Deploying Image Super Resolution Deep Learning Models in ZYNQ ULTRASCALE+ MPSoC

Gokula Krishnan Ravi

Abstract:

Zynq Ultrascale+ MPSoC ZCU102 offers high flexibility in order to provide solutions for varied applications including, Flight Navigation, Networking, Machine Vision etc. For applications that demand deep learning solution, Xilinx offers Deep Learning Processor Unit (DPU), a soft core co-processor that can be integrated in order to deploy deep learning models into programming logic. Leveraging this IP core, Image Super-Resolution Deep Learning Models were tested for feasibility. Image Super Resolution is a process of estimating High Resolution Images from a given Low Resolution image. A collection of State-of-the-art models were reviewed and as GAN based models trained with perceptual losses produced good results, two pre-trained models (Residual Dense Network (RDN) and Enhanced Super Resolution generative adversarial networks (ESRGAN), trained on GAN based setting were tested for deployment. An application that uses DPU IP was developed and models with different input image patch sizes were deployed. It can be observed that the colour tone of super resolved output patches from RDN model was not even and saturated. On the other hand, ESRGAN uses layer that are not supported by DPU IP and few work arounds were tried and resulted in failure. A few improvements like Knowledge distillation, use of multiple DPUs and alternative up sampling networks were suggested.

Table of Contents

List of Tables

List of Figures

Abbreviations

AI	Artificial Intelligence
AMBA	Advanced Memory Bus Architecture
API	Application Programming Interface
APU	Application Processing Unit
AXI	AMBA eXtensible Interface
BN	Batch Normalization
CNN	Convolutional Neural Network
DDR	Double Data Rate
DPU	Deep Learning Processor Unit
FPGA	Field Programmable Gate Array
FR	Fully Reference
GAN	Generative Adversarial Network
GFF	Global Feature Fusion
GPIO	General Purpose Input Output
GRL	Global Residual Learning
GTR	Gigabit Transceiver
HR	High Resolution
IP	Intellectual Property
LFF	Local Feature Fusion
LRL	Local Residual Learning
LR	Low Resolution
LUT	Look Up Table
MAC	Multiply Accumulate
MOPS	Million OperationS
MPSoC	Multi-Processor System on Chip
NIQE	Naturalness Image Quality Evaluator
NR	No Reference
PSNR	Peak Signal to noise Ratio

RDN	Residual Dense Network
RRDN	Residual-in-Residual Dense Network
RTPU	Real Time Processing Unit
SDK	Software Development Kit
SFF	Shallow Feature Fusion
SFM	Softmax
SR	Super Resolution
SSIM	Structual Similarity
TCM	Tightly Coupled Memory
UART	Universal Asynchronous Receiver Transmitter
CAN	Controller Area Network
USB	Universal Serial Bus

CHAPTER 1

INTRODUCTION

"Making Machines to perceive, analyse and act upon the world" has been a long life dream for inventors, researchers all over the world. Earlier, Machines were made to solve problems with finite search space, with minimal knowledge in the environment they are acting upon. For example, in 1997, IBM's Deep Blue defeated Gary Kasparov, the World champion in Chess. Situations like these, where it's hard for humans, are a piece-of-cake for Machines to solve. Now-a-days, Machines were made to solve problem that are part of day-to-day activities of humans. Solving such problems, need a tremendous amount of knowledge about the environment. Earlier methods, called symbolic AI, use the concept of Knowledge bases, where every piece of information is considered a symbol that a machine could interpret and a machine is hardcoded with the information about a task, its environment and its actions. Logically, though this type of knowledge representation yields good results for finite search space tasks like chess, it is difficult to hard code situations like speech or vision in this manner. This problem of hard-coded knowledge, led to a mechanism where machines were made to interpret and extract patterns from raw data. This mechanism is called Machine Learning. Machine learning algorithm maps given input and output data. Through invent of machine learning, a wide range of regression and classification problems could be solved. For example, weather prediction, email spam classification etc. The performance of machine learning algorithms highly depends on the feature representations of a given phenomenon. A feature is termed as an individual measurable property that represents a phenomenon. A machine learning problem could be solved by extracting right set of features from the task and feeding the features to map them to outputs. Extracting the right set of features for a given task could be challenging. One such task is Computer Vision. Computer Vision is the field of study, to develop solutions to understand a given image or video. For example, given a picture of a car, humans can easily identify it. But for a computer, it should learn that a car should have four wheels. But only given the pixel information, it is very difficult to recognise that the image has four wheels, because the image might be taken in different orientation or because of some intentional or unintentional colour variation. Therefore, feature learning is much more important for these types of tasks under varying situations [1] .

1.1 DEEP NEURAL NETWORKS

Deep Learning solves the problems associated with of feature learning of varying phenomena. Deep Learning is a subset set of Machine learning, which is mainly used to extract unknown useful features to represent a given data. Deep learning achieves this extent of learning representations by use of Neural Networks. Neural Networks / Multi-layer Perceptron, inspired by the structure of human brain, consist of interconnected layers, with elements, each called a neuron Figure 1.1. A neuron computes weighted summation / linear combination of inputs and gets

"activated" if the sum exceeds a threshold. The threshold function can be modelled by activation functions like sigmoid, Relu, tanh etc. Subjectively, a neural network consists of a large number of simple functions which collectively contribute to map a given input to output. This makes a neural network to map any kind of data by extracting unknown useful features. Because of mapping over wide variety of datasets, Deep Neural Networks are also termed as Universal Approximators [2]

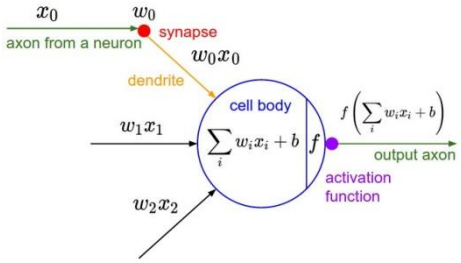

Figure 1.1 A Neuron

Neural Networks are trained based maximum likelihood estimation of weights and biases. Losses are used to steer the neural network to converge at optimum.

