest representable value in the 8-bit fixed-point format, it is clamped to this maximum limit. This approach ensures computational accuracy is maintained despite the reduced memory footprint.

SOFTWARE SIMULATION AND TESTING

Software simulations were conducted to validate the effectiveness of the overflow handling method before hardware integration. These simulations closely emulated expected hardware conditions, providing insights into the potential impacts and benefits of our approach.

EVALUATION OF RESULTS

The performance of our method was assessed using standard datasets such as MNIST and Celeb-A. The results confirm that our overflow handling strategy allows the GAN model to perform within acceptable limits, thus validating its feasibility for maintaining model accuracy with reduced-precision computing.

Table 3.1: Comparison of GAN model performance with and without overflow handling using MNIST and Celeb-A datasets.

	With Overflow Handling	**Original Software**
MNIST	20.486	12.453
Celeb-A	60.05	48.567

3.3.3. DECONVOLUTION METHODOLOGIES

We compare two prevalent deconvolution methods: **Direct Implementation** and **Zero Padding followed by Convolution**, focusing on their applicability to hardware implementations such as FPGAs.

DIRECT IMPLEMENTATION

This method applies deconvolution filters directly to the input data, enhancing computational efficiency by eliminating unnecessary data manipulation.

This method offers the following advantages:

- **Efficiency in Resource Use:** Avoids unnecessary computational overhead by not manipulating the input data with zero padding.
- **Simplified Memory Access Patterns:** Promotes uniform data access, which is advantageous for systems like FPGAs.

ZERO PADDING FOLLOWED BY CONVOLUTION

This method introduces zeros around the input data's perimeter before applying convolution, simplifying the deconvolution process into a more familiar convolution operation.

This method offers the following advantages:

- **Compatibility with Standard Convolution Operations:** Utilizes existing hardware and software architectures optimized for convolution.
- **Theoretical Simplification of Operations:** Simplifies operations at the cost of some efficiency.

3.3.4. COMPARATIVE ANALYSIS AND CONCLUSION

Both deconvolution methods perform similarly in software; however, Direct Implementation is more efficient for hardware use, particularly in FPGAs, due to its streamlined processing and efficient memory use.

Note on Parameters: The parameters (k, s, H, W, Nc, Nf, Ho,Wo) denote kernel size, stride, input height, input width, number of input channels, number of filters, output height, and output width, detailing the dimensional aspects of the convolutional processes.

Parameter (k,s,H,W,Nc,Nf,Ho,Wo)	Using Zero-padding			Direct Implementation		
	Multi	Add	Total	Multi	Add	Total
(4,2,1,1,100,128,4,4)	0.00328	0.00327	0.00655	0.00328	0.00327	0.00655
(4,2,4,4,128,64,10,10)	0.01311	0.01310	0.02621	0.01311	0.01310	0.02621
(4,2,10,10,64,32,22,22)	0.01586	0.01584	0.0317	0.01586	0.01584	0.0317
(3,1,24,24,32,1,24,24)	0.00017	0.00017	0.00033	0.00017	0.00017	0.00033

Table 3.2: Detailed comparison of computational operations required for zero-padding vs. direct implementation of deconvolution across various parameters.

3.3.5. FINAL CONCLUSION

Our research and surveys have provided substantial insights into optimizing GAN execution. The developed strategies for overflow handling and deconvolution have demonstrated their effectiveness, particularly in their potential for hardware implementation. These methods promise significant advancements in the efficient and effective use of computational resources in deep learning infrastructure.

4

IMPLEMENTATION

4.1. SYSTEM IMPLEMENTATION

In Figure 3.1, the system is described and not specific to any SoC or FPGA device. For this thesis, the system is implemented on Xilinx SoCs, so several implementations are only compatible with Xilinx devices. The component which is heavily device-dependent is the communication bus. AXI interface protocol is used for Xilinx SoCs to implement the bridge between PS and PL. Xilinx AXI is a bus protocol for on-chip communication between IP cores in FPGAs and SoCs. It provides a standard interface for high-performance data transfer between devices while minimizing interconnects. The protocol is widely used in Xilinx FPGAs and SoCs, as well as in many other digital systems. Besides the AXI interface, the system also needs several IPs to support the communication, as shown in Figure 4.1

4.1.1. PROCESSING SYSTEM

The ZYNQ Processing System is a pre-designed and pre-verified block of intellectual property that comprises the ZYNQ Processor System design. This IP can be incorporated into a larger system-on-chip (SoC) design, allowing designers to harness the processing power and flexibility of the ZYNQ platform without having to create the system from scratch. The ZYNQ Processor System IP includes the dual-core ARM Cortex-A9 processor, Xilinx programmable logic, on-chip peripherals, and interfaces, making it possible for designers to personalize the platform to meet their unique requirements.

Figure 4.2 shows the interface of this IP. First, *ZYNQ UltraScale+* provides a system reset port by *pl_clk0* and connects with an IP *Processor System Reset* to activate different reset ports (active high or active low are possible) for

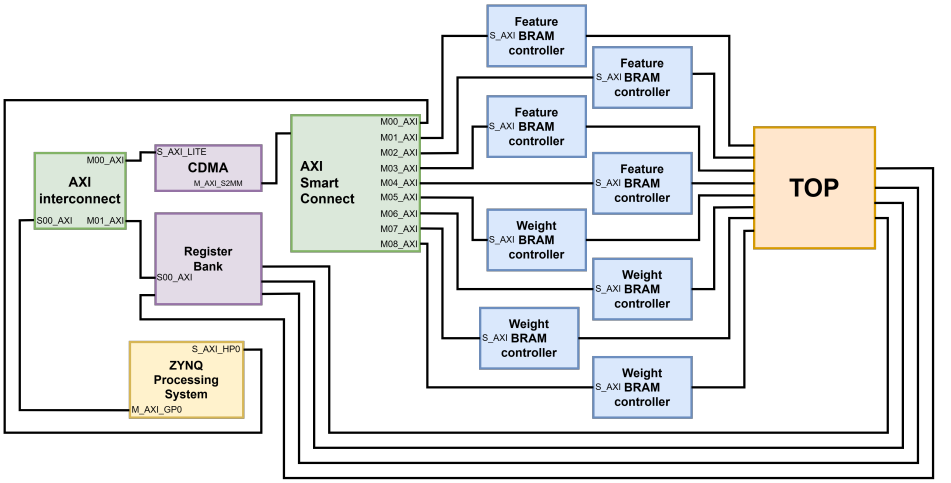


Figure 4.1: System implementation on Xilinx SoC

further designs. This IP supports two ports for communication between PS and PL: *S_AXI_HP0_FPD* and *M_AXI_HPM0_FPD*. *S_AXI_HPx_FPD* is a high-performance full-power domain port supporting connecting PL and DRAM. It provides high-bandwidth access to external memory and peripherals. For this system, *S_AXI_HPMx_FPD* is used to map memory space (weights, feature input maps and feature output maps) between PS DRAM and PL memory. *M_AXI_HPMx* is a high-performance interface that can be used to access large amounts of data from the PS to PL. For this thesis, *M_AXI_HPMx* is used for two tasks: controlling the CDMA IP and accessing the register bank to control and config the acceleration core.

Figure 4.2: System implementation on Xilinx SoC

4.1.2. PS-PL COMMUNICATION

This thesis uses two concepts for the communication between PS and PL: direct AXI-Lite interface and DMA. Both methods require *AXI Interconnect* IP. This IP has a function to connect one or more AXI memory-mapped Master devices to one or more memory-mapped Slave devices. For direct AXI, PS uses this method to access the register bank of the system by *M_AXI_HPMx* port of *ZYNQ UltraScale+ Processing System* IP via *AXI Interconnect*. In this case, the AXI master is PS, and the AXI slave is the register bank.

For DMA, this method is used for reading and writing a large amount of data between PS and PL. In this system, PS accesses the BRAM memory of the PL in Feature BRAMs and Weight BRAMs. The core IP of this method is CDMA IP, which stands for Central Direct Access Memory [24], as shown in Figure 4.3. CDMA allows for high-bandwidth data transfer between a memory-mapped source address and a memory-mapped destination address by utilizing the AXI4 protocol. It provides Direct Memory Access (DMA), enabling efficient and speedy data transfer without CPU intervention. Therefore, CDMA is suitable for mapping data between two memories, DRAM and BRAM. To invoke CDMA, it needs to be connected to *S_AXI_HPMx_FPD* port of *ZYNQ UltraScale+ Processing System* to be able to access PS DRAM. However, CDMA uses the AXI interface, while Block Memory Generators (BMG) do not. Therefore, CDMA also needs *BRAM controller* module, which acts as an interface bridge between AXI and BRAM interface. If there are multiple memory spaces, those spaces need a *AXI_SmartConnect* IP, which acts as a connector between a CDMA and multiple BRAM controllers. To maximize the transfer efficiency of CDMA, address spaces of memory in Feature BRAMs and Weight BRAMs are organized continuously as shown in Table 4.1. Hence, CDMA can transfer all feature maps and weights from PS to PL in a single transfer. This organization increases transfer speed due to the experiment in chapter 5

4.1.3. ACCELERATION CORE - TOP

The *TOP* component in Figure 4.1 is the acceleration core which contains Register Bank, Weight BRAMs, Feature BRAMs, Weight Arbitration, DCMKPs, Overlap Processors and Tilling Processors as shown in Figure 3.1. Design in TOP does not rely on any Xilinx SoCs. Therefore, the system is effortless to be implemented on different kinds of Xilinx SoCs without modifying the *TOP* components. All components within the *TOP* module are written in pure code and do not require Vivado Block Design for instantiation. Vivado Block Design, a feature of Vivado software, allows designers to connect components easily using a graphical user interface (GUI), especially for components with complex I/O interfaces like AXI. However, designing with Block Design can make ver-

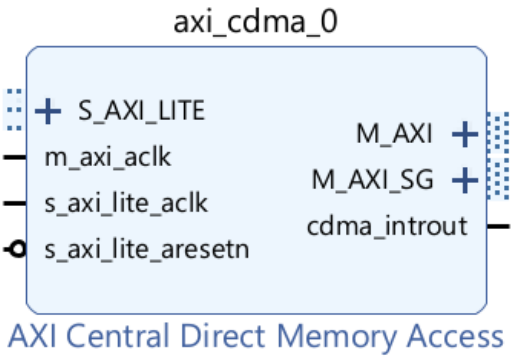


Figure 4.3: Central Direct Access Memory IP on ZYNQ UltraScale+ Processing System

Table 4.1: Address Mapping of PL memories

Memory Segment	Master Base Address	Range	Master High Address
Feature_BRAM_0	0xE000_0000	32K	0xE000_7FFF
Feature_BRAM_1	0xE000_8000	32K	0xE000_FFFF
Feature_BRAM_2	0xE001_0000	32K	0xE001_7FFF
Feature_BRAM_3	0xE001_8000	32K	0xE001_FFFF
Weight_BRAM_0	0xE002_0000	32K	0xE002_7FFF
Weight_BRAM_1	0xE002_8000	32K	0xE002_FFFF
Weight_BRAM_2	0xE003_0000	32K	0xE003_7FFF
Weight_BRAM_3	0xE002_8000	32K	0xE003_FFFF

sion control challenging. Several IPs, such as the *ZYNQ UltraScale+ Processing System* and *BRAM controller*, still require Block Design instantiation. Consequently, components not needing Block Design will be packaged in the *TOP* module.

4.2. DECONVOLUTION MULTI-KERNEL PROCESSOR IMPLEMENTATION

4.2.1. SIGNAL DESCRIPTIONS

Figure 4.4 describes the interface of a single DCMKP including input ports. This block diagram also shows the connection between DCMKP and other interfaces.

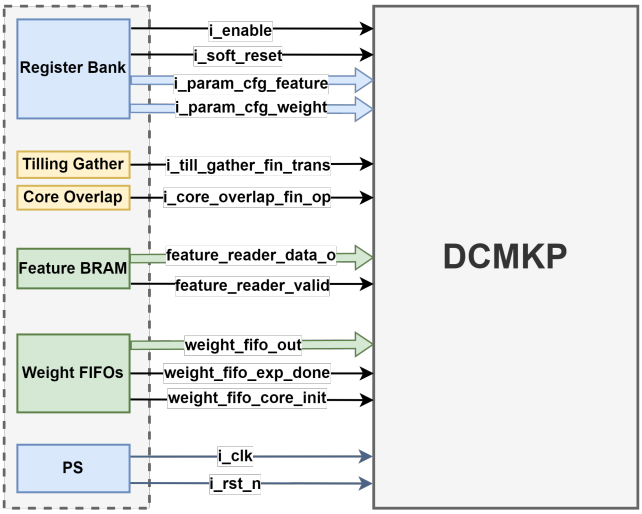


Figure 4.4: Input Interface of DCMKP

Figure 4.5 describes the interface of a single DCMKP including output ports. This block diagram also shows the connection between DCMKP and other interfaces.

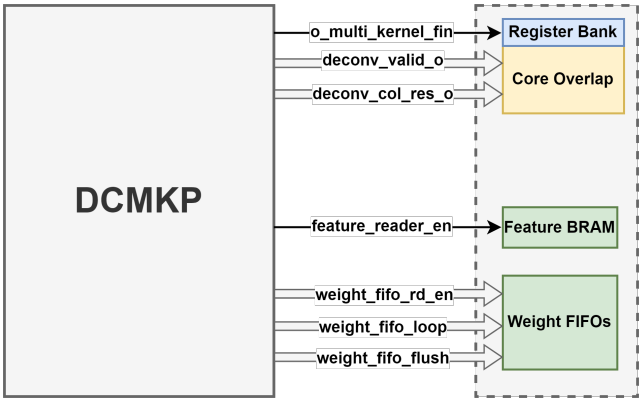


Figure 4.5: Output Interface of DCMKP

Table 4.2 describes the list of I/O pins of DCMKP interface.

Table 4.2: List of I/O pins of DCMKP

Port name	Direction	Bit Width	Description
i_clk	In	1	The processing clock signal
i_rst_n	In	1	Synchronous reset. Active low
i_param_cfg_feature	In	[31:0]	Input feature configuration
i_param_cfg_weight	In	[31:0]	Weight configuration
i_till_gather_fin_trans	In	1	Tilling Gather finish signal
i_core_overlap_fin_op	In	1	Core Overlap finish signal
feature_reader_data_o	In	[31:0]	Output Data of Feature Reader
feature_reader_valid	In	1	Valid Signal of Feature Reader
weight_fifo_out	In	[127:0]	Output Data of Weight FIFO
weight_fifo_exp_done	In	1	Export Done Signal of Weight FIFO
weight_fifo_core_init	In	1	Init Signal of Weight FIFO
o_multi_kernel_fin	Out	1	Output Finish Signal of DCMKP
deconv_valid_o	Out	[3:0]	Valid Signals of DCMKP
deconv_col_res_o	Out	[2303:0]	Output Data of DCMKP
feature_reader_en	Out	1	Feature Reader Enable of DCMKP
weight_fifo_rd_en	Out	[3:0]	Weight FIFO Reader Enable of DCMKP
weight_fifo_loop	Out	[3:0]	Weight FIFO Loop Enable of DCMKP
weight_fifo_flush	Out	[3:0]	Weight FIFO Flush Enable of DCMKP

4.2.2. FUNCTIONAL DESCRIPTIONS

DCMKP is a core processing component of the acceleration core. It fetches a couple of data including four sets of kernels and a quarter of channel feature maps to obtain deconvolution operation. Filtering the feature maps with the whole sets of kernels costs significant execution time of the system. Therefore, the throughput of DCMKP heavily affects the system performance.

In this design, the processing speed and hardware resources are considered and balanced as shown in Figure 3.1. Each DCMKP contains four single cores responsible for each kernel to speed up four times the processing speed instead of serially processing each set of kernels. Each core hardware structure is described in Figure 4.6. Firstly, when the whole system receives an enable signal, the core waits for the *Weight FIFO* and *Input feature map buffer* to fetch data into the core to start the processing phase. The data format is under a single column, meaning the *Weight FIFO* and *Input feature map buffer* will extract a column from left to right and pass them through the single core. Then, the fully-pipeline architecture of DCMKP is applied for memory optimization and hardware resource saving. Each feature pixel from up to bottom within a column is serially multiplied with a weight column and then continuously passes through the *Row-overlap accumulating* without waiting for the previous stage to finish. As explained above, the *Row-overlap accumulating* stage describes the

processing between overlapping pixels within consecutive temporary multiplication output data. The overlapping region appears due to the stride value in convolutional operation. When this module asserts an *accum_fin*, the DCMKP core will request an adjacent weight column to the core. Additionally, when this module returns *valid_i* HIGH, the corresponding output is passed through *Column-overlap accumulating*.

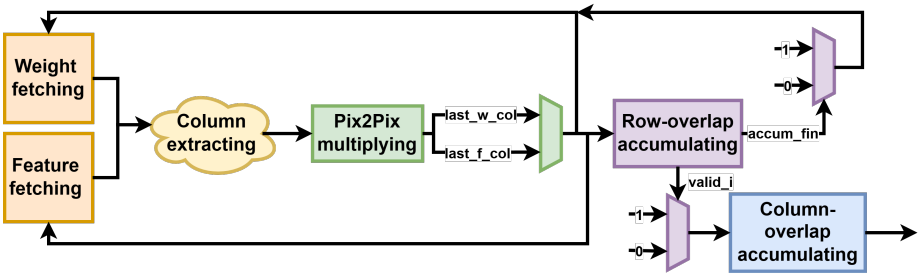


Figure 4.6: General design of Deconvolution Multi-kernel Processor

A detail of a fully-pipeline DCMKP architecture is shown in Figure 4.7. The list below shows several simple descriptions of the Figure:

- Blue components and signals are related to memory access and storage.
- Green components and signals are related to control signals and configuration signals
- Orange components and signals are related to DSPs or logic operators.
- Rectangular boxes are components with latency because of their internal registers or these can be called sequential circuits.
- Cloud component is a combinational circuit.

The architecture of DCMKP is designed in four main stages which are followed by the architecture of CPU architecture pipelining stages: *STAGE FETCH/UPDATE, STAGE MEM ACCESS, STAGE DECODE, STAGE EXECUTE/UPDATE*. Additionally, these steps are not the same as pipeline stages. The number of pipeline stages depends on how the process is divided into sections. However, these stages help to explain the working principles of the design. Functionalities of these stages are described below.

In ***STAGE FETCH/UPDATE***, data stored in *Weight FIFO* and *Feature Buffer* are accessed and selected to fetch into the registers. Initially, before the whole

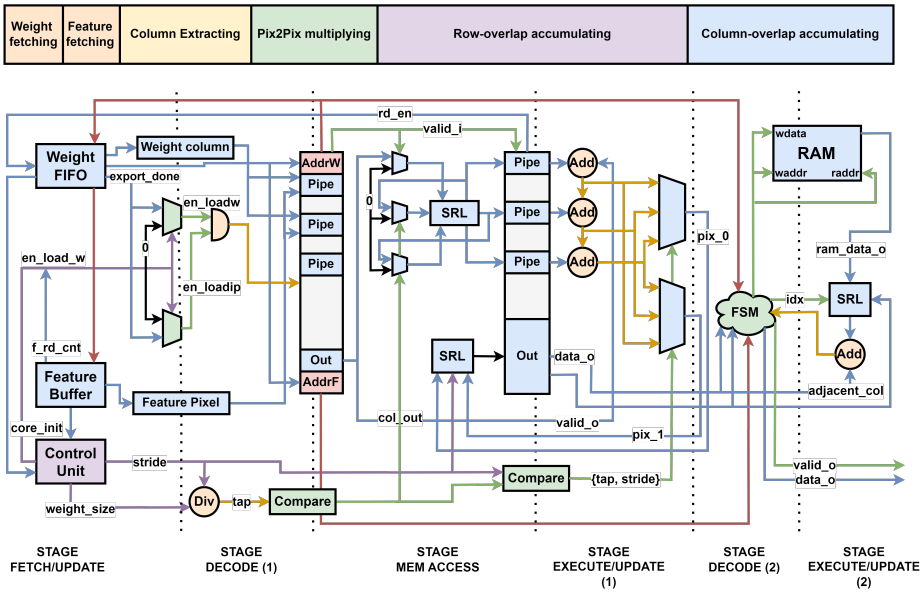


Figure 4.7: Deconvolution Multi-kernel Processor Implementation

system operation, *Control Unit* will get configurations from *Register Bank* to set the conditions for the core dynamically. These conditions affect the number of valid pixels in the *Weight column* and also the output from *Feature Buffer*. The purpose of those configurations is to adapt to multiple sizes of layers. In *Weight FIFO*, RAM stores multiple pixels. The mechanism of this type of FIFO is loading each pixel and storing intermediate results in a temporary register by continuously shifting left until it has enough pixels for each column. *Weight FIFO* also returns a signal called *export_done* which means the weight data is valid and available to enter the core. On the other hand, for the *Feature Buffer*, the output data is updated iteratively after all the weight columns are popped out from *Weight FIFO*; otherwise; the current output data remains unchanged. For the new feature column data, *Weight FIFO* resets its read pointer but not its memory which means the weight column out is looped back from the first order.

In ***STAGE DECODE(1)***, the leftmost of Figure 4.7 shows that there are multiplexers (muxes) to enable the flip-flops between internal registers and multiplier computations. The upper mux is enabled (*en_loadw*) when *Weight FIFO* export valid data (*export_done*). On the other hand, the lower mux enables (*en_loadw*) a feature column and this mux depends on the upper mux as well.

When two enable signals are HIGH, the multiplier IP is invoked to start the operation and two enable signals are forced to raise LOW. This concept is inspired by APB write transaction (APB stands for *Advanced Peripheral Bus* [25] protocol which enables the efficient data transfer within SoC in a simple way). At this time, one or two data ports of DCKMP (weight column and feature column respectively) are fed from *STAGE FETCH/UPDATE* due to the specific mechanism as explained above. In addition, to reduce the hardware overhead, the feature data input port is processed with each pixel serially.

In ***STAGE MEM ACCESS***, a chain of registers is connected together to create a pipeline sequence which helps improve the data throughput and resource saving when data is fetched continuously without waiting for the previous task to finish. In general, this stage consists of two core components in Figure 4.6, one is *Pix2Pix multiplying*, the other is *Row-overlap accumulating*. When data is ready in ***STAGE DECODE(1)***, it takes a latency of 3 clock cycles with 2 registers (before and after the Multiplier IP) and one internal pipeline stage Multiplier IPs settings to get the data out within a weight column and each pixel of a feature column. The data out is then simultaneously synchronized into the pipeline registers of *Row-overlap accumulating* module. In this module, the number of internal registers is 3; however; input data does not require to pass through all of those registers but depends on the determined configuration transferred by the *Register bank* 3.1 which is *Stride parameter* which is one of the full packages of configuration parameters transferred from PS to PL. The data path is controlled by three muxes in the center of the block diagram. Each mux is connected with separate registers to control which register is enabled. During each clock cycle since a valid signal is asserted, the data by-pass is shifted left *SRL* one or two pixels due to *Stride parameter* then stored in a register before fetching into the next stage. See Figure 4.8 for more details.

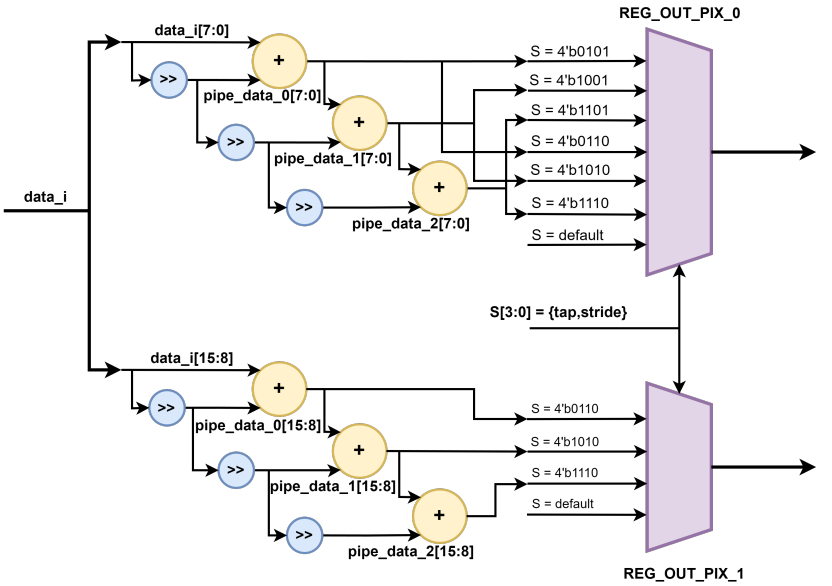


Figure 4.8: Row-overlap accumulating processor block diagram

In particular, the input data has at least one pipeline stage to get the serial output pixel data. Based on the configuration, the input data can go through one or multiple flip-flops and shift registers before feeding into an adder with input data. If *stride* is equal to 1, *reg_out_pix_0* has valid data and *reg_out_pix_1* remains 0. Otherwise, *stride* is 2 and these pixels are stored in temporary output data until the valid signal is asserted.

In ***STAGE EXECUTE/UPDATE(1)***, one signal (*rd_en*) feeds back to *Weight FIFO* module to request data. This signal plays a role of balancing the data arrival time and processing speed of DCMKP. At this stage, output data must wait a couple of clock cycles since each pixel is updated serially. Each output pixel is the addition between the last significant pixel of the registers and input data. If the stride value equals 1, there is only one adder to get an output pixel. However, if the stride value is 2, there is another adder with an addition between the second last significant pixel of registers and input data. The number of registers added depends on two signals (*tap* and *stride*) to select the correct sum value. The below block diagram 4.9 explains more about the concept of this module.

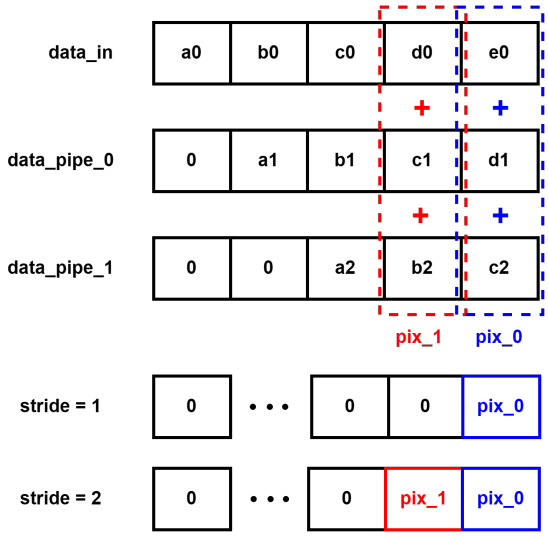


Figure 4.9: Block diagram for adding each pixel of registers and input data

In ***STAGE DECODE(2)***, the main component is FSM (Finite state machine) which is the main part of *Column-overlap accumulating* module in Figure 4.6. The purpose of this module is to analyze the overlapped pixels and accumulate those between adjacent output column data from *Row-overlap accumulating module*.

In Figure 4.14, there are four states which are *INIT, WRITE, WAIT, READ*. In more details,

- **INIT:** This state is the default state of FSM. In this state, the data is synchronously written into RAM and then read out in the next clock cycle in Figure 4.10. The number of data written into RAM depends on stride value. If the stride value equals 1, only one data is stored in the memory. Otherwise, the stride value equals 2, data is written and read twice alternatively. When RAM reads out enough data, *done_init* is asserted to change to the next state. This state is only operated once for each input feature channel execution. When DCMKP is fed the next input channel, *release_fsm* signal is asserted to jump to state **INIT**. See Figure 4.14 for more details.

- **WRITE:** This state simply controls the data to write into the memory when *valid_i* is HIGH with the valid write pointer. In Figure 4.11, the *is last weight column* flag indicates that the last column of the weight channel

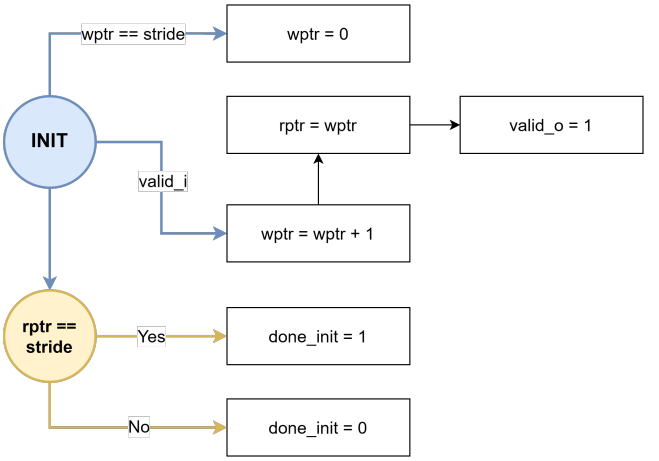


Figure 4.10: Block diagram of INIT state

filters the current data. The *first feature column* indicates that the current data is the dot product of the random weight column with the first feature column per channel (One iteration of the weight column corresponds with the next execution of the feature column). If these two flags are asserted, write pointer resets 0. Otherwise, if the previous state before **WRITE** is **WAIT** or **READ**, the write pointer is updated due to the current read pointer. These conditions are applied before the next *valid_i* is met.

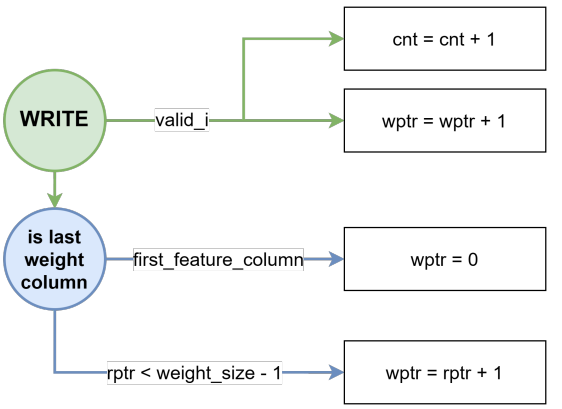


Figure 4.11: Block diagram of WRITE state

- **WAIT:** This state 4.12 is designed to recognize the position of overlapped pixels. To address this issue, the write pointer needs to be set to the current read pointer remainder (*Read Pointer % Weight Size*) if the current state is not **INIT, WAIT**. When it is in **WAIT** state, the write point is incremented as state **WRITE** when *valid_i* is HIGH; however; the incoming data and the data in memory with the same write pointer are accumulated. To reduce hardware overhead, we divide this accumulation step into several clock cycles. It means that data accumulating is processed with each couple of pixels within a clock cycle.

  There is a parameter **OUTPUT_SIZE** which indicates for the output data width of this module. The formula of this parameter is:

$$OUTPUT\_SIZE = STRIDE * \left(\frac{FEATURE\_SIZE}{2} - 1\right) + WEIGHT\_SIZE \quad (4.1)$$

  where *STRIDE* and *WEIGHT_SIZE* are the part-select [31:30] and [9:0] value of *Weight parameter register* respectively and *FEATURE_SIZE* is the part-select [9:0] value of *Input parameter register* in component *Register Bank* of Figure 3.1

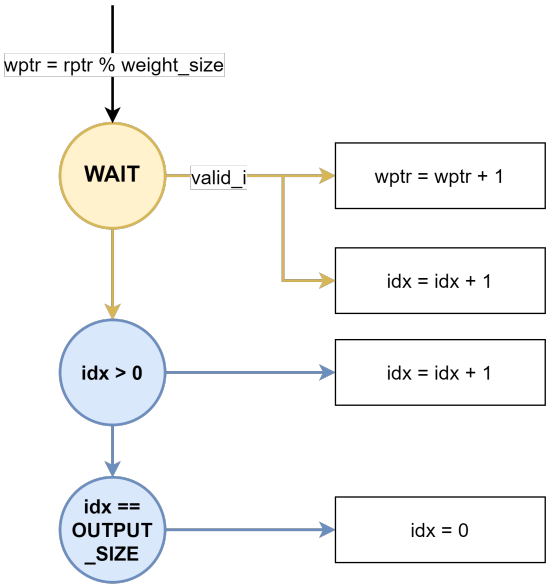


Figure 4.12: Block diagram of WAIT state

- **READ:** This state 4.13 simply reads the data out when it is finished to accumulate. It also raises a *valid_o* within a clock cycle to the next processor

to start operation.

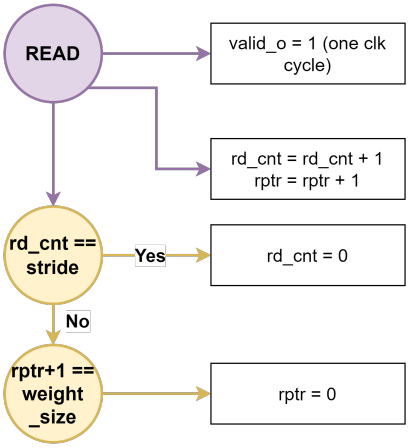


Figure 4.13: Block diagram of READ state

Figure 4.14 below shows the general design of component *Column-Overlap Accumulating Processor* with 4 distinct states.

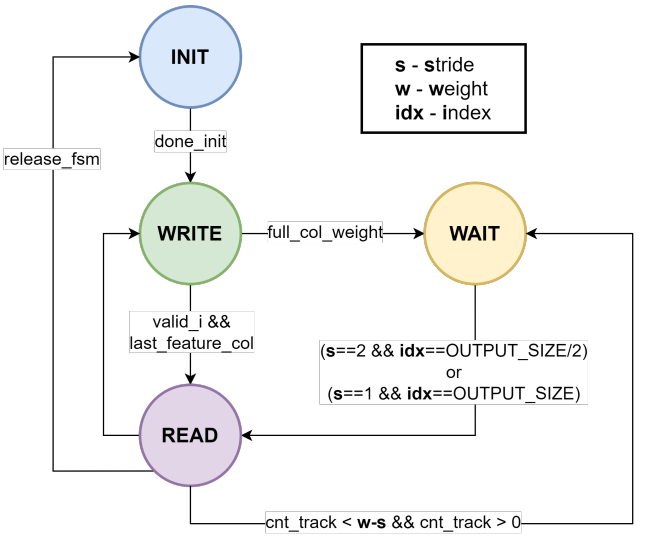


Figure 4.14: Column-overlap accumulator FSM

In ***STAGE EXECUTE/UPDATE(2)***, the *Column-Overlap Accumulating Processor* asserts relative signals and data to execute the following core with is *Core Ovelap Processor* and releases the Finite State Machine.

4.3. CORE OVERLAP PROCESSOR IMPLEMENTATION

The deconvolution process within DCMKP is executed in a column-wise manner. This specific architectural design describes that each individual DCMKP outputs data in a single-column format, representing a quarter segment of the entire input data. As detailed in section 4.2, every DCMKP is directly connected to its designated *Feature Block RAM* (Feature BRAM) and four *Weight Block RAM* (Weight BRAM) units. These essential components are integral in facilitating a three-dimensional deconvolution operation, which is precisely tailored to the quarter segment of input features allocated to each DCMKP, using four distinct sets of weights.

As a result of this configuration, each DCMKP is capable of generating four sets of output data, each corresponding to one of the unique weight sets. This leads to the simultaneous generation of 16 output data, leveraging the inherent parallel processing capabilities of the architecture. Due to this arrangement, the further processing requires a specific processor to manage the pixel overlaps among the four quarter segments associated with each weight set. To address this challenge, an additional four cores, known as the **Core Overlap Processor**, are instituted to handle these tasks with efficiency and responsiveness. Within this system, each core is assigned the critical role of identifying and managing the overlap areas between the input data segments. After processing, each core then exports the validated output data, which is then compiled on the intermediate channel output associated with each weight set.

Figure 4.15 illustrates the specific overlapping areas processed by the Core Overlap Processor. The diagram highlights that angle subpart 0 overlaps with angle [2] in the yellow-marked row, and similarly, angle [1] overlaps with angle [3], indicating these specific rows as the areas of overlap. Additionally, there is a noticeable column overlap, as demonstrated in the blue section of the figure. In the processing priority, overlapped rows are given precedence over overlapped columns, a decision whose rationale is further discussed in next sections.