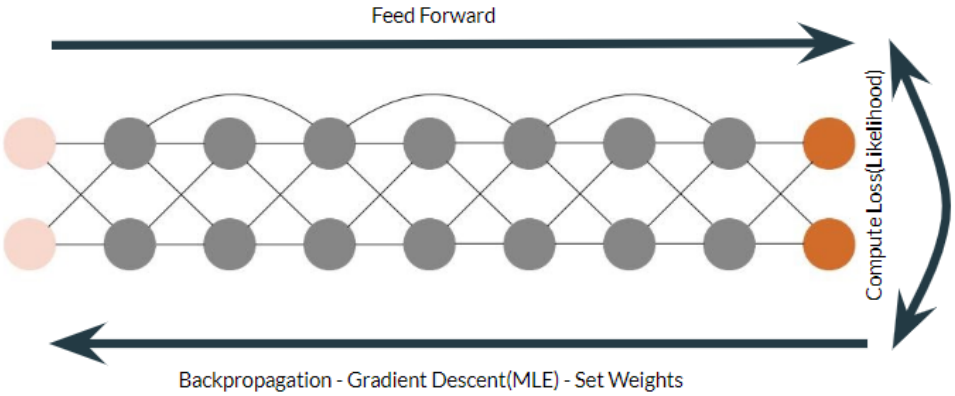

Figure 1.2 Training a Neural Network

For computer vision tasks more sophisticated type of layer called Convolutional layer is used. Convolutional layer takes an image as-is and outputs a 3-D layer of activations. With the use of variants in Convolutional layer an image can be down sampled, up sampled, and features can be learned in any specific dimension that suits a problem. In practice several Convolutional Layers are used in order to extract more useful features for Image oriented tasks.

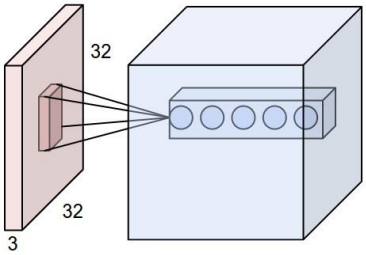

Figure 1.3 A Convolutional Layer

1.2 IMAGE SUPER RESOLUTION

Image Super-Resolution is a field of Computer Vision or Image Processing, where the goal is to make an input image clearer. In other words, given a Low Resolution (LR) Image, the problem is to estimate a plausible High Resolution (HR) Image, filling the missing details of LR image. It is one of the Image Restoration problems like Image de-noising, Image de-blurring tasks. It has found variety of applications for medical Imaging [3], computational photography in mobile phones [4], surveillance [5] etc. Image Super Resolution is an ill posed problem, i.e. for a Low Resolution image; there are infinite possible High Resolution images. There exist a variety of classical solutions to this problem, including patch based, edge-based methods, sparse representations, statistical methods etc. With the evolution of deep learning models for Image oriented tasks with high accuracies, developing Deep learning models for Super Resolution have been widely addressed and the models often produce high quality results as compared to classical Super Resolution Algorithms.

Figure 1.4 An example of Super Resolution

1.3 ZYNQ ULTRASCALE+ MPSoC – ZCU102

Deploying Deep Learning Models in Xilinx based edge embedded systems is discussed here. To start with, Zynq Ultrascale+ MPSoC is ZCU102 is tested.

1.3.1 Tools

Vitis – Earlier known as Xilinx SDK, is used for application development. Version 2019.2

Vivado – Tool to design and interface IPs in Programming Logic and interface it with Processing Sytem of ZCU102

Petalinux- Tool to build a Linux Operating system with respect to the designed Hardware

Vitis AI SDK – A docker container that contains all necessary tools to deploy deep learning models in ZCU102

1.3.2 Architecture

ZCU102 is an EG device with a speed grade -2 [6]. ZCU102 offers high flexibility in order to provide solutions for varied applications including, Flight Navigation, Networking, Machine Vision etc. It consists of two parts 1) Processing System (PS) 2) Programming Logic (PL)

Processing System consists of

- APU - Quad-core Arm Cortex-A53 MPCore with CoreSight; NEON & Single/Double Precision Floating Point; 32KB/32KB L1 Cache, 1MB L2 Cache
- RTPU - Dual-core Arm Cortex-R5 with CoreSight; Single/Double Precision Floating Point; 32KB/32KB L1 Cache, and TCM
- Memory embedded and external- 256KB On-Chip Memory w/ECC; External DDR4; DDR3; DDR3L; LPDDR4; LPDDR3; External Quad-SPI; NAND; eMMC
- 214 PS I/O; UART; CAN; USB 2.0; I2C; SPI; 32b GPIO; Real Time Clock; WatchDog Timers; Triple Timer Counters
- 4 PS-GTR; PCIe Gen1/2; Serial ATA 3.1; DisplayPort 1.2a; USB 3.0; SGMI
- Arm Mali™-400 MP2; 64KB L2 Cache

Programming Logic is the FPGA (Field Programmable Gate Array), reconfigurable Hardware which is interfaced with PS. It has the following features:

Table 1.1 Specifications of Programmable Logic in ZCU102

System Logic Cells	599550
Configurable Logic Block Flip Flop	548160
Configurable Logic Block LUT	274080
Distributed RAM (MB)	8.8
Block RAM blocks	912
Block RAM (MB)	32.1
DSP Slices	2520
Clock Management Tools	4
Max. HP I/O	208
Max. HD I/O	120
System Monitor	2
GTH Tx 16.3Gb/s	24
GTY Tx 32.75 Gb/s	0
Transceiver Fractional PLLs	12