The post-processing architecture utilizes adders for executing overlap operations. Besides eight Xilinx Adder IPs shown in Figure 4.16, symbolized by circles with a plus sign, arithmetic operations can also be synthesized to **Digital Signal Processing** (DSP) units using a user-defined macro. Furthermore, a dual-port RAM serves as a critical buffer to maintain the overlapped column data between DCMKP1 and DCMKP3 for subsequent processing of column overlaps. The design of the processor includes two identical sub-block architec-

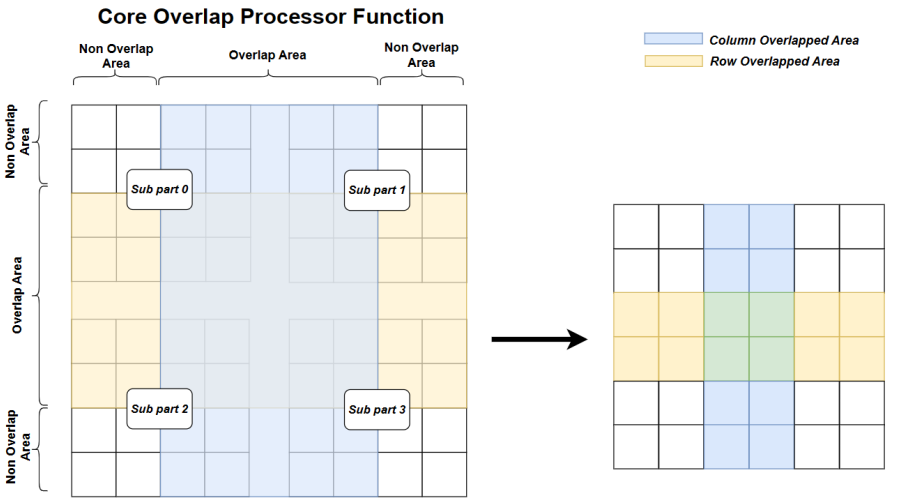


Figure 4.15: Block diagram to illustrate the Core Overlap Processor Function

tures, each equipped with four adders, two multiplexers (mux), one Serial-In Parallel-Out (SIPO), and two shift registers. These sub-blocks are strategically deployed to manage row overlap processing between the results from DCMKP0 and DCMKP2, and between DCMKP1 and DCMKP2, all conducted in parallel. This configuration is designed to optimize processor utilization and minimize processing time. Additionally, the same sub-block setup is used for column overlap processing, enhancing resource efficiency and contributing to overall power conservation.

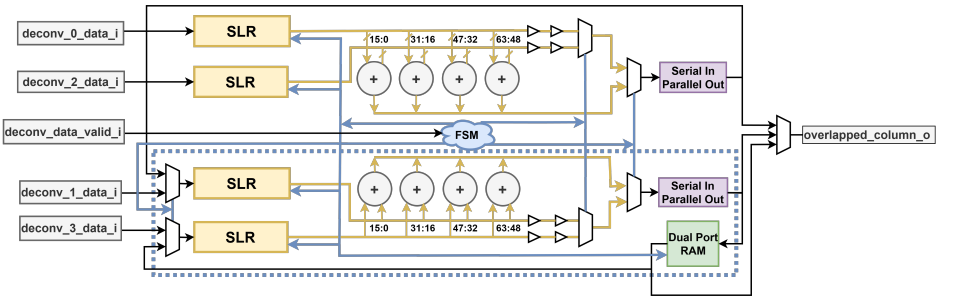


Figure 4.16: Core Overlap Processor Implementation

During each processing cycle, the processor handles four continuous 16-bit pixels, storing the resulting data in a Serial In and Parallel Out (SIPO) buffer. Once all pixels are processed, the overlapped column is validated and then either extracted for output or saved to local memory for further calculations. This management strategy ensures efficient handling of data and minimizes delays in output delivery. The processor's operations are governed by a finite state machine that oversees all activities, including when to save or retrieve data to or from local memory, the selection of data paths for calculations at each multiplexer, and the validation of the final output. There are two interdependent sub-state machines within this system, enhancing the processor's ability to manage complex data paths and ensure accurate processing results.

1. **Row Overlap Processing:** The processor operates in four main states, as present in 4.17:

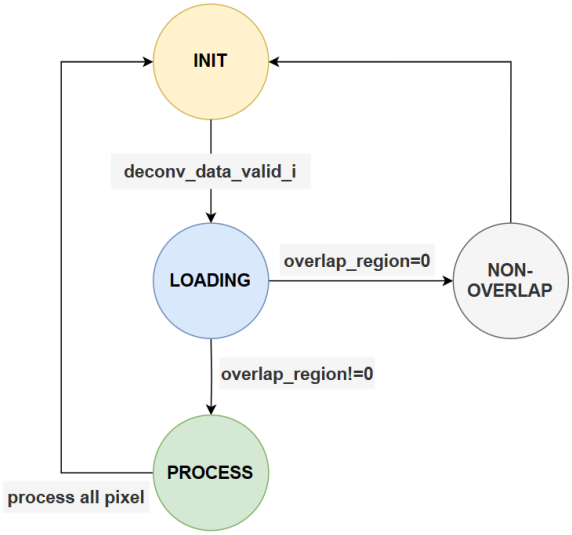


Figure 4.17: Row overlap process state machine

- **INIT:** The initial state where the processor waits for a valid signal indicating that the data from the DCMKP is ready. Upon receiving this signal, the processor transitions to either the LOADING or NON_OVERLAP state.
- **LOADING:** In this state, the processor samples dynamic parameters and stores them in internal memory. It also loads deconvolution

results from the DCMKP into a shift register, preparing it for the next operation.

- **PROCESS:** Here, the processor checks four pixels each cycle based on the dynamic parameters and decides whether to proceed with the operation for each pixel. After processing is complete, it selects the necessary data path for output and returns to the INIT state.
- **NON_OVERLAP:** No operation occurs; the processor merely selects the data path for output.

2. **Column Overlap Processing:** Similar to row processing which shown in 4.18, it includes:

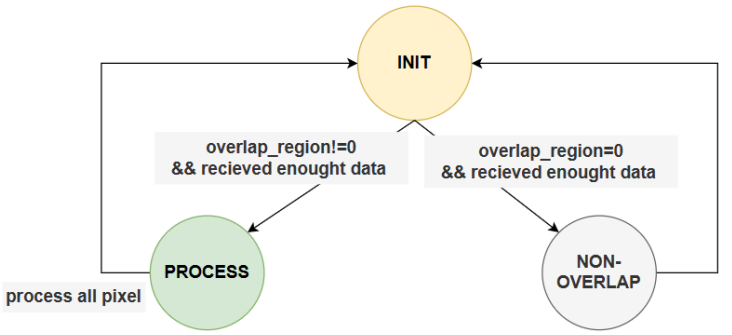


Figure 4.18: Column overlap process state machine

- **INIT:** A waiting state where the processor begins operations only upon receiving sufficient data. In the waiting time, data in the dual RAM is prefetch the upcoming process .
- **PROCESS:** The main operational state where the processor uses the same hardware setup as row processing to manage data. Upon completion, it returns to the INIT state.
- **NON_OVERLAP:** If no overlapping operation is required, the processor transitions to this state, effectively skipping the overlap process.

This approach not only enhances the efficiency of the processor but also optimizes the overlap management in both row and column processing, ensuring streamlined and error-free operations.

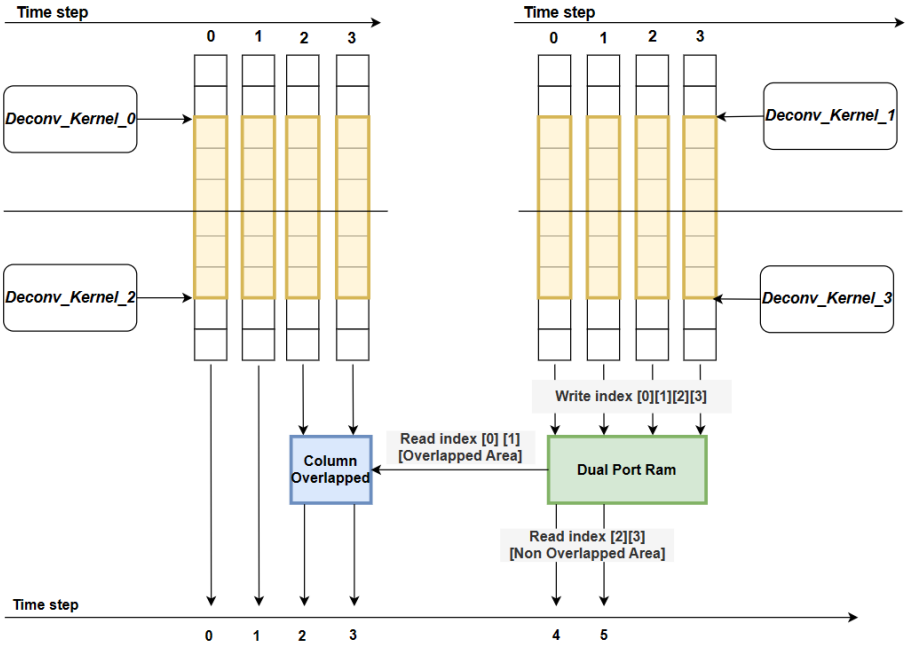


Figure 4.19: Core Overlap Processor Timeline Diagram

To understand the processor's functionality comprehensively, Figure 4.19 provides a detailed example of the core overlap processing in action. The block diagram highlights the row overlap, indicated by the yellow section, occurring between parts positioned at identical offsets within the result, and the column operations, represented in blue. Initially, upon loading all inputs, the shift register begins shifting the valid data to the right. Specifically, the overlap region between the Most Significant Bit (MSB) pixels of DCMKP0 and the Least Significant Bit (LSB) pixels of DCMKP2 is addressed first by shifting DCMKP0, followed by DCMKP2. When it is necessary to process overlaps, the Shift Register Latch (SRL) containing the value from DCMKP2 is shifted accordingly. The output from each cycle is then fed into a Serial-In Parallel-Out (SIPO) buffer.

Once all required pixels, as determined by dynamic parameters, have been processed, the processor retrieves the values. This sequence occurs simultaneously for DCMKP1 and DCMKP3. Following this, the processor evaluates whether to send this data directly to the output or to continue processing for column overlap. Overlap results for DCMKP1 and DCMKP3 are consistently saved to the Dual-Port RAM. If processing for column overlap between DCMKP0

and DCMKP2 needs to continue, the processor activates the Dual-Port RAM's read port to load the previously saved values and proceeds with column processing.

The dynamic parameters play a crucial role in determining which regions experience column overlap. The entire channel is divided into three distinct parts for processing: initially, only the row overlapping between DCMKP0 and DCMKP2 is handled, and the results are directly transferred to the output, while simultaneously, results from DCMKP1 and DCMKP3 are saved to memory. This process is repeated until column overlap processing is executed, and the values stored in the RAM are processed and outputted. In the final stage, the remaining columns within the RAM are read and outputted cycle by cycle. With each output delivery, the core overlap processor seamlessly transitions to the Tilling and Gather Processor, ensuring continuous and efficient data processing.

4.4. TILLING AND GATHER PROCESSOR IMPLEMENTATION

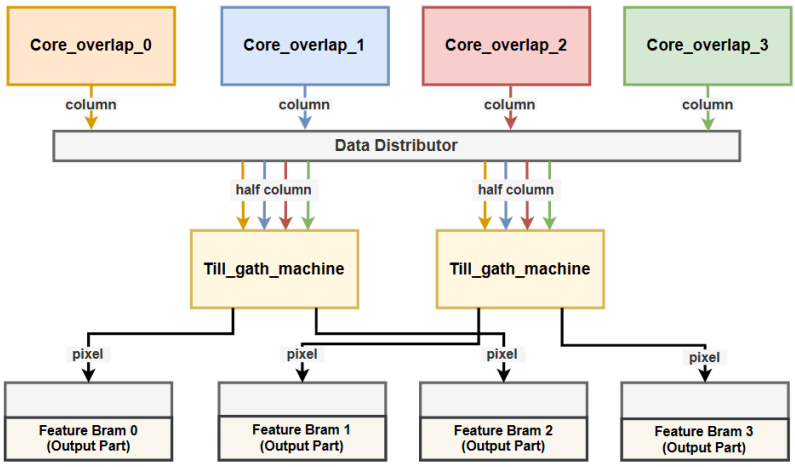


Figure 4.20: Data flow from Core Overlap Processor to Tilling and Gather Processor

The architecture discussed involves a sequence of data processing through different components, where each plays a vital role in handling and transforming the data for further stages. The process begins with the **Core Overlap Processor**, which outputs data one column at a time. Though these outputs initially appear as individual, isolated columns, they are concatenated to form a complete data channel. This concatenation is crucial as it reconstructs the segmented data into a meaningful whole, which is necessary for the subsequent processing stages.

However, forming a complete channel from these columns is not straightforward and requires a systematic accumulation of outputs. This accumulation is not static but dynamic, governed by a parameter N, which specifies the number of channels that need to be processed and accumulated concurrently. The dynamic nature of N introduces flexibility and adaptability into the system, allowing it to handle varying data sizes and requirements efficiently.

The critical component designed to manage this complexity is the **Tilling and Gather Processor**. This processor is responsible for accumulating the required number of channels and ensuring that the accumulated data is coherent and correctly ordered. It achieves this by transferring the output result to the same block memory generator that received the input features. This shared use of memory resources ensures that the data remains aligned and that the output

channels are constructed correctly from the processed columns.

Once the accumulation is complete, the Tilling and Gather Processor has the additional responsibility of routing the accumulated data to the correct memory addresses. This routing is crucial because it ensures that the data channels are stored in the correct sequence and location, preserving the integrity and order of the processed data. The ability of the processor to handle these tasks efficiently is pivotal in maintaining the flow and accuracy of the data processing pipeline within the architecture.

4.4.1. DETAILED OVERVIEW

Figure 4.20 presents a critical aspect of our system architecture by mapping out the data flow from the Core Overlap Processor to the Tilling and Gather Processor. This arrangement is essential to understand the distribution and management strategy for data as it progresses through the processing stages.

TILLING MACHINE FUNCTION

Each Tilling Machine is allocated the task of processing half of the columns received from the Core Overlap Processor. This division of labor is strategic, as it reduces the processing burden by splitting the data across two processors:

- One processor manages the upper half of the data columns.
- The other processor is responsible for the lower half.

Figure 4.21 shows the setup of the processing flow, enhancing throughput and minimizing latency. Within each Tilling Machine, there are two distinct Till Gather Buffers. Each buffer handles a quarter of the overall channel output, processing the data columns in an overlapped fashion. This overlapping is significant as it ensures that the final output maintains a coherent sequence and is correctly aligned in terms of its absolute channel position.

The operational connectivity is highlighted by their links to individual Block RAMs (BRAMs), which are used for storing intermediate results. Specifically:

- **Till Gather Buffer 0** of **Tilling Gather Machine 0** is responsible for processing the first quarter of the channel output.
- **Till Gather Buffer 1** of the same machine takes care of the second quarter.

The other Tilling Machine mirrors this setup for the third and fourth quarters of the channel output, thereby ensuring a balanced processing load across the system.

This configuration not only facilitates efficient data processing but also guarantees that each component can function at optimal capacity without risk of data congestion or bottlenecking.

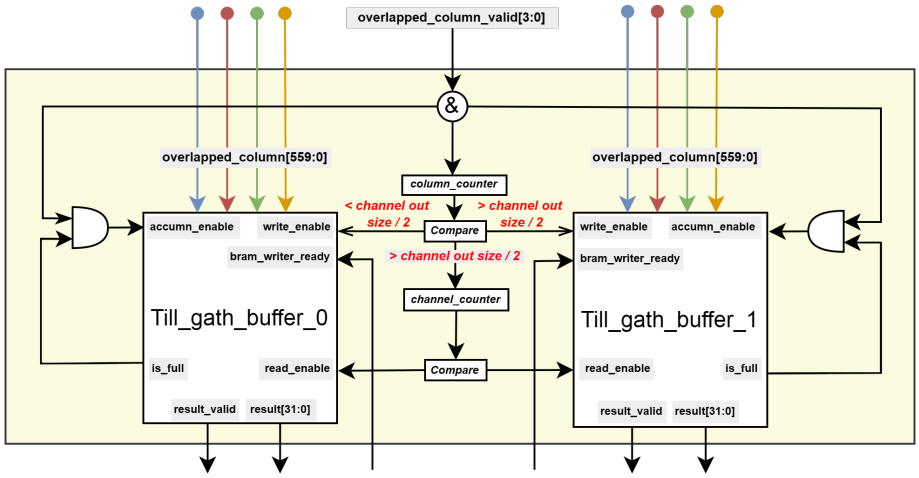


Figure 4.21: Tilling and Gather machine implementation

4.4.2. INTEGRATION OF TILL GATHER CORES AND BUFFERS

As outlined in Section 3, the architecture's accelerator is capable of handling a single feature input alongside four different weights, producing four distinct channel outputs, as illustrated in Figure 3.1. This complex processing requirement necessitates a robust system capable of managing and routing multiple data streams efficiently.

TILL GATHER BUFFER FUNCTION

Each Till Gather Buffer within the system has four **Till Gather Cores** as described in Figure 4.22. These cores are meticulously designed to each handle one quarter of the channel output, thereby dividing the workload evenly among them. This division is critical for managing the flow of data and ensuring that processing times are minimized while maintaining accuracy in the output.

After the processing tasks are completed, the Till Gather Buffer takes on the role of routing the processed results. This routing is followed by two main signals:

- A **ready signal** from the Block RAM (BRAM) which indicates that the memory is prepared to receive new data.
- A **valid signal** from the cores, confirming that the data to be sent is correct and processed.