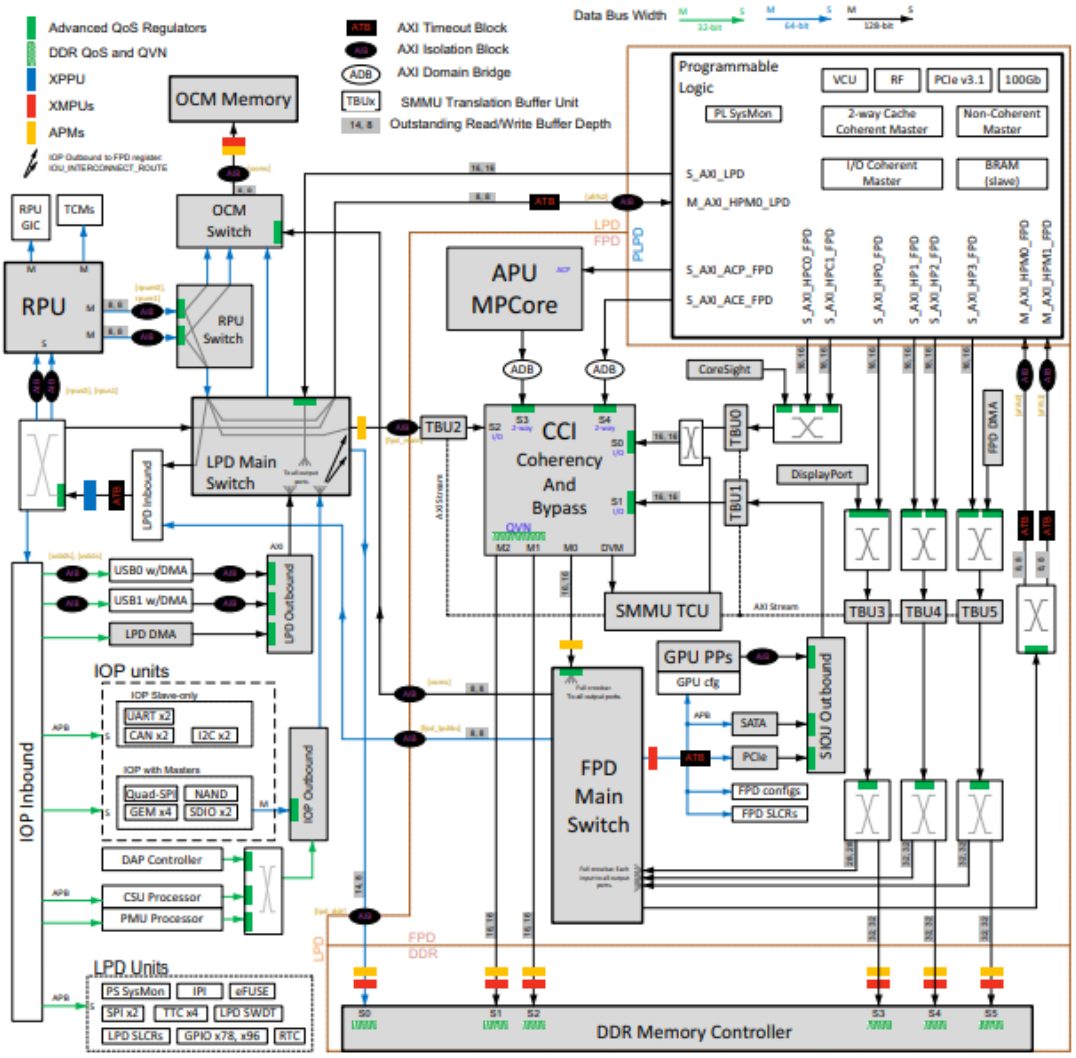

Figure 1.5 Achitecture of ZCU102 [6]

The PS-PL architecture use shared memory model, where data is shared via DDR memory between PS-PL. The data is shared via AXI (AMBA Advanced eXtensible interface) buses.

1.4 DEEP LEARNING PROCESSING UNIT

Deep Learning Processing Unit (DPU) is a soft core co-processor IP developed by Xilinx for deploying Convolutional Neural Network (CNN) based models. This section highlights some key aspects of DPU and integration of DPU with processing system using Vivado. The DPU IP supports up to 3 DPUs to run in parallel. A wide range of options are available for configuring DPU [7]

1.4.1 Design

The following design Figure 1.6 follows the reference design given in [7], instead, only one DPU is used in this implementation. After creating the design, Petalinux is used to configure ROOTFS and to generate BOOT.BIN. The device tree is similar to reference design and the interrupt numbers are changed accordingly.