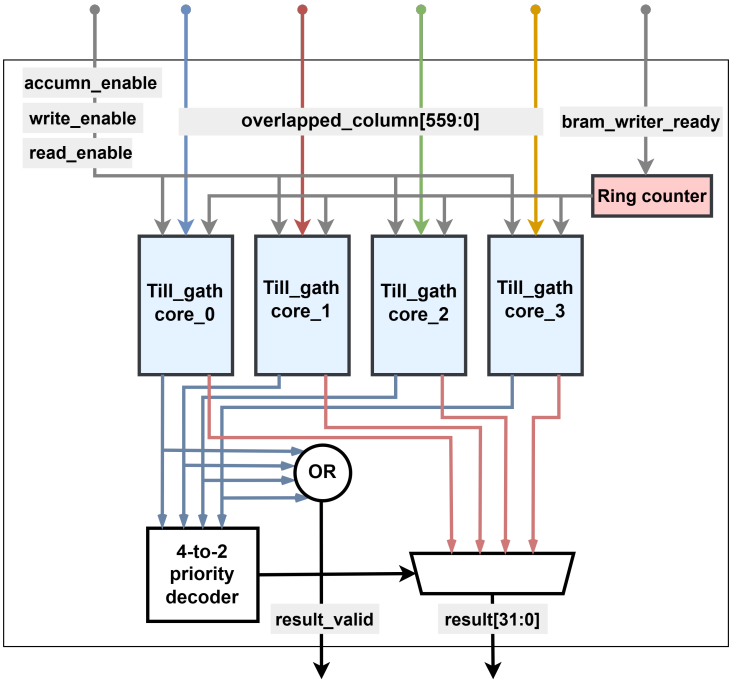


Figure 4.22: Tilling Gather Buffer implementation

The sequence of data transfer to the BRAM is strictly managed based on a predefined order — 0, 1, 2, and 3. This ordering ensures that the data is stored in a manner that preserves the integrity and sequence of the outputs, allowing for systematic retrieval and further processing.

ROLE OF TILL GATHER CORE:
The Till Gather Core is essentially the atomic unit of this processing architecture. It not only processes the data but also plays a pivotal role in the overall operation of the system by receiving and responding to signals from the Till Gather Machine. The core functions as both a processing unit and a control node, directing the flow of data based on the operational signals it receives. This dual functionality allows the core to efficiently manage its tasks while also ensuring that the entire system operates harmoniously.

MONITORING AND CONTROL:
Besides processing, the Till Gather Machine encompasses a broader role that includes monitoring the operation of the Till Gather Cores and managing the

data flow within the system. It acts as a wrapper that not only encapsulates the cores but also provides oversight, ensuring that every part of the data processing is executed correctly and efficiently.

4.4.3. OPERATIONAL DETAILS OF TILLING GATHER CORE

Figure 4.23 describes the detailed setup of Till Gather Cores and Buffers illustrates the complex yet efficient management of data processing tasks within the architecture, ensuring high throughput and precision in handling multiple data inputs and outputs.

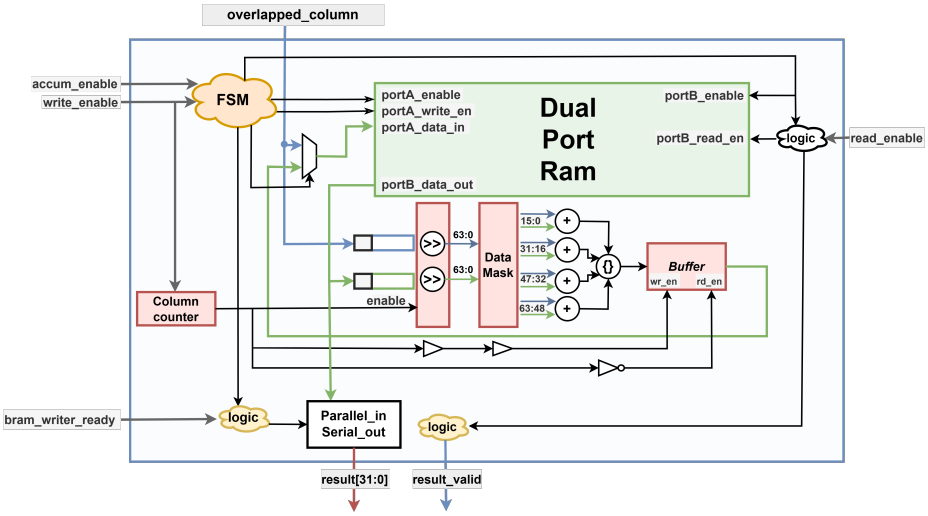


Figure 4.23: Tilling Gather Core Implementation

In the block diagram, there is a **dual-port RAM** with dimensions tailored to match the output of one channel, both in depth and width. This configuration is crucial as it allows the RAM to store results from one accumulation cycle and utilize these stored results for the subsequent cycle, thereby enhancing the efficiency and speed of the data processing.

Initially, the data distributor receives an overlapped column from the Core Overlap Processor. This data is then evenly split into two parts:

- The upper half is directed to Till Gather Machine 0.
- The lower half is routed to Till Gather Machine 1.

This distribution is dynamically managed based on parameters that dictate when and how data should be directed to each buffer.

Upon receiving the data, each buffer evaluates whether to proceed with accumulation based on signals from the Till Gather Machine. These signals ensure that data is accumulated only when appropriate, maintaining the integrity and relevance of the processed results.

The column data is loaded into the dual-port RAM, positioned exactly where it corresponds to its absolute channel position. Data is then transferred to a Shift Register (SRL), where for every shift right of four 16-bit pixels, an adder aggregates the values.Upon completion of the addition, the data is moved to a Serial In Parallel Out (SIPO). When the SIPO is full, the data is again stored in the RAM at the original absolute position.

This process repeats for M channel inputs, where M represents a dynamic parameter adjusting to the data volume and processing requirements.

4.5. PARAMETERS OF THE ACCELERATION CORE

The acceleration core has several parameters to be configured. These parameters can be static and dynamic. Static parameters are only able to be changed before synthesis and can not be changed when the core is operating. Dynamic parameters are only able to change while the core is operating. For maximum flexibility, all parameters should be dynamic. However, some parameters that are hardly changed while operating will create unnecessary logic and resources while synthesizing. That is the reason why we have static and dynamic parameters. Static parameters are implemented using Verilog keyword parameter. These parameters are located at the top module of the design. Changing these parameters will affect the whole design and need synthesis to update the design with a new configuration. These are the static parameters of the acceleration core:

- **BRAM_DATA_WIDTH:** This parameter configures the width of the BRAM data. Each data can contain two to four pixels depending on the size of each pixel.
- **BRAM_ADDR_WIDTH:** This parameter configures the width of the BRAM address. Change this parameter if we want to scale up the Feature BRAMs or Weight BRAMs size. This value must be an integer greater than 0. In addition, it must equal the address width configured in the Block Memory Generator IP of Xilinx.
- **SIZE_OF_FEATURE:** This parameter configures the maximum size of Input Feature Maps in Deconvolution layers datasets.
- **SIZE_OF_WEIGHT:** This parameter configures the maximum size of Weight

Table 4.3: Register Bank description

No.	Address	Type	Description
1	0x0000_0000	RW	Control register
2	0x0000_0004	RO	Status register
3	0x0000_0008	RW	Input parameter register
4	0x0000_000C	RW	Weight parameter register
5	0x0000_000F	RW	Output parameter register

in Deconvolution layers datasets. This value must be an integer greater than 1.

- **PIXEL_WIDTH:** This parameter configures the data width of each pixel. Each pixel is described under fixed point data type. The default value of this parameter is 8.

- **STRIDE_VAL:** This parameter configures the maximum of stride values for Deconvolution layers. This value must be an integer greater than 1.

- **NO_OF_CHANNELS_EACH_WEIGHT:** This parameter configures the maximum number of channels per weight in Deconvolution layers. In general, this value is equal to the Input Feature Channels.

- **NO_OF_WEIGHTS:** This parameter configures the maximum number of weights in Deconvolution layers. The number of weights determines the number of Output Feature Maps.

4.6. REGISTER BANK

The acceleration core has several parameters to be dynamically configured to fit the demand of applications (number of input feature map channels, size of input feature maps, number of kernels, size of kernel, number of output feature map channels or stride). In addition, the acceleration core needs some control registers and status registers to operate, and application software needs some status registers to manage the acceleration core. Therefore, all of this information will be stored and accessed via the Register Bank component as described in Figure 3.1.

Register Bank consists of 32-bit registers which can be accessed from both PS and PL. Application on PS can access Register Bank via AXI memory-mapped interface. The acceleration core in PL can read and write the registers directly. Table 4.2 shows the description of Register Bank from the view of PS.

For **Control register**, it consists of several bits that are used to control the operation of the acceleration core:

- **Bit 0:** PS writes 1 to this bit to enable the computing process of the acceleration core. If PS writes 0, the computing process will stop until it is enabled again.
- **Bit 1:** PS write 1 to this bit to transfer a software reset to reset the acceleration core. This signal is similar to an asynchronous hardware reset.
- **Bit 31-2:** Reserved

For **Status register**, it consists of several bits used to track the operation of the acceleration core.

- **Bit 3-0:** Each bit indicates that the acceleration core has finished the DCKMPs processing phase. There are four DCKMP cores so there are 4 bits corresponding to each core.
- **Bit 7-4:** Each bit indicates that the acceleration core has finished the Core Overlap processing phase. There are four Core Overlap cores so there are 4 bits corresponding to each core.
- **Bit 11-8:** Each bit indicates that the acceleration core has finished the Data Tilling phase. There are four Data Tilling cores so there are 4 bits corresponding to each core.
- **Bit 12:** Enable value of **Control register**
- **Bit 13:** Software reset value of **Control register**
- **Bit 31-14:** Reserved

For *Input parameter register*, it consists of several bits used to configure the operation of the acceleration core.

- **Bit 9-0:** PS writes data corresponding to input size of the deconvolution layer. This value is dynamically configured to initialize the structure of the acceleration core. This value cannot be changed during the computation processing.
- **Bit 19-10:** PS writes data corresponding to the number of input channels of the deconvolution layer. This value cannot be changed during the computation processing.

- **Bit 31-20:** Reserved

For *Weight parameter register*, it consists of several bits used to configure the operation of the acceleration core.

- **Bit 9-0:** PS writes data corresponding to the weight size of the deconvolution layer. This value is dynamically configured to initialize the structure of the acceleration core. This value cannot be changed during the computation processing.
- **Bit 19-10:** PS writes data corresponding to the number of channels per weight of the deconvolution layer. This value cannot be changed during the computation processing.
- **Bit 29-20:** PS writes data corresponding to the number of weights of the deconvolution layer.
- **Bit 31-30:** PS writes a stride value of the deconvolution layer. This value cannot be changed during the computation processing.

4.7. SOFTWARE IMPLEMENTATION

In addition to hardware designing, software programming is a critical section supporting the acceleration core. All the software programs are developed in Python. In this thesis, there are two types of software programs.

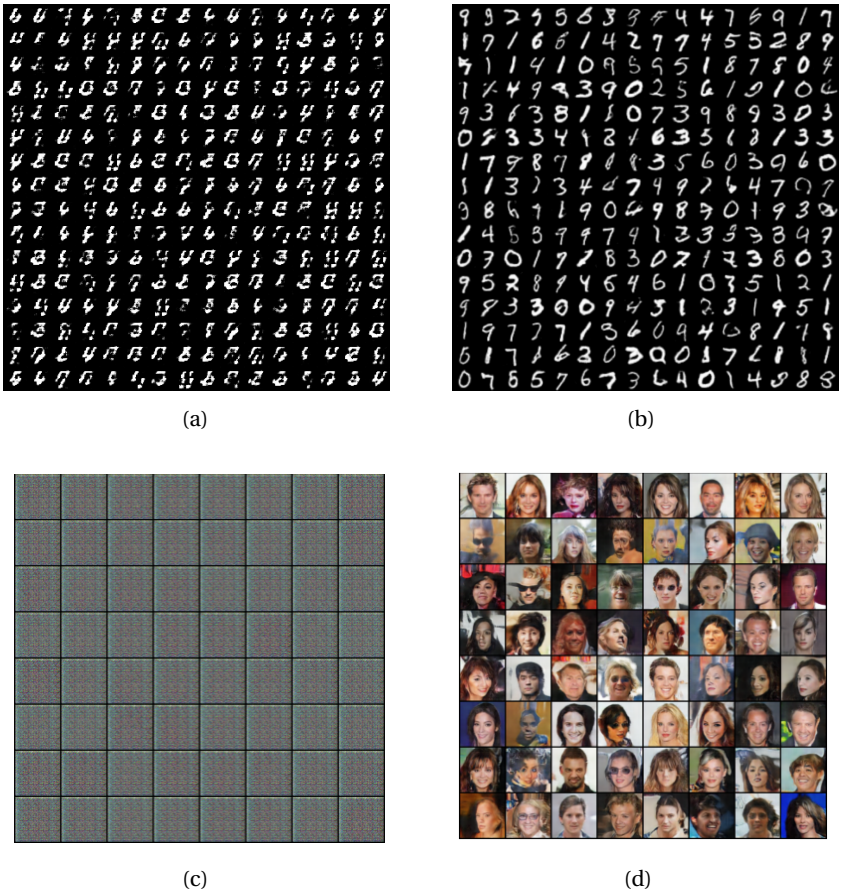
- **The first type** is supporting software which run off-line. The execution of this software is not included in the execution of the acceleration core. These software programs are a simulation data generator, a software version of random forest, an acceleration core results generator, and a files generator. These software programs will be discussed in more detail.
- **The second type** is a software program that will run on the Arm processor of Xilinx SoC while the acceleration core is operating. These applications need to be accelerated by the acceleration core in PL. This application includes an application program, data pre-processing program, DMA controller program, and acceleration core controller program. These programs will be discussed in more detail.

4.7.1. SUPPORTING SOFTWARE

TRAINING RESULTS

FRECHET INCEPTION DISTANCE (FID)

Many popular metrics are used for evaluating GANs for image generation [26]. However, some metrics such as visual inspection, inception score (IS) [27], Frechet



(a) (b)

(c) (d)

Figure 4.24: MNIST and Celeb-A datasets were trained from scratch for 60 epochs

Inception Distance (FID), etc. are common ones. In this thesis, FID is chosen to evaluate the generalization of GANs.

FID stands for Frechet Inception Distance [28] which is a metric used to evaluate the quality and diversity of generated images produced by a Generative Adversarial Network (GAN) or other generative models. FID was introduced as a way to measure the similarity between the distributions of real and generated images in a feature space. Firstly, FID begins by extracting feature representations from a pre-trained deep neural network, typically an Inception-v3 model. This network is used to capture high-level features of images. FID is based on the statistical calculations with Mean (μ) and Covariance (cov).

- Mean (μ): The mean of the feature representations for all images.

- Covariance (cov): The covariance matrix of the feature representations.

FID then calculates the Frechet distance (a statistical distance) between the two multivariate Gaussian distributions defined by the mean and covariance of the feature representations of real and generated images.

$$FID = || \mu_1 - \mu_2 ||^2 + Tr(cov_1 + cov_2 - 2\sqrt{cov_1 * cov_2}$$

A lower FID score indicates that the distribution of real and generated images is closer in feature space meaning that the generated images are more similar to real ones in terms of quality and diversity.

There is no specific range or scale for measuring the FID. However, a lower score indicates better results for both datasets. One thousand images in the original distribution and one thousand from the generated distribution to calculate the FID.

Dataset	FID Score
MNIST	12.453
Celeb-A	48.567

In the first application (image generation), DCGAN is chosen which is used to generate new images from the learning data distribution. The target is implementing efficient deconvolution to save power consumption and on-chip memory access for inference acceleration, not the training phase. Therefore, complete model training is mandatory to extract the weight extraction for further use.

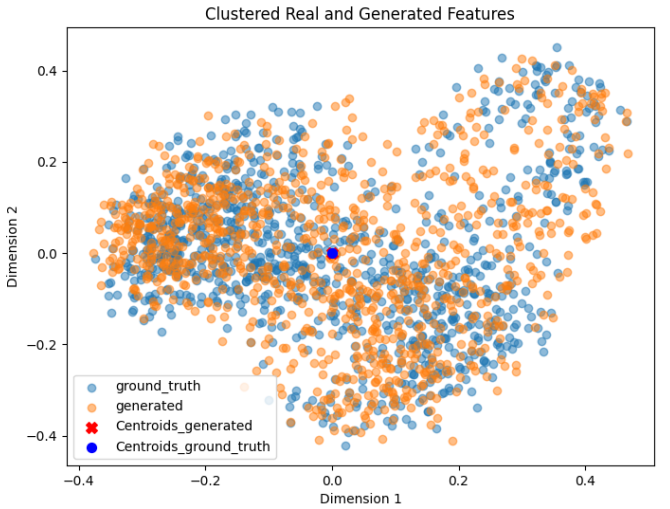
DISTRIBUTION OF GROUND TRUTH AND GENERATED IMAGE VISUALIZATION

A pre-trained model called VGG-16 to extract and capture the high-level feature of the picture. Then, analyzing large datasets with numerous dimensions per observation is effectively facilitated by the widely utilized technique known as PCA (Principal Component Analysis [29]). This approach enhances data interpretability, retains maximum information, and allows for the visualization of multidimensional data. Finally, K-means is used to find the centroid in each distribution dataset and calculate the distance between vectors.

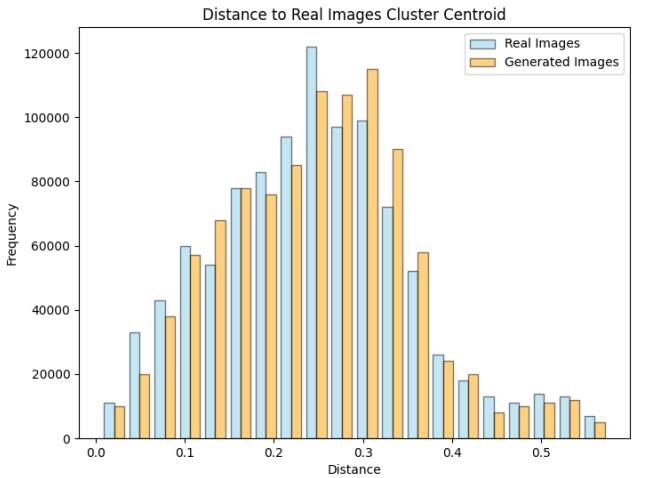
The below figures analyze the difference between the generated images and the truth ones by displaying in scatter plots and histograms for visualizations.

q

From the visualization and results of different evaluation metrics, it can be concluded that the result of weights is good enough and the generated image is closely similar to the real one.