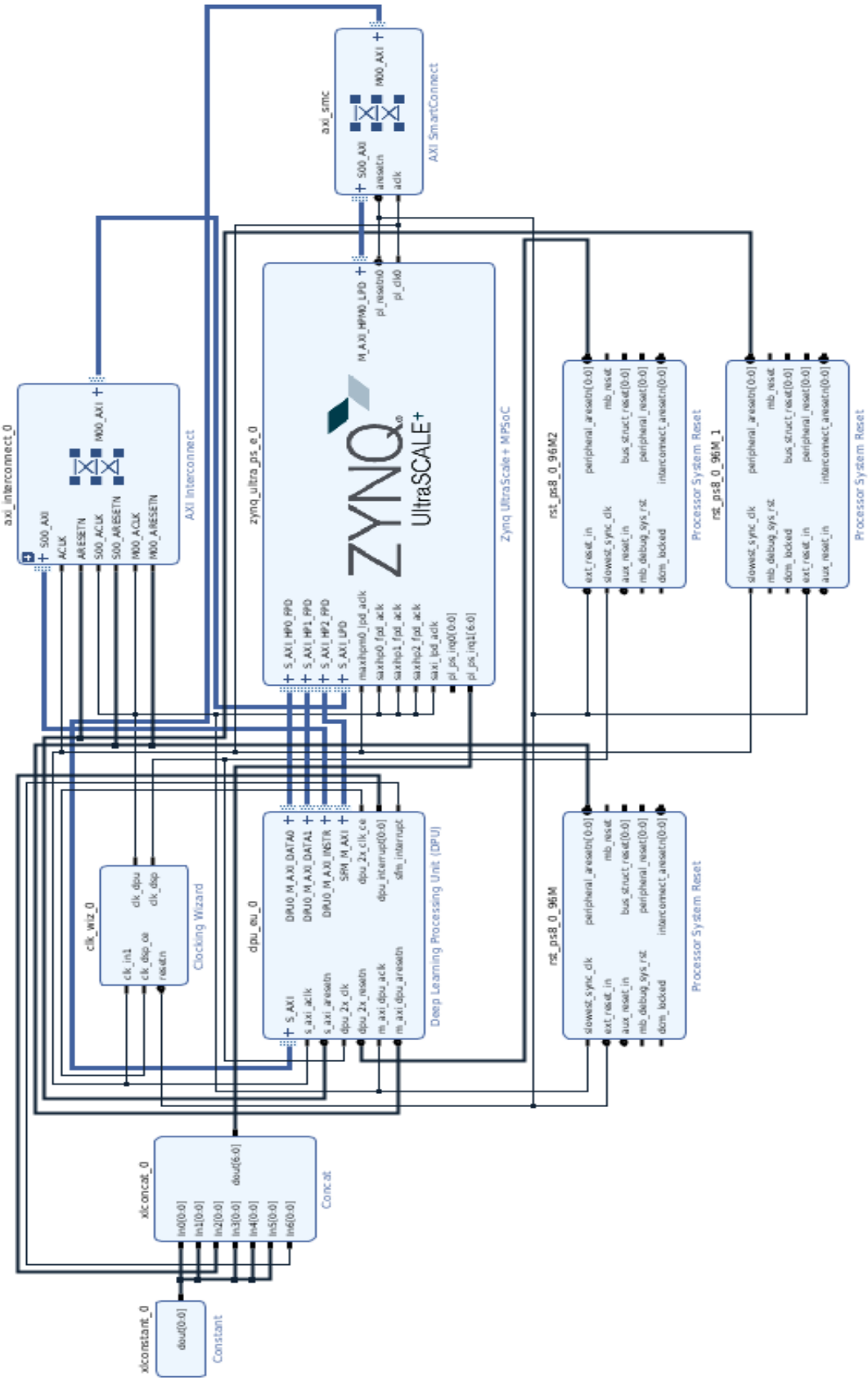

Figure 1.6 Block Design of Hardware

1.4.2 Specification

Table 1.2 Configuration of DPU

Number of DPU cores	1
Arch of DPU	B4096
RAM Usage	Low
Channel Augmentation	Enabled
DepthWiseConv	Enabled
AveragePool	Enabled
Relu Type	Relu+ LeakyRelu+Relu6
Number of SFM (softmax) core	1
S-AXI Clock Mode	Independent
Dpu_2x Clock Gating	Enabled
DSP48 Maximal Cascade Length	4
DSP Usage	High
Ultra-RAM per DPU	0
Target version	1.4.0
AXI Protocol	AXI4
S-AXI Data Width	32
M-AXI GP Data Width	32
M-AXI HP Data Width (DPU)	128
M-AXI HP Data Width (SFM)	128
M-AXI ID Width	2
DSP Slice Count	704
Ultra-RAM Count	0
Block-RAM Count	261.0

Table 1.3 Clocking Configuration

Clock	**Name**	**Output Frequency (MHz) Requested**	**Output Frequency (MHz) Actual**	**Phase (degrees)**	**Duty Cycle (%)**	**Drives**	**Matched Routing**
clk_out1	clk_dsp	650	649.9	0	50	Buffer with CE	yes
clk_out2	clk_dpu	325	324.9	0	50	buffer	yes

1.5 MOTIVATION

Apart from advantages of deploying deep learning models in edge, like low latency, secure operation etc, with presence of FPGA along with arm processor core, Zynq Ultrascale+ MPSoC can be used to integrate various modules and IPs flexibly to offer suitable solution for a specifc task and hence, for problems that require Deep learning solution, DPU can be easily integrated with a suitable design. This is unlike other edge devices, where a separate Deep Learning accelerator needs to be used.

CHAPTER 2

LITERATURE REVIEW

This review broadly discusses on deploying deep learning models in Zynq UltraScale+ MPSoC and Image Super-Resolution Deep Learning Model selection.

2.1 DEPLOYING DEEP LEARNING MODELS IN ZYNQ MPSOC

Vitis AI SDK [8] is used for deploying Deep learning models in XILINX based ZYNQ MPSoC. To deploy a deep learning model, a set of steps are to be followed. These steps are inspired from the methodology proposed by **Han et al** [9] in the paper *Deep Compression*.

Vitis AI supports two Deep Learning frameworks, 1) Tensorflow [10] 2) Caffe [11]. Since Tensorflow has a large community base than Caffe, implementation in Tensorflow is discussed here. Also, a higher level API framework for Tensorflow: Keras is used in the implementation of Deep Learning models and will be used here.