(a) Scatter plot of generated and truth image in different dimensions for MNIST dataset



(b) Histogram of distance comparison between generated image and truth image to its center for MNIST dataset

Figure 4.25: Visualization of evaluation techniques for MNIST dataset training (2 dimensions - 1 channel)

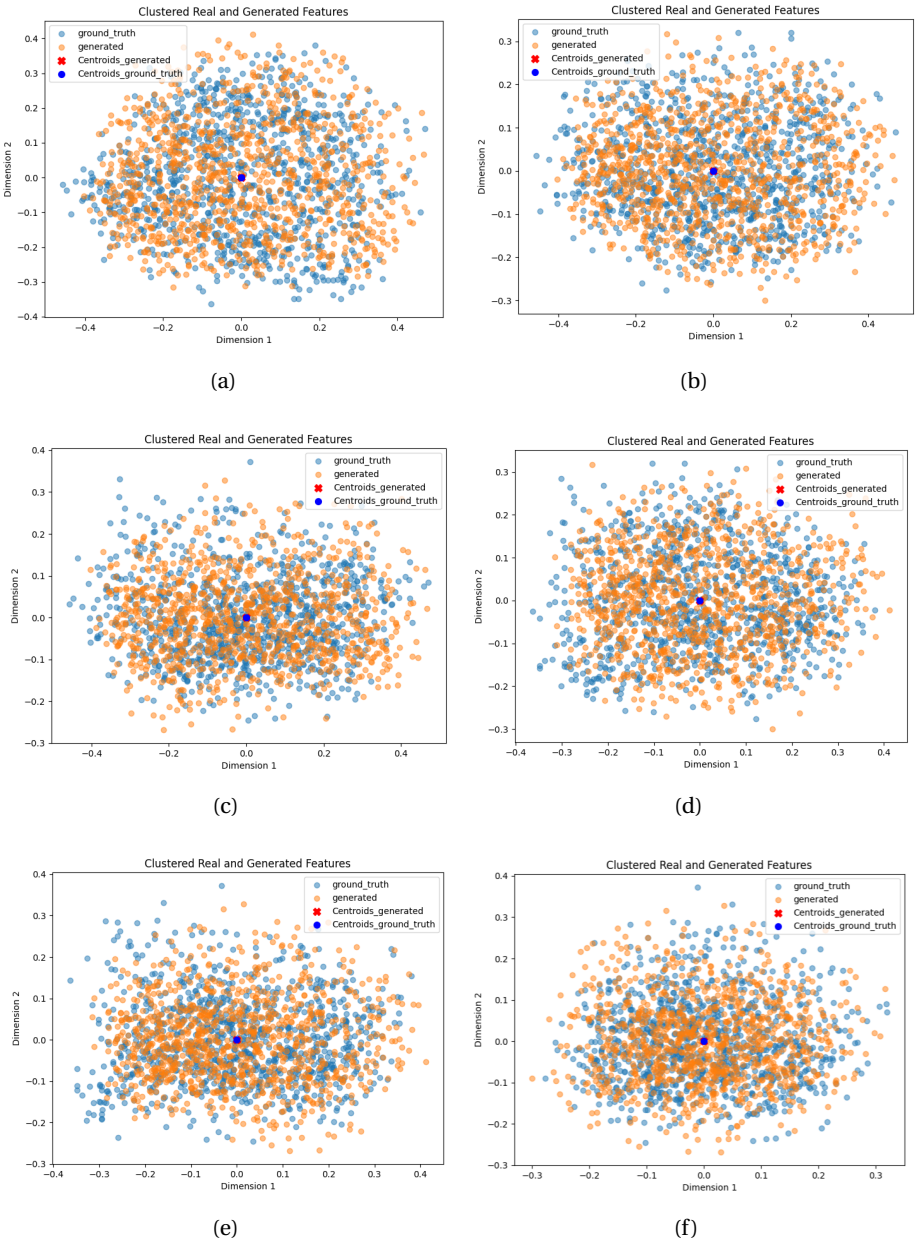


Figure 4.26: Scatter plots of generated and truth image in different dimensions for Celeb-A dataset (4 dimensions - 3 channels)

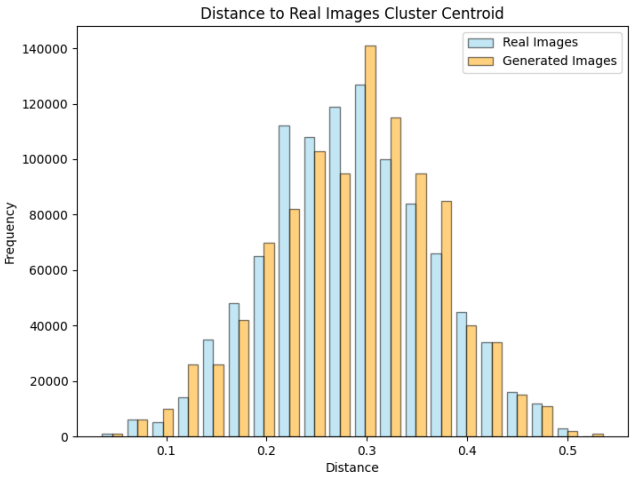


Figure 4.27: Histogram of distance comparison between generated image and truth image to its center for Celeb-A dataset

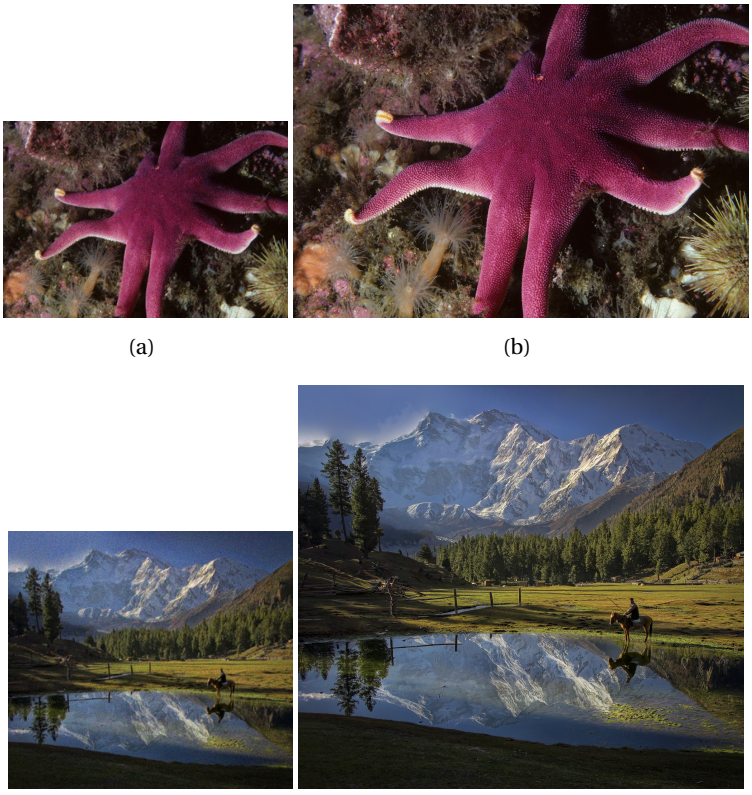
SUPER-RESOLUTION GAN

SRGAN, or Super-Resolution Generative Adversarial Network, is a state-of-the-art deep learning model specifically developed for enhancing the resolution of low-resolution images while preserving intricate details and textures. Introduced by Christian Ledig and his team in their 2017 paper, "Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network" [3], SRGAN utilizes the robust capabilities of Generative Adversarial Networks (GANs) to produce high-quality, realistic images.

Super-resolution involves the process of upscaling and enhancing the quality of an image, which often leads to challenges such as blurriness and loss of detail in traditional methods. SRGAN addresses these issues by employing a dual-component loss function that ensures the enhanced images retain a photo-realistic quality. The model comprises a generator and a discriminator: the generator aims to upscale images by generating high-resolution outputs from low-resolution inputs, while the discriminator evaluates the authenticity of the upscaled images against real high-resolution images.

The training utilizes the DIV2K dataset—a well-known benchmark for single-image super-resolution—comprising 1,000 varied images split into 800 for training, 100 for validation, and 100 for testing.

For this thesis, the SRGAN architecture has been adapted by replacing the



(a) (b)

(c) (d)

Figure 4.28: Comparison of low-resolution and high-resolution images from the DIV2K dataset.

original PixelShuffler layer with a Deconvolution layer to better suit FPGA acceleration, thereby optimizing the inference performance of the super-resolution tasks.

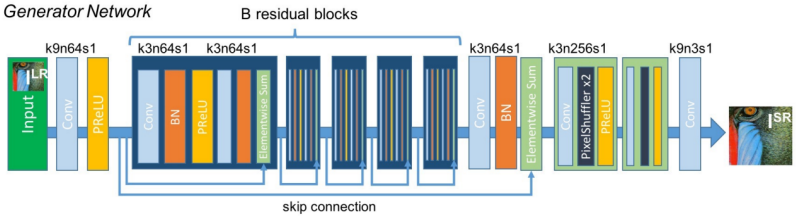


Figure 4.29: Modified SRGAN Generator architecture incorporating deconvolution layers. [3]

DISCRIMINATOR ARCHITECTURE

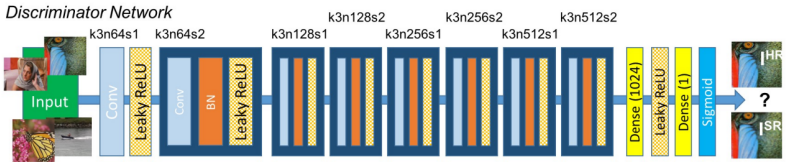


Figure 4.30: SRGAN Discriminator architecture. [3]

4

LOSS FUNCTION

SRGAN employs a sophisticated loss function strategy [30] that combines adversarial, perceptual, and content loss components to refine the super-resolution process:

- **Adversarial Loss**: Pushes the generator to produce high-resolution images indistinguishable from actual high-resolution images.
- **Perceptual Loss** [31] : Measures discrepancies between feature representations of generated images and real images to ensure perceptual similarity.
- **Content Loss**: Focuses on maintaining content and structure integrity between generated images and their original low-resolution counterparts.

The generator's overall loss function is formulated as:

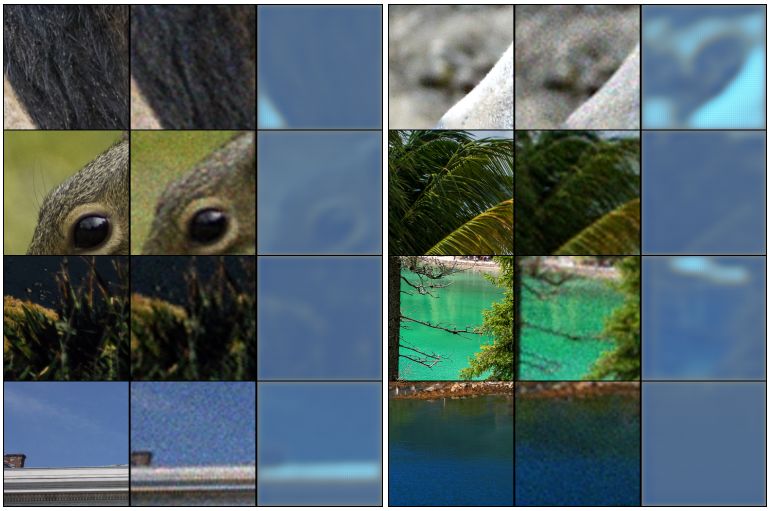
$$L_G = \lambda_{\text{adv}} L_{\text{adv}}(G) + \lambda_{\text{perceptual}} L_{\text{perceptual}}(G) + \lambda_{\text{content}} L_{\text{content}}(G) \tag{4.2}$$

where λ_{adv}, $\lambda_{\text{perceptual}}$, and λ_{content} are hyperparameters that balance the contributions of each loss component. The training process aims to minimize this combined loss, encouraging the generation of high-quality and perceptually convincing images.

Experiments demonstrate that integrating perceptual and content losses significantly enhances the visual quality of generated images, closely mirroring the original's texture and color nuances.

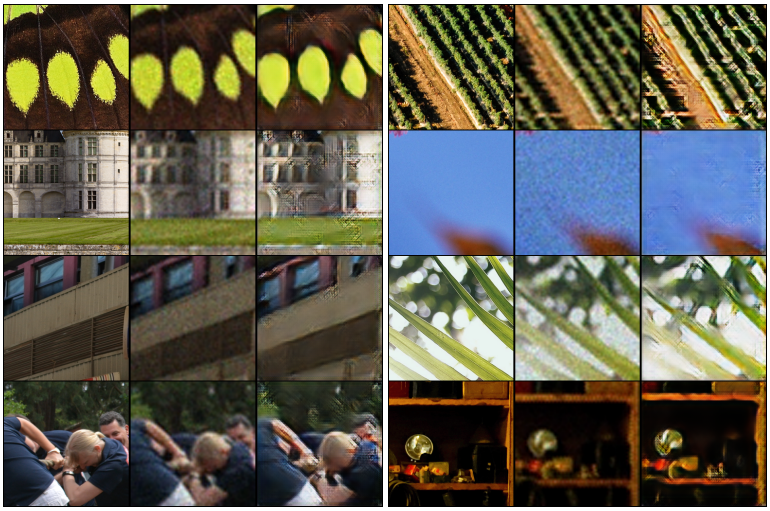
EVALUATION METRICS

Structural Similarity Index Measure (SSIM) SSIM is an advanced metric used to measure the similarity between two images. [32] It is different from traditional metrics like PSNR, which focus on pixel-level differences. SSIM considers changes in structural information, luminance, and contrast, providing



(a) (b)

Figure 4.31: Evaluating the impact of GAN and content losses on image quality.



(a) (b)

Figure 4.32: Assessment of GAN and perceptual losses on image generation.

a more comprehensive understanding of perceived changes in image quality. The SSIM index is a decimal value between -1 and 1, where 1 indicates perfect

(a) (b)

Figure 4.33: Integration of GAN, content, and perceptual losses and their effect on the output.

similarity. The formula for SSIM is given by:

$$SSIM(x,y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)} \tag{4.3}$$

where μ_x, μ_y are the average intensities, σ_x^2, σ_y^2 are the variances, σ_{xy} is the covariance of images x and y, and c_1, c_2 are constants used to stabilize the division.

Peak Signal to Noise Ratio (PSNR) PSNR is another widely used metric for image quality assessment, particularly in image compression and super-resolution. It measures the ratio between the maximum possible power of a signal and the power of corrupting noise that affects its fidelity. PSNR is usually expressed in logarithmic decibel scale. Higher PSNR values indicate better image quality. The formula for PSNR is:

$$PSNR = 10 \cdot \log_{10}\left(\frac{MAX_I^2}{MSE}\right) \tag{4.4}$$

where MAX_I is the maximum pixel intensity of the image (255 for 8-bit images), and MSE is the mean squared error between the original and the degraded image.

Evaluation Results The following table presents the evaluation results of the super-resolved images using SSIM and PSNR metrics. These results provide insight into the effectiveness of the super-resolution techniques implemented in the thesis.

Table 4.4: SSIM and PSNR evaluation results for DIV2K datasets.

Dataset	SSIM	PSNR (dB)
DIV2K	0.98	58.27

4.7.2. APPLICATION SOFTWARE

The application software runs on the Arm processor of Xilinx SoC. This application has two main tasks. The first task is to run the main application. The second task is to manage the PL. This task includes managing communication between PS and PL, and managing the acceleration core. This section will focus on the second task.

To control and verify Deconvolution, the PYNQ framework [7] is used. This framework can run in Jupyter Notebook, so it is easy to manage the source code and allows us to illustrate the results through figures. Firstly, we need to load the design of the acceleration core to configure the PL. This can be achieved in lines 1 and 2 of Listing 4.1. The first line imports hardware library *Overlay*. Overlays have versatile applications as they can be utilized to speed up the performance of a software application or modify the hardware platform to suit the requirements of a specific application. In this case, *Overlay* configures the PL based on the bitstream file.

```
1 from pynq import Overlay
2 gan_accelerator = Overlay("design_1.bit")
```

Listing 4.1: PL configuration in PYNQ

After running the overlay function, the acceleration core is successfully configured. Then, we need to set some address space and block memory generator for PS to access the memory in PL as shown in Listing 4.2. Line 2,3 configure the address space for accessing Register Bank. Line 5-9 configure the address space for accessing the `Control` register. Line 11-16 configure the address space for accessing the `Status` register. Line 18-22 configure the address space for accessing the `Parameter` registers. Line 24,25 configure the address space for accessing 4 Weight BRAMs. Line 26,27 configure the address space for accessing 4 Feature BRAMs. Finally, the last three lines configure the address space for accessing Out BRAMs.

```
1 # reg bank base address
```

```
REG_BANK_BASE_ADDRESS   = 0x00_A001_0000
REG_BANK_ADDRESS_RANGE = 64*1024

CONTROL_REG   = {
    "BASED_ADDRESS"  : 0,
    "CORE_ENABLE"    : 0,
    "USER_RESET"     : 1
}

STATUS_REG = {
    "BASED_ADDRESS"                : 4,
    "DECONV_OPERATION_FINISH"      : 0,
    "DECONV_OVERLAP_FINISH"        : 1,
    "DECONV_TILLING_FINISH"        : 2
}

PARAM_REG = {
    "FEATURE" : 8,
    "WEIGHT"  : 12,
    "OUTPUT"  : 16
}
# Feature base address
WEIGHT_BRAM_BASE_ADDRESS     = [0xE000_0000, 0xE000_8000, 0xE001_000, 0
    xE001_8000]
WEIGHT_BRAM_ADDRESS_RANGE    = 8*1024
FEATURE_BRAM_BASE_ADDRESS    = [0xE002_0000, 0xE002_8000, 0xE003_0000, 0
    xE003_8000]
FEATURE_BRAM_ADDRESS_RANGE   = 4*1024

# OUTPUT BRAM BASE ADDRESS
OUTPUT_BRAM_BASE_ADDRESS   = [0xE002_4000, 0xE002_C000, 0xE003_4000, 0
    xE003_C000]
OUTPUT_BRAM_ADDRESS_RANGE = 4*1024
```

Listing 4.2: Configure address spaces in PYNQ framework

Then, we write functions for accessing registers in Register Bank in Listing 4.3. Line 1 imports the MMIO library to allow a Python object to access addresses in the system memory. In particular, registers and address space of peripherals in the PL can be accessed. Then, *mmio_write* and *mmio_read* provide address and data accessing in the register. Line 9-12 show how to use the function.

```
from pynq import MMIO
def mmio_read(mmio, address):
    return mmio.read(address)
def mmio_write(mmio, address, value=0):
    mmio.write(address, value)

```

```
mmio = MMIO(REG_BANK_BASE_ADDRESS, REG_BANK_ADDRESS_RANGE)

mmio_write(mmio,PARAM_REG["FEATURE"],65546)
mmio_write(mmio,PARAM_REG["WEIGHT"],2181103620)
# RESET deconvolution core
mmio_write(mmio,CONTROL_REG["BASED_ADDRESS"],2)
mmio_write(mmio,STATUS_REG["BASED_ADDRESS"],4)
```

Listing 4.3: Configure address spaces in PYNQ framework

To control the CDMA, we also need a procedure to configure its registers. This procedure is developed in master thesis [33]. Listing4.4 shows the example of using the procedure function.

```
feature_buffer = [0,0,0,0]
weight_buffer  = [0,0,0,0]
# Transfer input to deconvolution core
for feature_sub_idx in range(0,4):
    feature_buffer[feature_sub_idx] = allocate(shape=(4*1024,), dtype=np.
    uint32)
    for random_value in range (0,4*1024):
        feature_buffer[feature_sub_idx][random_value] = random_value
        #print(feature_buffer[feature_sub_idx])
    axi_cdma.transfer(0, feature_buffer[feature_sub_idx].physical_address,
     0x00_A000_0000, FEATURE_BRAM_BASE_ADDRESS[feature_sub_idx], 4*1024)
# Transfer kernel to deconvolution core
for weight_sub_idx in range(0,4):
    weight_buffer[weight_sub_idx] = allocate(shape=(8*1024,), dtype=np.
    uint32)
    for random_value in range (0,8*1024):
        weight_buffer[weight_sub_idx][random_value] = random_value
    axi_cdma.transfer(0, weight_buffer[weight_sub_idx].physical_address, 0
    x00_A000_0000, WEIGHT_BRAM_BASE_ADDRESS[weight_sub_idx], 8*1024)
```

Listing 4.4: Configure address spaces in PYNQ framework

Figure 4.34 below describes the general top-view of application software operation.