2.1.1 Deployment Methodology

Vitis AI model deployment methodology gives an intuitive way to software developers to deploy a model in FPGA, without having a prior knowledge on technical aspects of FPGA. Vitis AI SDK gives a set of tools to transform a deep learning model to a library package that can be included in C++ or Python application. To deploy a model in Zynq UltraScale+ MPSoC, one has to perform the following steps:

- Train a model constructed with Vitis supported layers (discussed below) using Keras with Tensorflow backend. Versions Supported: v1.12.0 and v.1.15.0
- Generate Model Checkpoints, along with calling tf.keras.backend.set_learning_phase(0)
- Prune the model using Vitis AI Pruner tool. This step is optional and can be used for larger models, if needed
- Quantize the model using Vitis AI Quantisation tool. This is a mandatory step, without which, the model cannot be deployed.
- Compile the model to obtain a .elf file. Compilation is the process of creating instructions for DPU. The .elf file contains model information and the operations that are needed to be performed by the DPU
- For debugging purposes Vitis AI profiler can be used to get layer wise information after compilation. Profiler can also be used while runtime
- Convert the .elf file to a .so Shared library file using GCC toolchain for cross compilation
- Copy the shared library file to /usr/lib in the hardware platform

2.1.2 Vitis Supported Layers

Though the methodology is intuitive, it is important to note the layers supported by Vitis AI tools. Tensorflow / Keras layers are discussed here. The following table [8] Table 2.1 lists the layers and operations supported by Vitis

Table 2.1 Supported Tensorflow and Keras layers for Vitis

Type	Operation Type	tf.nn	tf.layers	tf.keras.layers
Convolution	Conv2D DepthwiseConv2dNative	atrous_conv2d conv2d conv2d_transpose depthwise_conv2d_native separable_conv2d	Conv2D Conv2Dtranspose SeperableConv2D	Conv2D Conv2DTranspose DepthwiseConv2D SeperableConv2D
Fully Connected	MatMul	/	Dense	Dense
BiasAdd	BuasAdd Add	bias_add	/	/
Polling	AvgPool Mean MaxPool	avg_pool max_pool	Averagepooling2D MaxPooling2D	AveragePooling2D MaxPool2D
Activation	Relu Relu6	relu relu6 leaky_relu	/	ReLU LeakyReLU

Table 2.1 Contd.

Type	Operation Type	tf.nn	tf.layers	tf.keras.layers
BatchNorm	FusedBatchNorm	Batch_normalisation Batch_norm_with_global_normalization fused_batch_norm	BatchNormalization	BatchNormalization
Upsampling	ResizeBilinear ResizeNearestNeighbour	/	/	Upsampling2D
Concat	Concat ConcatV2	/	/	Concatenate
Others	Placeholder Const Pad Squeeze Reshape ExpandDims	dropuout softmax	Dropuout Flatten	Input Flatten Reshape Zeropadding2D Sofmax

2.2 MODEL SELECTION

Image Super-Resolution Deep learning models can be discussed broadly based on whether a model is trained using supervised methods or unsupervised methods. Considering low inference latency, unsupervised model tend to have higher latency figures than supervised methods. The reason lies in the methodology itself, as unsupervised methods map input to output at inference [12], while supervised models can be trained with intended accuracy and can be compressed to infer at edge.

Considering supervised models, they can be discussed broadly based on four following classes: 1) Network Architecture, 2) Losses, 3) Metrics and 4) Data.

2.2.1 Network Architecture

Image Super-Resolution, as stated before, is an ill posed problem and the goal is to increase the clarity i.e introduce high frequency components, so that the image looks natural and clear when upsampled. To learn these high frequency components, one has to construct a neural network, so that that the model learns multiple sets of representations/features to reconstruct the image. Architecting a neural network solely depends on the complexity of the problem that one has to solve. Choosing shallow networks might be good option, but it might not guarantee that the model learns the feature representations required for solving the problem. As Image Super- Resolution is a type of problem where clarity is of most importance, one has to devise an architecture that learns different kind of representations of a given image. Therefore, all neural network architectures made for super resolution have larger depth, so that the model learns large set of features to reconstruct the image. But, architecting a deep neural network is not an easy task. Deep neural networks face the problem of exploding/vanishing gradient descent (i.e. exponential increase/decrease in network parameters with increase in depth) and degradation problem (decrease in accuracy with increase in depth)

2.2.1.1 Residual Connections

Consider H(x) to be the mapping function represented by the neural network and x is the input. With the problems mentioned above, deep neural network also face another problem - learning the Identity function (i.e. failing to learn the input or shallow feature representations). To solve these problems, **He et al**. [13] proposed the architecture of residual network (also termed as local residual learning), where in, the model learns the residues, denoted F(x) in the Figure 1.1 rather than just directly learning H(x). Here H(x) = F(x) + x

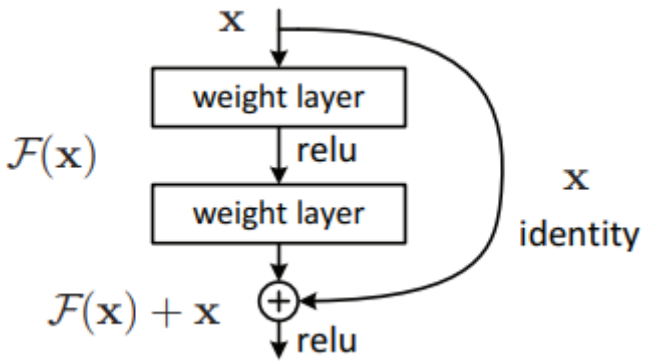

Figure 2.1 A Residue

Residual network learns the identity function because of the skipped connection present in the network. Even if the residues fail to learn, the model will still be able to represent the shallow network, hence, learns the Identity function. By learning Identity function, the depth of the model can be increased with no effect from degradation problem. Residual connections do not add any extra parameters to learning, and hence can be learned using Stochastic Gradient Descent.

Another advantage given by Residual connection is the Global Residual learning. Like local residual learning, Global Residual learning refers to the placement of skipped connections, skipping large number of layers or connecting shallow features with deep features of the network. Global Residual learning has a greater advantage in Image Super-Resolution as a model learns to generate output that is consistent with the input image distribution.