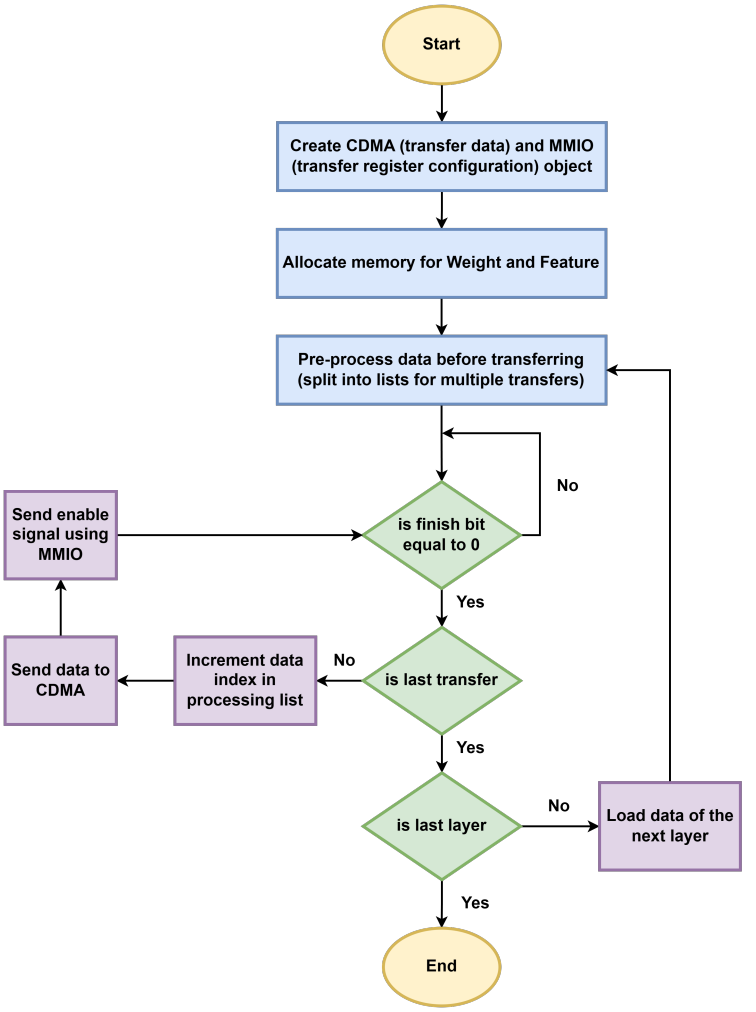


Figure 4.34: Application Software Operation Flow Chart

5

EXPERIMENTAL RESULTS

The following chapter outlines our experiments to evaluate the effectiveness of our proposed FPGA-based flexible deconvolution operation acceleration for edge computing platforms. Initially, we introduce our performance model after using optimizations. Then, we will discuss our testing environment, which features two edge computing platforms: the ZCU106 and Kria KV260 boards. We subsequently examine the results of our experiments which compare the performance of our system with that of Intel-based and ARM-based systems, focusing on execution time and image-generated quality. Additionally, we analyze and compare power and energy consumption.

5.1. PERFORMANCE MODEL

This section provides a detailed analysis of the performance of our hardware deconvolution accelerator using some optimization techniques.

Before analyzing specific terms related to performance measurements, the optimizations we have implemented are discussed below with 2 main points:

BLOCKING STRATEGY

The input buffer cannot accommodate a layer's entire input feature map due to the constrained memory resource on FPGA. Therefore, designing a blocking methodology is necessary to solve the memory resource problems. Blocking involves partitioning the input feature maps into four smaller segments. The maximum size of these blocked inputs depends on the available memory resources and the allocating size of the input buffers. These blocked inputs are processed in parallel to improve the throughput. However, this method has a trade-off between throughput and hardware-resource efficiency.

PARALLEL COMPUTATION

Other standard optimization techniques are parallel computation and data quantization. In this work, we fully parallelize the computation of kernel matrices within deconvolution layers since the kernel (weight) size is typically small ($k = 2,3,4$). We delve into the design space, specifically focusing on data parallelism (P_v) and filter parallelism (P_f):

- *Data Parallelism (P_v):* P_v denotes the extent to which input data is processed concurrently. Multiple weights can be computed simultaneously for input data set during matrix multiplication. This level of parallelization involves reading multiple inputs per cycle and replicating computation kernels P_v times. However, P_v is constrained by the block size and memory bandwidth.

- *Filter Parallelism (P_f):* P_f is the number of filters processed in parallel; N_F represents the maximum number of filters. Filter parallelism requires replicating both compute kernels and output buffers, increasing memory and logic overheads compared to data parallelism (P_v). However, because the block width or height limits P_v due to the blocking method, P_f becomes essential for even more performance improvement.

For each computation layer, the execution time includes: (1) loading data from off-chip memory into on-chip memory (Feature BRAMs and Weight BRAMs); (2) finishing computation for all the computations of involved channels and filters; (3) writing computation results back into off-chip memory. Table 5.1 describes some terms definitions in our performance models:

Table 5.1: Terms definition in the performance model

Parameter	Definition
P_v	Data-level parallelism
P_f	Filter-level parallelism
Freq	Clock frequency
BW_i	Memory bandwidth for loading data into kernel
BW_o	Memory bandwidth for writing data into kernel
DW	Data width
H	Height of the input feature map
W	Width of the input feature map
H_o	Height of the output feature map
W_o	Width of the output feature map

1. Load time i.e., time to load N_C channels and $N_C * N_F$ weights into cache:

$$T1 = \frac{N_C * H * W * DW}{Freq * BW_i} + \frac{N_C * N_F * k * k * DW}{Freq * BW_i} \tag{5.1}$$

2. Computation time i.e., time to compute all channels for the entire weights:

$$T2 = \frac{N_C * H * W * N_F}{P_v * P_f * Freq} + \left(\frac{6 * H_O + H_O * W_O}{16 * P_f * Freq}\right) \tag{5.2}$$

3. Write time i.e., time to write the output result into memory

$$T3 = \frac{N_F * H_O * W_O * DW}{Freq * BW_o * P_f} \tag{5.3}$$

Therefore the overall execution time to compute the result of one deconvolution layer is:

$$T_{layer} = T1 + T2 + T3 + t_{delays} \tag{5.4}$$

The total execution time of the whole network is the sum of the time of each layer.

5.2. EXPERIMENTAL SETUP

In this chapter, we present our experimental setup to thoroughly evaluate the performance and inference capabilities of our FPGA-based scalable deconvolution operation acceleration for edge computing platforms. We deploy our system on two boards, the Kria KV260 and ZCU106, with limited hardware resources. If our accelerator performs well on these boards, it can easily be deployed on various FPGA devices. The Kria KV260 board is ideal for beginners and advanced PYNQ framework projects. It features a K26 System on Module (SOM) with a Zynq Ultrascale+ MPSoC xck26-sfvc784-2LV-c chip with 117120 LUTs, 234240 Flip-Flops, and 144 BRAMs on the PL side. The PS has a Quad-core Arm Cortex-A53 MPCore up to 1.5GHz. On the other hand, the ZCU106 board is suitable for video encoding and decoding applications with a ZYNQ Ultrascale+ XCZU7EV-2FFVC1156. This chip has 230400 LUTs, 460800 Flip-Flops and 312 BRAMs which allows the large-scale of hardware design. We use Verilog to implement the entire system for flexibility and scalability. The processors of the two boards act as host processors for running a software application, and the deconvolution computation of the application is offloaded to the acceleration core in PL through DMA. Our primary objective was to assess the extent to which hardware acceleration reduces inference time and resource consumption compared to a software-only implementation. Although only the deconvolution layer was implemented in hardware, it's important to note that a typical

GAN architecture includes additional layers, such as batch normalization and activation functions, which also impact overall system performance. Figure 5.1 and Figure 5.2 illustrate the diagrams of the two boards.

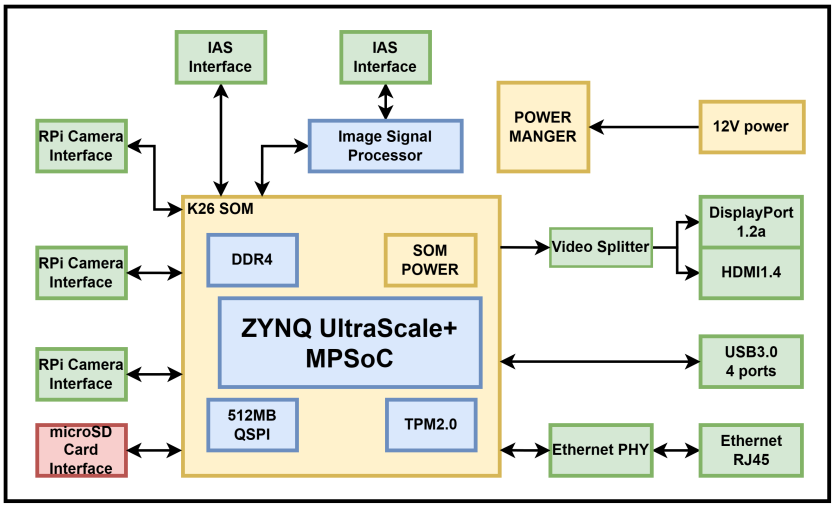


Figure 5.1: Kria KV260 board diagram

We evaluate the performance of our system by running some workloads on different hardware platforms:

- **Software on Intel Processors:** Software is deployed on Intel Core i7-10750H CPU 2.60GHz (desktop processor). Library Pytorch is used to execute the workload.

- **Accelerator on FPGAs:** Our system is deployed on Kria KV260 and ZCU106 boards. Deconvolution operation is done completely by the accelerator, which is deployed with different configurations of Deconlution layers. PYNQ framework is used to develop PS tasks and maximize the efficiency of PS-PL communication. PS controls the accelerator via the Register Bank as specified in Figure 3.1. Weights and input features are accessed in DDR4 memory and stored in PS DRAM, then transferred to the accelerator via DMA. For testing and evaluation purposes, we run inference time with different layers and datasets. The following sections present our results with these configurations.

The workload for evaluating **deconvolution** acceleration is the handwritten digits [34] and CelebFaces Attributes [35]. The motivation behind studying image

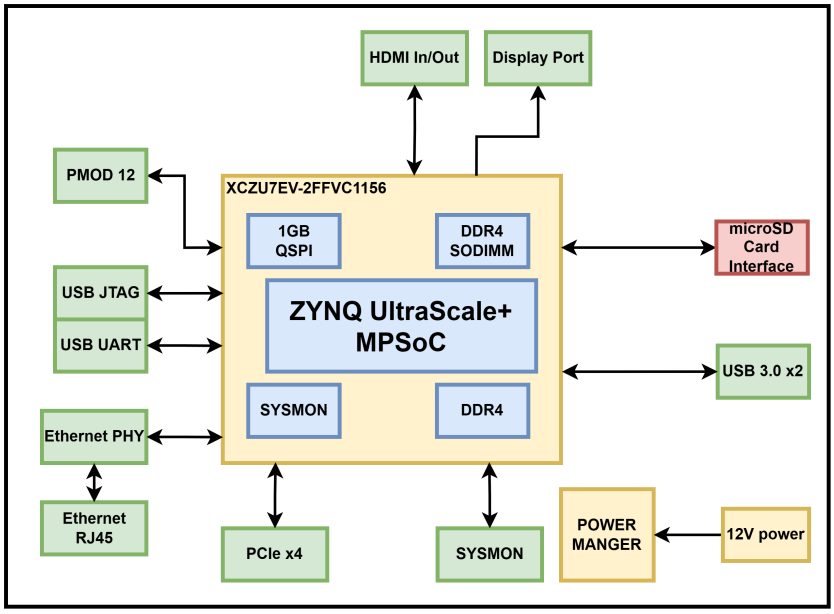


Figure 5.2: ZCU106 board diagram

generation of various types of datasets is to understand the concept of generating modeling. In addition, those datasets play as the cornerstone for assessing the performance and efficacy of generative models. MNIST dataset is a collection of 70,000 grayscale images of handwritten digits (0 through 9), each measuring 28x28 pixels. It serves as a foundational benchmark in machine learning, particularly for tasks like image classification and digit recognition. On the other hand, the CelebA dataset is a large-scale face attributes dataset with more than 200,000 celebrity images, annotated with 40 binary attributes per image such as hair color, age, gender, and presence of eyeglasses. After training and fine-tuning, the data is defined as a 32-bit floating point as a default format. However, it is hard to compute in a hardware environment. Therefore, we need to quantize them into an 8-bit fixed-point format.

For deploying on an **application**, we choose the image super-resolution application with the dataset of 1000 different scenes [36] to evaluate the performance of the acceleration core. DIV2K comprises high-resolution images, typically at 2K resolution (2048x1080 pixels), making it ideal for tasks like image super-resolution. Its diverse content, covering various scenes and objects, enables GANs to learn and generate realistic images across different domains effectively. However, the problem's input is the large size of the input feature

maps compared to the constrained hardware resources of FPGA which may lead to performance bottlenecks and scalability issues. To address this problem, we decide to divide the training data into smaller groups and offload sequentially to the acceleration core via CDMA. The complete application consists of three main steps: reading the input features and weights, performing upsampling data, and generating the output image.

As said earlier in Chapter 3 and Chapter 4, all DCMKPs operate dependently with each other and the acceleration core can compute four set of weights in parallel for each execution. The next weights will be processed if all DCMKPs finishes passing the current output data to the *Core Overlap Processors*. Therefore, the task of updating data buffers including *Input Feature Loaders* and *Weight FIFOs* in DCMKPs also operate at the same time. This mechanism could save data transferring time to DMA instead of waiting for the final output written into the output *Feature BRAMs*. Synchronous updating increases the transferring speed significantly based on the following experiment. The survey is about transferring four blocks of data from DRAM to four BMG (Block Memory Generator) (the first block is transferred to BMG_0, the second block is transferred to BMG_1, the third block is transferred to BMG_2, the last block is transferred to BMG_3). See Figure 5.3 for more details. Each BMG is 64KB in size and contiguous in address; each data block is 16KB in size. There are two methods of transferring listed below. The experiment was carried out to evaluate which way is more efficient.

- **First method: 4-shot transfer**. Transfer each data block into each BMG, which means there are four separate DMA transfers. Each transfer will move 16KB of data to a single BMG. In total, DMA will transfer 4*16KB data.

- **Second method: 1-shot transfer**. DMA will transfer one block of data *[16KB first block data - 48KB meaningless data - 16KB second block data - 48KB meaningless data - 16KB third block data - 48KB meaningless data - 16KB fourth block data - 48KB meaningless data]*. The 16KB meaningful data block is one of four blocks we want to transfer to a BMG. In total, DMA will transfer 4*64KB data in a single transfer. Therefore, the transferred data is four times larger than the first method.