2.2.1.2 Batch Normalization

Though Residual Connections solve the degradation problem, Vanishing / exploding gradients is still a problem. To alleviate the problem, along with stabilising the training of deep neural networks, Batch Normalization (BN) [14] is used. BN refers to normalising intermediate feature representations along with normalising inputs. Though Batch Normalization reduces overfitting, speeds up and stabilizes training, **X.Wang et al** [15] argues that Batch Normalization introduces artefacts.

2.2.1.3 Dense Connections

Dense Connections, as proposed by **Huang et al** [16], reduces model parameter size, enhance signal and gradient propagation, and solves the vanishing/ exploding gradients problem. For a layer in a network that uses dense connections, the feature maps of all preceding layers are used as inputs and the layer's feature representations are used as inputs for subsequent layers. Instead of addition of layers, concatenation along the channel axis is performed. Therefore for a layer L, there are L(L+1)/2 connections up until the layer, unlike other networks where the connections were only L.

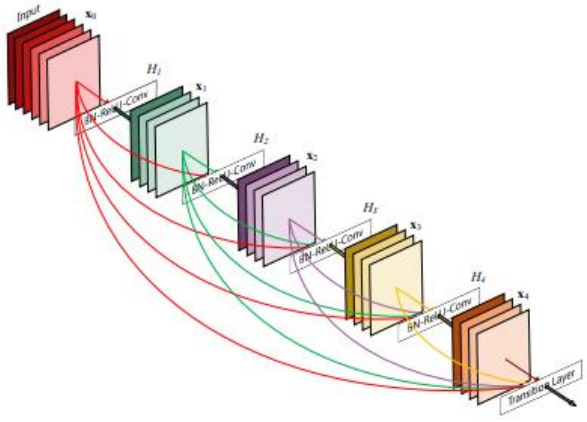

Figure 2.2 Dense Connections [16]

Although, the number of connections increased, the model requires only fewer parameters, as opposed to network architectures like Resnet. This is because of direct flow of information in Densely Connected layers. As all preceding layers for a layer are used as inputs, the layer gets all types of information and the layer just need to add a little amount of extra information. The final Output takes all layers into consideration and hence predicts the output based on collective information rather than abstracted information. As the gradient flow is easier in this architecture, though not explicitly performed, this architecture solves the exploding / vanishing gradient descent.

2.2.1.4 Up Sampling

2.2.1.4.1 Strategy

This section discusses about two strategies used for up sampling a neural network: 1) Pre-Up sampling and 2) Post-Up sampling.

Pre-Up sampling is a way of architecting a model, where input or shallow features are up sampled / interpolated first and then deep neural network is applied to the interpolated image / features. This makes a network to learn the representations from interpolated image to the output. Though it makes the up sampling step easier with conventional up sampling methods, and reducing the learning difficulty, this method introduces unwanted noise amplification and blurring during the process. Also, the time and space required to make the model converge are high [17]

Post-Up sampling, on the other hand, allows a neural network to learn all representations in Low Resolution (LR) space, thus having a good set of information about the input before Up sampling. Also with the invent of learnable Up sampling (i.e Deconvolution / Transposed Convolution [18] and Sub-Pixel network [17]), the neural network can be made learnable end-to-end.

2.2.1.4.2 Up Sampling Methods:

Up sampling methods can be classified into 1) Interpolation based 2) Learning based

Nearest Neighbour Interpolation: Nearest Neighbour is an Interpolation based Up Sampling method, wherein the up sampled pixels will have nearest original pixel values. The associated layer in keras is UpSampling2D.

Bilinear Interpolation: In Bilinear interpolation, Pixel values are linearly interpolated subsequently along the two axes. Bilinear Interpolation has a receptive field of 2x2 pixels. The associated layer in keras is UpSampling2D.

Bicubic Interpolation: Similar to Bilinear Interpolation, In Bicubic interpolation, Pixel values are interpolated using a cubic function, subsequently along the two axes. Bicubic Interpolation has a receptive field of 4x4 pixels, and gives a smoother output than bilinear interpolation.

Deconvolution: Deconvolution or Transposed Convolution or Strided Convolution [18] is a learning based Up Sampling method. Intuitively it is considered an Inverse operation of convolution. In Convolution, a kernel multiplies and adds with the input matrix element wise to produce a single element value in the resulting matrix. While in Transposed Convolution, a single element value in the matrix is multiplied with each element in a kernel of specified size and stride, resulting in a larger matrix.

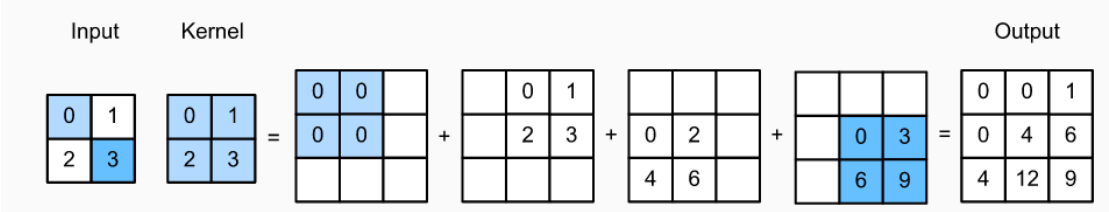

Figure 2.3 Deconvolution

Deconvolution can also denote, up sampling the input features and applying a convolution operation over the up sampled feature.