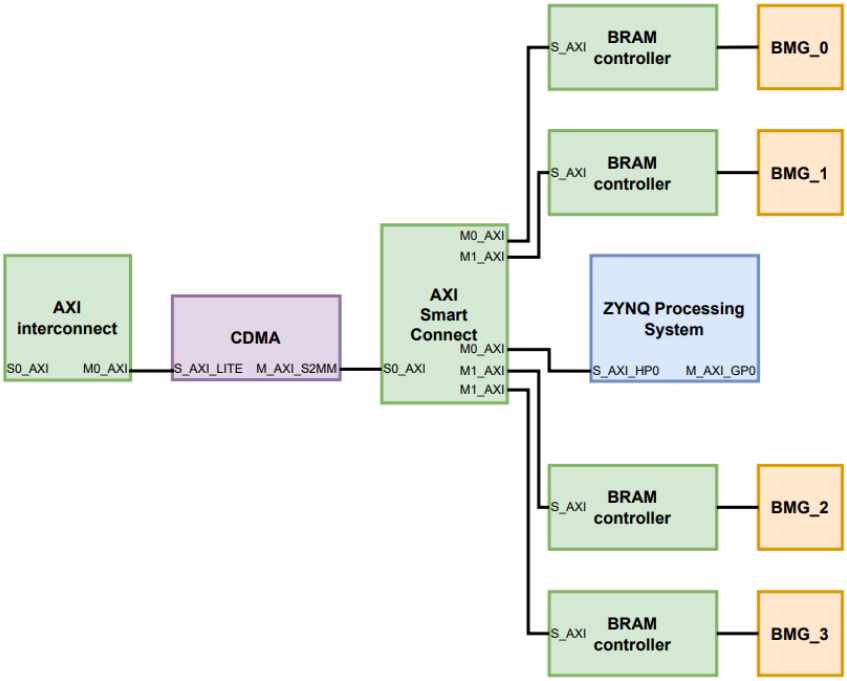


Figure 5.3: Block design for DMA transferring time experiment

In addition, we employ an intermediate approach for efficient data transfer between the software and hardware layers to streamline the experimental process. This involved transferring relevant data, such as weights and input values, from software to Block RAM (Random Access Memory) for computation in the hardware layer. Once the computation is complete, the output is sent back to the software layer for further processing.

This approach provides an effective feedback loop between the two environments, with the following key steps:

- **Data Preprocessing:** Input data is preprocessed in software to ensure compatibility with the hardware layer.
- **Data Transfer:** Preprocessed data and layer-specific weights are sent to Block RAMs for the acceleration process.
- **Hardware Computation:** The deconvolution layer processes the data using optimized FPGA (Field-Programmable Gate Array) logic.
- **Output Transfer:** The computed output is transferred back to the software for subsequent layers.

This setup allows us to benchmark performance improvements and resource consumption reductions achieved through hardware acceleration. Figure 5.4 illustrates the data flow between the software and hardware components.

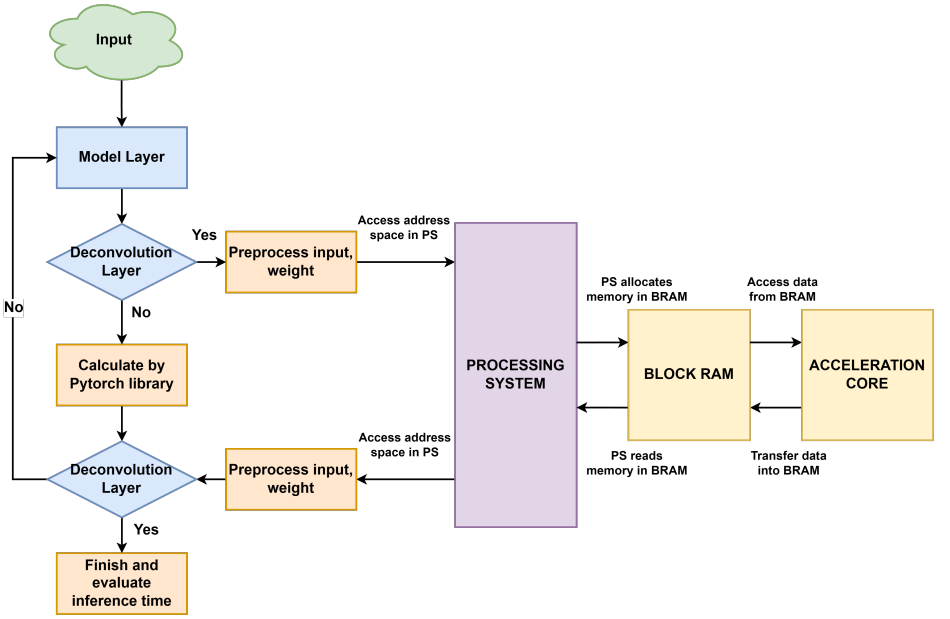


Figure 5.4: A flow diagram illustrates the data transfer and processing between software and hardware components.

Since the experiment is finished, we analyze the results in various aspects including inference time efficiency, resource consumption, and image quality.

- **Inference Time Efficiency:** The hardware-accelerated approach shows a significant reduction in inference time compared to the software implementation. On average, it achieved an X% decrease in inference time, leading to a Y-fold speedup.
- **Resource Consumption Reduction:** By offloading computational tasks to the hardware, software memory usage and computational overhead are reduced by Z%, enabling more efficient resource utilization.
- **Image Quality Assessment (FID Score):** The Frechet Inception Distance (FID) score is used to measure the quality of generated images. The hybrid approach achieved an FID score, similar to or better than the software implementation, indicating that hardware acceleration maintained or improved image quality.

5.3. SYNTHESIS RESULTS

The acceleration core is designed to be flexibly configured. Therefore, we will utilize its flexibility and scalability to configure the acceleration core to fit a specific Generative Adversarial Networks model but with multiple configurations. This will help us to maximize the performance and flexibility of the problem. Therefore, the following results are divided into different boards.

Table 5.2: Hardware resources on Kria KV260 and XCU106 boards

Device	Maximum Input Size	LUTs	FFs	BRAMs	DSPs	Frequency
Kria Kv260	10x10	97130	91634	144	NA	130MHz
		82.93%	39.12%	100%	NA	
ZCU106	22x22	172570	103195	232	734	155MHz
		74.90%	22.39%	74.36%	42.48%	

Kria KV260 Board: The Kria KV260 exhibits a high LUT utilization at 82.93%, indicating a substantial use of the available logic resources which may restrict further functional additions or handling of larger designs without design modifications. Additionally, the utilization of flip-flops is relatively low, suggesting that the current design is heavily combinatorial. The BRAM utilization is fully maximized at 100%, reaching the capacity limit for on-board memory resources.

ZCU106 Board: For the ZCU106, LUT utilization is at 74.90%, which represents a moderate level of logic usage and a slight decrease, allowing for potential additions or alterations in the design. Flip-flop utilization on this board is 22.39%, which is relatively low, potentially indicating under-utilization of available sequential logic resources or efficiency in logic design. BRAM utilization has slightly decreased to 74.36%, suggesting either a more efficient use of memory or changes in the design that free up some memory resources. Furthermore, the board operates at a higher frequency of 155MHz, compared to the KV260, suggesting the potential for higher throughput or a balance between power consumption and processing speed.

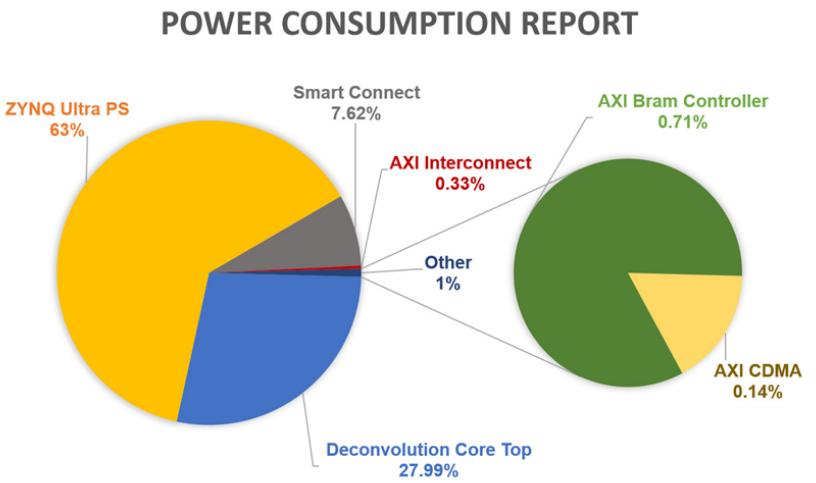


Figure 5.5: Detailed Power Consumption report

Table 5.3: GOPs and GOPs/DSPs

Device	GOPs	GOPs/DSPs
ZCU106	149.7	0.2

In the analyzed dynamic power consumption with totally - **4.216W**, as shown in figure 5.5 of the given architecture implemented on board ZCU106, the **Deconvolution Core** (`deconv_core_top`) represents a significant share of the total power usage, accounting for **27.9782%**. This places it as the second largest consumer of power in the system, trailing only behind the **Zynq Ultra Programmable System** (`zynq_ultra_ps`), which dominates the power distribution at **63.17%** of Dynamic Power. On the other end of the spectrum, the **Configuration Space** (`configuration_space`) is observed to use a minimal **0.0475%** of the total power, highlighting its efficiency in energy consumption relative to its functionality.

Otherwise, Table 5.3 shows the report of GOPs (Giga Operations Per Second) together with Resource Efficiency (GOPs/DSPs):

5.4. SIMULATION RESULTS

Figure 5.6 shows how input weights and features are fed into Deconvolution Multiplier modules in DCMKP. These modules are the processors that describe the deconvolution operation. When the core begins to compute, DCMKP will

request data from corresponding buffers which store available data. Since DCMKP contains 4 small processing cores inside to compute parallel multiple weights, the data is distributed into different data paths. The number of weight and input feature pixels loading for each execution is equal to the size of input features and weights already configured through *Register Bank* before the core starts. In this module, we use Multiplier IPs generated by *IP Catalog*. The number of IPs is dependent on the maximum size of the weight. The bit-width for input data is 8 bits and the number of pipeline stages is 1. The core is waiting for the enable signals (*i_enable_w* and *i_enable_f*) to start computing. From now on, the output data is figured out with a latency of 3 clock cycles.

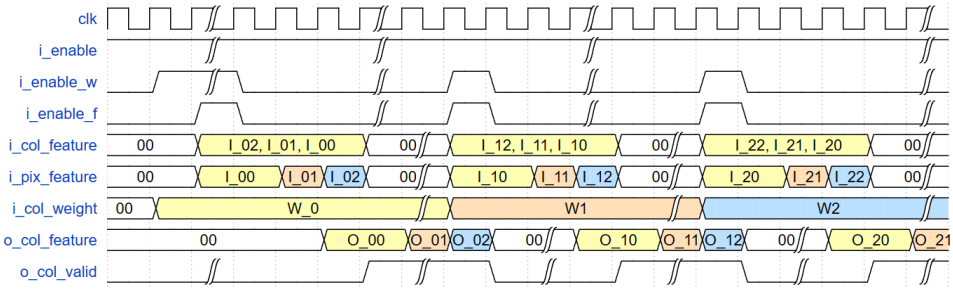


Figure 5.6: Deconvolution multiplication simulation on Vivado

5.5. PERFORMANCE VALIDATION AND ANALYSIS

5.5.1. DMA TRANSFERRING TIME

As discussed in the Experimental Setup section, we compare the performance of a 4-shot transfer and a 1-shot transfer. Despite the 1-shot transfer involving four times more data than the 4-shot transfer, the 1-shot transfer is still approximately 3.5 times faster than the 4-shot transfer, as shown in Table 5.4. This performance difference is attributed to the overhead incurred when starting a DMA transfer, which requires a significant number of clock cycles to configure the register for the transfer. Based on the results of this experiment, we have decided to update all DTPs simultaneously. This approach helps reduce DMA transfer time between PS and PL.

5.5.2. EXECUTION TIME AND SPEED UP

Upon the successful completion of the simulation and synthesis phases, we proceeded to evaluate our architecture's performance in real-time operations. This phase was critical to gauge the practical implications of our design choices and to assess the operational efficacy across various configurations. Our evalu-

Table 5.4: DMA transferring time of 2 methods.

	4-shot(seconds)	1-shot(seconds)	Ratio(4-shot/1-shot)
	0.0123538971	0.0067250729	1.8369908179
	0.0189239979	0.0075993538	2.4902114576
	0.0084810257	0.0043120384	1.9668251686
	0.0229396820	0.0022013187	10.4208816203
	0.0241966248	0.0040044785	6.0423910455
	0.0150949955	0.008042812	1.8768304974
	0.0096974373	0.0021996498	4.4086277910
	0.0092284679	0.0020589828	4.4820518759
	0.0092284679	0.0162391663	0.5682845901
	0.0118052959	0.0042371750	2.7861242404
	0.0079023838	0.0042371750	1.8650123790
	0.0079023838	0.0042605400	1.8547845551
	0.0086059570	0.0024759769	3.4757823784
	0.0105762482	0.0020997524	5.0369024639
MEAN	0.0126383475	0.0050495352	3.5079786344

ation focused on different layer complexities, data throughput levels, and computational intensities, providing a robust examination of the architecture under a spectrum of use cases.

The performance assessment was carried out on three distinct computing platforms, each representing unique environments with varied capabilities:

- **Intel(R) Core(TM) i3-1005G1 CPU @ 1.19GHz:** This processor is known for its reliable performance in standard computing tasks. It served as a baseline to demonstrate the efficiency of our architecture under modest performance conditions.
- **Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz:** With significantly higher processing power, this CPU tested the scalability and speed limits of our architecture, providing insights into its performance in high-end applications.
- **ARM Cortex-A53:** Common in mobile and embedded devices, this processor helped us understand how our architecture performs in power-sensitive and resource-constrained environments.

Each platform ran identical configurations to measure and compare the performance directly. This comparative analysis is crucial as it reveals how our

architecture adapts to different hardware specifications and operational demands. The results are presented below, showcasing processing times, power consumption, and throughput rates to provide a comprehensive view of our architecture's real-world performance.

Here are some methodology notes:

- *Average Values:* The results depicted in the table are based on average values derived from multiple inference runs.
- *Testing Conditions:* Tests were conducted on a laptop, potentially subject to performance variations due to:
  - Background processes
  - Thermal throttling
- *Environmental Variability:* Such variability could potentially affect the inference times, although measures were taken to minimize these impacts during testing.

Table 5.5: Performance comparison across different computing platforms. The table showcases processing times (in seconds) for various deconvolution layers.

Parameter (k,s,H,W,Nc,Nf,Ho,Wo)	FPGA	Intel Core i3	Intel Core i7 10750H	ARM Cortex-A53
(4,2,4,4,128,64,10,10)	0.00058	0.003495	0.0009885	0.007
(4,2,10,10,64,32,22,22)	0.001784	0.00096	0.0007975	0.0048
(3,1,22,22,32,1,24,24)	0.00022526	0.0095	0.000635	0.0007
(2,2,4,4,512,256,8,8)	0.0026407	0.0041	0.000959	0.005
(2,2,8,8,256,128,16,16)	0.001134	0.0008	0.000625	0.002365
(4,2,16,16,128,64,34,34)	0.001459	0.00226	0.00135	0.0161

From Table 5.5, our comprehensive evaluation of the architecture's performance across various computing platforms yields several key insights into its efficacy and potential areas for improvement. While our architecture demonstrates notable advantages in certain aspects, it also reveals some limitations when benchmarked against conventional CPUs.

PERFORMANCE AGAINST ARM CORTEX-A53:
Our architecture significantly outperforms the ARM Cortex-A53 across all tested configurations. This result is particularly encouraging, as it confirms our design's capability to enhance computational efficiency in power-sensitive environments typically dominated by ARM processors. The superior performance

in these settings illustrates our architecture's potential for applications in mobile and embedded systems, where energy efficiency and processing speed are crucial.

COMPARISON WITH INTEL CORE I3:

When compared to the Intel Core i3, our architecture shows mixed results. It achieves faster processing times in some deconvolution layers but does not consistently surpass the i3's performance across all configurations. This performance variance suggests that while our architecture can be optimized for specific tasks, its general applicability across diverse operational conditions remains a challenge. The slight improvements observed in certain layers underscore the need for targeted optimizations to fully leverage the architectural benefits.

LIMITATIONS WITH INTEL CORE I7:

The comparison with the Intel Core i7-10750H reveals a significant performance gap. Our architecture does not reach the high processing speeds afforded by the i7, which remains superior in handling complex computations and intensive data processing tasks. This shortfall highlights the limitations of our current design when facing the demands of high-performance computing environments.

When comparing the execution time the total networks that combine multiple deconvolution layers, the result is different since there are intervals of communication time between PS and PL. The computation time for the whole system depends on Equation 5.1, 5.2, 5.3, 5.4.

Table 5.6 shows the latency of the whole network using FPGA core. Most of the latency is computed due to the DMA transferring time.

Table 5.6: DMA Execution time for the Deconvolution Neural Networks (not including Batch Normalization and Activation Function

Dataset	Execution time with DMA	Execution time without DMA
MNIST	0.004129	0.00259
Celeb-A	0.007367	0.00524

Table 5.7: Execution time for the Deconvolution Neural Networks between different platforms (not including Batch Normalization and Activation Function

Platforms/Datasets	MNIST	Celeb-A
FPGA	0.004129	0.007367
ARM-Cortex A53	0.0125	0.0234
Intel Core i3	0.01396	0.00716
Intel Core i7	0.0024	0.002934

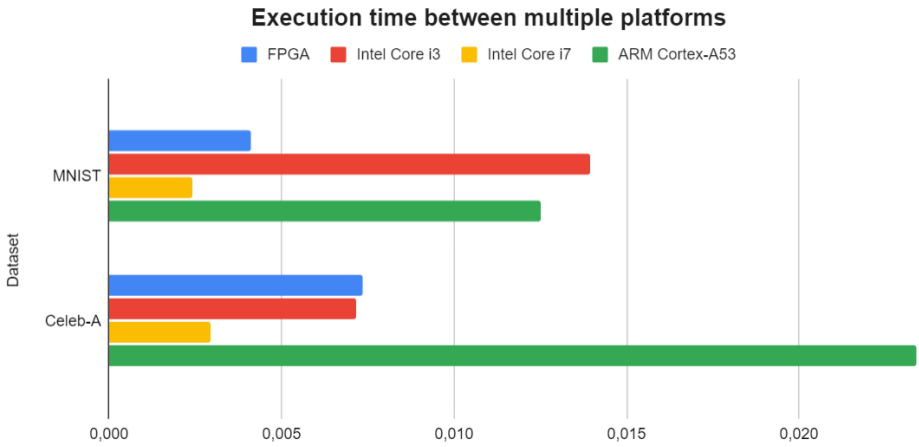


Figure 5.7: Execution time of Deconvolution Neural Networks for 2 datasets between multiple platforms

Moving forward, it is clear that while our architecture presents a promising foundation, further enhancements are necessary to bridge the performance gaps identified during testing. Future developments will focus on refining the design to improve its performance across all types of computational layers, with particular attention to optimizing for high-end CPU comparisons. Additionally, exploring advanced technological integrations and algorithmic optimizations may provide the breakthroughs needed to elevate our architecture's capabilities to compete with and potentially surpass high-performance CPUs like the Intel Core i7.