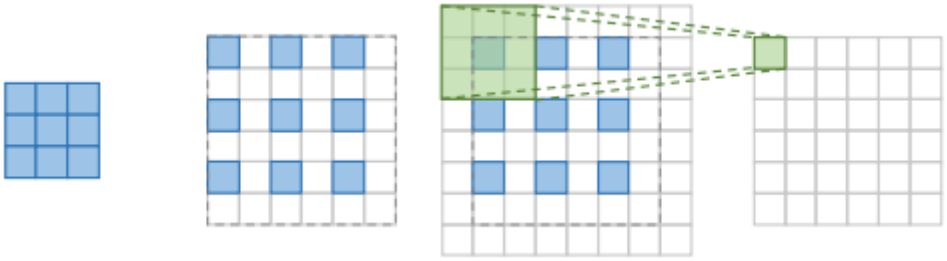

Figure 2.4 Another variant of Deconvolution

Although, Deconvolution gives an end to end approach for up sampling, it faces checker board problem, where the adjacent pixels have uneven pixel intensities.

Sub-Pixel: Efficient Sub-Pixel Convolutional Neural Network (ESPCN) [17] is used for up sampling the features in LR space.

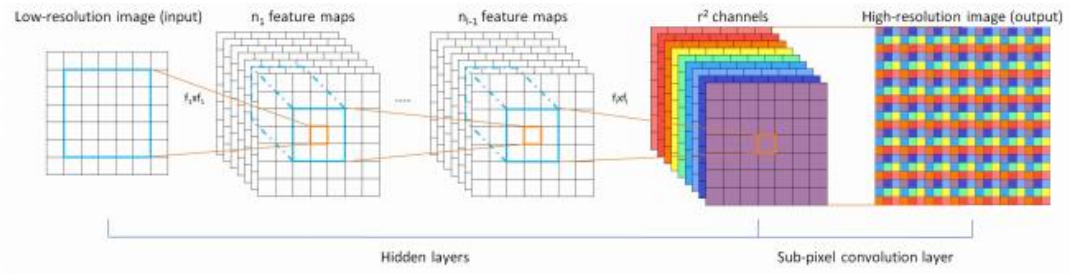

Figure 2.5 Sub-Pixel Network [17]

Super-Resolution in Pixel space is the conversion of an image of dimensions H X W X C to an image of dimensions H*r X W*r X C where H, W, C are Height, Width and Channels of an image, r is the up sampling factor. Unlike conventional image interpolation, in Sub-Pixel layer, there are two stages in up sampling, 1) create r^2

channels from input LR feature space, making a LR feature space of dimensions H X W X r^2C 2) shuffle: reshaping the LR feature space into r*H X r*W X C. The authors argue that this methodology do not introduce artefacts.

PixelTCN: The methods discussed before, Deconvolution and SubPixel layers generate features independent of each other. In other words, those features are a result of convolution of different kernels applied directly with the input of the layer. The authors of PixelTCN [19] – Pixel Transposed Convolutional Neural Network, argue that this independent nature of features produce blocky output. This is also called checker board problem. To alleviate this problem, the authors of PixelTCN propose to generate interdependent features instead. In other words, a feature is generated by convolution operation on juxtaposition of preceding features. This is similar to Dense Connections discussed before. The output is then the shuffling or reshaping of these interdependent features.

2.2.2 Generative Adversarial Networks

Generative Adversarial Networks belong to a class of generative modelling. Generative modelling, in short, is generating new set of data points from the probability distribution of the given dataset. Though Generative Modelling is an unsupervised task, the problem is made discriminative and supervised because of the zero-sum game neural network based generative modelling [20]. A GAN is a combination of generator and discriminator network. As the name suggests, Generator network generates plausible images from random input vector, and discriminator tries to discriminate between original data and the generator data. The goal for the generator is to produce data such that discriminator maximises the likelihood for "fake" data and penalises for real images. For the discriminator, the goal is to maximise the likelihood for real data and penalise the network for fake data. With the invent of Deep Convolutional GANs by [21] and the extension – Conditional GANs, it is now possible to set training data as input and use state-of-art convolution neural network for generating data rather than a random vector.

GANs that use Convolutional Neural Networks have been consistently giving better results for many computer vision tasks, including image to image translation. The goal of using GANs for Image to image translation task has diverged from being pixel wise similar to making the image look more natural. The tendency of an Image looking more natural is called Perceptual Quality. A state of art GAN for Super-Resolution (SRGAN [22]) is the first breakthrough for generating naturally looking images. SRGAN uses Residual blocks, subpixel layer for the generator along with Perceptual and Adversarial loss functions. Using combined loss function makes the generator to generate data that look more natural and at the same time generates images that arise from the same distribution as that of the original data. Later inventions of GAN based networks; EnhanceNet [23] and ESRGAN [15] use similar approach and provide further improvement in Quality.

2.2.3 Losses

This section discusses some of the well-used losses for Deep Learning for Image Super-Resolution. Losses can be categorised into distortion and perceptual based losses. Distortion losses account for pixel wise similarity whereas perceptual losses account for making an image look natural. Consider $\hat{I}$ and I be the Super-Resolved Image and High Resolution Image respectively.

2.2.3.1 Per-Pixel Loss

In most of the Deep Neural Networks for Image Super-Resolution, the following distortion losses: L1 and L2 are used. The use of L1 and L2 losses focus on pixel wise similarity between the real image and the generated image.

$$\mathcal{L}_{l1}\left(\hat{I}, I\right) = \frac{1}{hwc}\sum_{i,j,k}\left|\hat{I}_{i,j,k} - I_{i,j,k}\right| \tag{2.1}$$

$$\mathcal{L}_{l2}\left(\hat{I}, I\right) = \frac{1}{hwc}\sum_{i,j,k}\left(\hat{I}_{i,j,k} - I_{i,j,k}\right)^2 \tag{2.2}$$

hwc is the product of dimensions of the image

Comparing L1 and L2 losses, L2 penalises the model for larger errors and thus produces smoother results than L1. Although, these losses focus on similarity, models trained with these losses, tend to miss high frequency details in the reconstructed image [22].