Figure 5.7 and Table 5.7 illustrates the disparity in execution time across different platforms. Our architecture demonstrates superior performance with the pure ARM Cortex A53, achieving less than a threefold and fourfold increase in speed compared to the MNIST and Celeb-A datasets, respectively. Additionally, while our architecture may not match the performance of the Intel Core i3 in certain layers with large inputs, the overall execution time remains less

than double that of the MNIST dataset and significantly slightly higher for the Celeb-A dataset.

In conclusion, our architecture holds significant promise, particularly for energy-efficient applications, but requires further development to fully realize its potential across a broader range of computing environments.

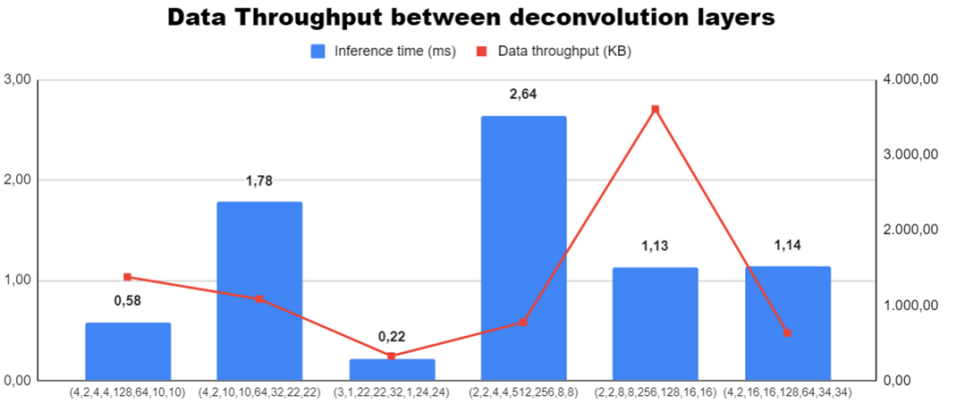
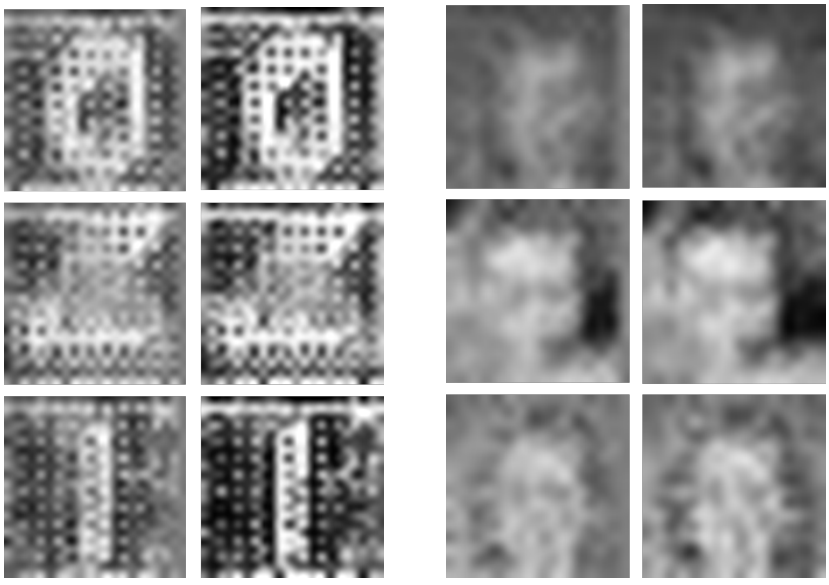


Figure 5.8: Data throughput and inference time of acceleration core correlation

In Figure 5.8, the representation of data throughput for the acceleration core is denoted in *KBps* units. The range of data throughput varies for each layer configuration. Nonetheless, the hardware processor's maximum data throughput can reach up to **3.6 MBps**, with an average of **1.3 MBps**.

5.5.3. IMAGE GENERATION QUALITY

This subsection details the results of image generation, showcasing side-by-side comparisons of outputs produced by the FPGA and software implementations.



(a) (b)

Figure 5.9: Comparison of output image between two datasets on single layer

The results from figure 5.9 show that the FPGA setup does a good job of capturing and keeping the main features of the input data, similar to the original software model. This proves that the FPGA can handle the essential tasks needed by our application well. We confirmed this by visually comparing the feature maps produced by both the FPGA and the software.

However, we noticed that the FPGA does not preserve details as well as the software model. This issue mainly comes from the quantization process used in the FPGA, which helps save on resources and energy but at the cost of some data loss. This data loss is especially noticeable in the finer details of the feature maps, where the lack of precision affects the representation's accuracy.

RESULTS ON MNIST

Figure 5.10: Comparison of MNIST images: the left side shows images generated by the FPGA, and the right side shows the generated images on software.

Has been investigated in Subsection 4.7.1, FID proves as the most compatible metric to evaluate the perceptual quality and diversity of generated images. Additionally, FID tends to be more robust than other metrics such as pixel-wise Mean Squared Error (MSE) or Structural Similarity Index (SSIM) because it considers the high-level features extracted by the Inception network rather than pixel-level differences.

Table 5.8 shows the comparative results between FPGA and its baseline (CPU-based). FPGA-based acceleration core shows a downgraded image quality by around 5.6 times compared to software-based. This is because the data quantization from 32-bit floating point to 8-bit fixed-point between multiple layers in a network.

However, as the visualization aspects, our acceleration core successfully generated the images for MNIST digit dataset by keeping the main features of the images.

Table 5.8: FID scores for FPGA generated-image and software training

	Data type	**FID Score**
CPU	32-bit floating-point	12.453
FPGA	8-bit fixed-point	70.5

RESULTS ON CELEB-A

Figure 5.11: Comparison of Celeb-A images: the left side shows images generated by the FPGA, and the right side shows the generated images on software.

OVERALL SYSTEM EVALUATION

- **MNIST Digit Generation:** The results from both the FPGA and software systems for generating MNIST digits are nearly indistinguishable. This equivalence in performance suggests that both platforms are capable of handling simpler image processing tasks with high accuracy and reliability. Therefore, for applications involving straightforward image generation like digit recognition, both FPGA and software solutions can be considered effective and efficient.

- **Complex Image Generation (Celeb-A Dataset):** The distinction becomes more pronounced when generating complex images such as human faces from the Celeb-A dataset. The software-based system that produced the sharper and more detailed image.The FPGA system, while still capable, appears to generate slightly less detailed and potentially lower-resolution images.

- **Application Suitability:** The FPGA, although slightly less capable in this instance, could still be suitable for lower-power or real-time applications where speed and efficiency are more critical than ultimate image detail.

5.6. STATE-OF-THE-ART COMPARISONS

Based on the data presented in the table, our FPGA implementation exhibits a robust performance balance between resource utilization and efficiency. Compared to other implementations like those by Zhang X., Liu S., and Di X., our design optimizes the use of LUTs, FFs, DSPs, and BRAMs to achieve a notably higher GOPs of 149.7 at a reduced dynamic power consumption of only 4.2 Watts. Specifically, our design utilizes fewer LUTs and FFs than Di X.'s 196.7k and 148.6k respectively, yet delivers more than 10% additional GOPs performance compared to Di X.'s 133.8. Moreover, the power efficiency of our archi-

	LUTs	FFs	BRAMs [MB]	DSPs	Frequency [MHz]	GOPs	Dyn. Power [Watt]
Zhang X. [19]	25.5k	30.9k	2.35	220	100	2.6	NA
Liu S. [37]	161.8k	148.6k	15.3	810	150	NA	NA
Di X. [38]	196.7k	NA	10.9	603	167	133.8	5.8
Ours	**172.6k**	**103.2k**	**8.2**	**734**	**155**	**149.7**	**4.2**

Table 5.9: Comparison of FPGA implementations

tecture is particularly highlighted when contrasted with Di X.'s consumption of 5.8 Watts, demonstrating our implementation's ability to deliver superior performance while consuming approximately 28

5.7. CONCLUSION

The results of our experiment highlight the significant benefits of adopting a hybrid hardware-software approach. This architecture accelerates inference, reduces resource consumption, and maintains high image quality.

- **Accelerated Inference Time:** The hybrid approach significantly reduced inference time, essential for real-time applications.
- **Resource Efficiency:** Offloading tasks to hardware reduced resource consumption, freeing up the main processor for other tasks.
- **Preserved Image Quality:** Despite faster inference, image quality remained high due to optimized resource allocation.
- **Scalability:** This architecture minimizes computational variance and is scalable by integrating additional layers into hardware.

The hybrid architecture is particularly beneficial in fields requiring fast, high-quality image generation, such as computer vision, generative art, and simulation. Although the current implementation is limited to a single layer, extending this approach to other layers promises comprehensive improvements across the entire network. Further optimizations in hardware will yield even greater gains, making this approach highly applicable for performance-critical machine learning tasks.

In conclusion, the hybrid architecture successfully harnesses the strengths of both hardware and software. It delivers a compelling solution that balances performance, resource efficiency, and quality, laying the groundwork for future developments in high-performance GAN systems.

6

CONCLUSION

Deep learning algorithms have been increasingly utilized across various applications in recent years, ranging from high-performance computing to edge computing. One popular algorithm that has gained widespread use in many fields is Generative Adversarial Networks for the application of image generation and super-resolution applications. However, due to its computational and resource-intensive nature, especially for large datasets, its edge computing has been limited. This limit is because edge computing devices often have limited resources and computing power, making implementing the deconvolution algorithm more challenging.

To address this challenge, we introduce a meticulously designed hardware architecture tailored for deconvolutional layers, parameterized and deeply optimized to enhance performance. Our proposal architecture offers hardware templates featuring configurable parameters for different types of layers and datasets, facilitating seamless integration with diverse Generative Adversarial Network (GAN) models and streamlining designer workflow. We also implement a new tilling method that helps to manage data flow from the Processing System to Programmable Logic through many layers without changing the pre-processing data method. Additionally, we developed pipeline Deconvolution Multi-kernel Processors that efficiently compute deconvolution operations, which are the main component of the deconvolution algorithm. Utilizing FPGA-based design alongside the Deconvolution Multi-kernel Processors enables us to realize substantial enhancements in throughput speed compared to conventional computing platforms.

Our findings reveal a substantial improvement in inference speed when utilizing a hybrid process on FPGA boards like ZCU106. In some configurations,

the speed-up exceeds a factor of 2, and in specific cases, it's even three times faster compared to executing the same task on a standalone CPU such as the ARM Cortex-A53.

While our design showcases impressive performance gains, it still falls short of matching the computational efficiency of the highest-performing CPU series. Despite nearly equaling the computation time of lower-performance CPU series, it has yet to reach the same level of efficiency as the top-tier CPU models.

However, there are still a lot of potential aspects that can be improved. Firstly, a detailed analysis can be conducted to identify the bottlenecks within the design. This involves examining the various stages of processing at every checkpoint where delays occur, particularly within the combination circuit. By strategically incorporating additional pipeline stages, the aim is to mitigate these timing critical paths, thereby optimizing the overall performance of the system. Secondly, interrupt signals can be implemented to circumvent the need for the software on the ARM processor to continuously poll for updates which allows the processor to be notified instantly when specific events occur. Thirdly, DMA loading can be upgraded to load the set of weights while the core is processing with the current weights by controlling the status of internal hardware components. This requires more space to pipeline the DMA and involves integrating filters prior to each Multiplier IP, which preprocesses incoming data for execution. These filters serve to eliminate unnecessary multiplication operations between zero and non-zero values, thereby optimizing computational efficiency.

BIBLIOGRAPHY

[1] Xilinx, "Kria kv260 vision ai starter kit." https://www.xilinx.com/products/som/kria/kv260-vision-starter-kit.html. visited on November 6, 2022.

[2] Xilinx, "Zynq ultrascale+ mpsoc zcu106 evaluation kit." https://www.xilinx.com/products/boards-and-kits/zcu106.html. visited on April, 2024.

[3] C. Ledig, L. Theis, F. Huszar, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang, and W. Shi, "Photo-realistic single image super-resolution using a generative adversarial network," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[4] K. K.-D. Ian Colbert, Jake Daly and S. Das, "A competitive edge: Can fpgas beat gpus at dcnn inference acceleration in resource-limited edge computing applications," pp. 1–4, 2021.

[5] A. Pandit, "Introduction to fpga and it's programming tools." https://circuitdigest.com/tutorial/what-is-fpga-introduction-and-programming-tools. visited on November 6, 2022.

[6] L. Crockett, D. Northcote, and C. Ramsay, *Exploring Zynq MPSoC: With PYNQ and Machine Learning Applications*. Strathclyde Academic Media, 2019.

[7] PYNQ, "Pynq: Python productivity." http://www.pynq.io/. visited on November 6, 2023.

[8] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, p. 85–117, Jan. 2015.

[9] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," 2015.

[10] R. M. Schmidt, "Recurrent neural networks (rnns): A gentle introduction and overview," 2019.

[11] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," 2022.

[12] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. C. Courville, and Y. Bengio, "Generative adversarial nets," in *International Conference on Neural Information Processing Systems*, p. 2672–2680, 2014.

[13] C. Ye, M. Evanusa, H. He, A. Mitrokhin, T. Goldstein, J. A. Yorke, C. Fermüller, and Y. Aloimonos, "Network deconvolution," 2020.

[14] C. Ledig, L. Theis, F. Huszár, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang, and W. Shi, "Photo-realistic single image super-resolution using a generative adversarial network," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 105–114, 2017.

[15] R. Sood, B. Topiwala, K. Choutagunta, R. Sood, and M. Rusu, "An application of generative adversarial networks for super resolution medical imaging," in *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pp. 326–331, 2018.

[16] S. Liu, C. Zeng, H. Fan, H.-C. Ng, J. Meng, Z. Que, X. Niu, and W. Luk, "Memory-efficient architecture for accelerating generative networks on fpga," in *2018 International Conference on Field-Programmable Technology (FPT)*, pp. 30–37, 2018.

[17] A. Yazdanbakhsh, M. Brzozowski, B. Khaleghi, S. Ghodrati, K. Samadi, N. S. Kim, and H. Esmaeilzadeh, "Flexigan: An end-to-end solution for fpga acceleration of generative adversarial networks," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 65–72, 2018.

[18] J.-W. Chang and S.-J. Kang, "Optimizing fpga-based convolutional neural networks accelerator for image super-resolution," in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 343–348, 2018.

[19] X. Zhang, S. Das, O. Neopane, and K. Kreutz-Delgado, "A design methodology for efficient implementation of deconvolutional neural networks on an fpga," 2017.

[20] G. Li, Z. Liu, F. Li, and J. Cheng, "Block convolution: Toward memory-efficient inference of large-scale cnns on fpga," *IEEE Transactions on*

Computer-Aided Design of Integrated Circuits and Systems, vol. 41, no. 5, pp. 1436–1447, 2022.

[21] D. Xu, K. Tu, Y. Wang, C. Liu, B. He, and H. Li, "Fcn-engine: Accelerating deconvolutional layers in classic cnn processors," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–6, 2018.

[22] L. Bai, Y. Lyu, and X. Huang, "A unified hardware architecture for convolutions and deconvolutions in cnn," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, 2020.

[23] T. Pham-Dinh, B. Bach-Gia, L. Luu-Trinh, M. Nguyen-Dinh, H. Pham-Duc, K. Bui-Anh, X.-Q. Nguyen, and C. Pham-Quoc, "An fpga-based solution for convolution operation acceleration," in *Intelligence of Things: Technologies and Applications* (N.-T. Nguyen, N.-N. Dao, Q.-D. Pham, and H. A. Le, eds.), (Cham), pp. 279–288, Springer International Publishing, 2022.

[24] Xilinx, "Axi central direct memory access logicore ip product guide." `https://docs.xilinx.com/r/en-US/pg034-axi-cdma/CDMA-Lite-Mode-Restrictions`. visited on April 24, 2024.

[25] AMBA, "Apb amba protocol specification." `https://developer.arm.com/documentation/ihi0024/latest/`. visited on May 9, 2024.

[26] Q. Xu, G. Huang, Y. Yuan, C. Guo, Y. Sun, F. Wu, and K. Weinberger, "An empirical study on evaluation metrics of generative adversarial networks," 2018.

[27] S. Barratt and R. Sharma, "A note on the inception score," 2018.

[28] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter, "Gans trained by a two time-scale update rule converge to a local nash equilibrium," in *Advances in Neural Information Processing Systems* (I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds.), vol. 30, Curran Associates, Inc., 2017.

[29] K. Pearson, "Liii. on lines and planes of closest fit to systems of points in space," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 2, no. 11, pp. 559–572, 1901.

[30] A. Sanakoyeu, D. Kotovenko, S. Lang, and B. Ommer, "A style-aware content loss for real-time hd style transfer," 2018.

[31] J. Johnson, A. Alahi, and L. Fei-Fei, "Perceptual losses for real-time style transfer and super-resolution," 2016.

[32] J. Nilsson and T. Akenine-Möller, "Understanding ssim," 2020.

[33] Q. Nguyen-Xuan and C. Pham-Quoc, "An fpga-based convolution ip core for deep neural networks acceleration," *REV Journal on Electronics and Communications*, vol. 12, 05 2022.

[34] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[35] Z. Liu, P. Luo, X. Wang, and X. Tang, "Deep learning face attributes in the wild," 2015.

[36] E. Agustsson and R. Timofte, "Ntire 2017 challenge on single image super-resolution: Dataset and study," in *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pp. 1122–1131, 2017.

[37] S. Liu, C. Zeng, H. Fan, H.-C. Ng, J. Meng, Z. Que, X. Niu, and W. Luk, "Memory-efficient architecture for accelerating generative networks on fpga," in *International Conference on Field-Programmable Technology*, 2018.

[38] X. Di, H.-G. Yang, Y. Jia, Z. Huang, and N. Mao, "Exploring efficient acceleration architecture for winograd-transformed transposed convolution of gans on fpgas," *Electronics*, vol. 9, no. 2, 2020.