2.2.3.2 Content Loss

Content loss / Perceptual loss proposed by ***Johnson et al*** [24]**,** compares images based on feature representations of pre-trained neural network. Models trained with this loss tend to be similar in feature representations rather than pixel wise similarity.

$$\mathcal{L}_{percep}\left(\hat{I}, I, \phi, l\right) = \left\|\phi\left(\hat{I}\right) - \phi(I)\right\|_2^2 \tag{2.3}$$

Mathematically, for a given neural network/ feature extractor ϕ, layer l, perceptual loss computes frobenius norm of feature maps of images $\hat{I}$ and I. In practice VGG [25] or Resnet [13] is used as feature extractors. As discussed before, this loss is used in GAN based Super Resolution models for reconstructing images with visually pleasing results.

2.2.3.3 Adversarial Loss

As discussed before, training a GAN can be called as a zero-sum game between generator and discriminator network. A GAN based setting uses the following losses,

$$\mathcal{L}_{generator} = -\log D\left(\hat{I}\right) \tag{2.4}$$

$$\mathcal{L}_{discriminator} = -\log D(I) - \log\left(1 - D\left(\hat{I}\right)\right) \tag{2.5}$$

D(.) is the output of the discriminator.

The generator is trained to minimize equation (2.4) and discriminator is trained to minimize equation (2.5). This loss is used in SRGAN [22] and EnhanceNet [23]for generating plausible naturally looking images. Similarly ESRGAN uses Relativistic Average Loss [26] to create more realistic images.

2.2.4 Metrics

Metrics are the parameters used to assess a model's performance. They can be categorised into Full Reference and No Reference Metrics. Consider $\hat{I}$ and I be the Super-Resolved Image and High Resolution Image respectively.

2.2.4.1 Full Reference Metrics

$$PSNR = 10\ \log_{10} \frac{255^2}{\frac{1}{N}\sum_{i=1}^{N}(I(i) - \hat{I}(i))^2} \tag{2.6}$$

PSNR is the measure of amount of noise present in an image given a reference image. It is considered as a distortion measure as it does a pixel wise comparison of given set of images.

$$SSIM\left(I, \hat{I}\right) = [\mathrm{C}_l\left(I, \hat{I}\right)]^{\alpha}[\mathrm{C}_c\left(I, \hat{I}\right)]^{\beta}[\mathrm{C}_s\left(I, \hat{I}\right)]^{\gamma} \tag{2.7}$$

$$\mathrm{C}_l\left(I, \hat{I}\right) = \frac{2\mu_I\mu_{\hat{I}} + C}{\mu_I^2 + \mu_{\hat{I}}^2 + C} \tag{2.8}$$

$$\mathrm{C}_l\left(I, \hat{I}\right) = \frac{2\sigma_I\sigma_{\hat{I}} + C}{\sigma_I^2 + \sigma_{\hat{I}}^2 + C} \tag{2.9}$$

$$\mathrm{C}_l\left(I, \hat{I}\right) = \frac{\sigma_{I\hat{I}} + C}{\sigma_I\sigma_{\hat{I}} + C} \tag{2.10}$$

Where the mean of pixels of a given image is μ_I, σ_I is the standard deviation of the pixels in the image and $\sigma_{I\hat{I}}$ is the covariance of pixels between image under test and reference image. C is any constant for stability. α, β, γ are parameters for controlling importance.

SSIM (Structural Similarity Index [27]) is a distortion measure. Unlike directly comparing images pixel-wise as PSNR, SSIM calculates image statistics of image –

luminance (mean), contrast (standard deviation) and structure (covariance) for image under test and a given reference image.

2.2.4.2 No Reference Metrics

No reference metrics / learning based quality metrics are a type of metrics which compute statistical deviations from "natural" images. There are many No reference metrics, out of which the following two metrics are used widely for evaluation.

Ma [28]*:* Ma uses random forest regression model. The features that are extracted from natural images are DCT (termed local frequency), DWT (termed Global frequency) and PCA (termed spatial discontinuity). These features are modelled using independent regression forests and the outputs are linearly regressed on perceptual scores evaluated by human subjects (also termed as Mean Opinion Score). These perceptual scores vary from 0-10, higher value denotes good quality

NIQE [29]*:* NIQE- Naturalness Image Quality Evaluator is a Quality Aware, opinion unaware, distortion Unaware metric. Natural images (Standard fixed set) are taken and Patches that have good sharpness (Quality) are extracted. Once the patches are selected, coefficients are computed and fitted in Multi Variate Gaussian Model. Maximum Likelihood Estimates (Mean and Variance) are calculated for the model. The above procedure is performed for all patches of Image under test and, distance (NIQE) between features (Estimates) of Image under test and Natural image are computed. Lower the value better is the Quality

2.2.4.3 Perception Distortion Trade-off

The goal of super-resolution is to estimate an image from the degraded / low resolution image, such that the higher frequency signals in the image are enhanced appropriately. As the models for super resolution use losses and metrics from distortion as well as perception, the study by **Blau et al** [30] show that the models evaluated with perception and distortion metrics / losses are at the odds with each other.

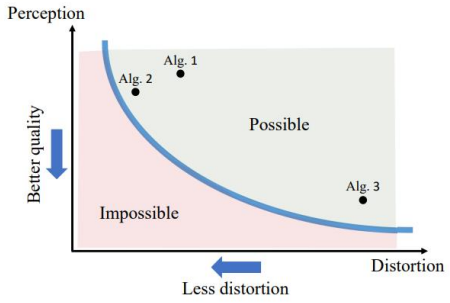

Figure 2.6 Perception-Distortion Plane [30]

From the study, it can be inferred that a model cannot produce results with less distortion as well as visually pleasing. The study also shows that the perception distortion trade-off exists for any distortion measure and is less severe for models that are evaluated based on distortion measures that capture semantic similarity between images (e.g. SSIM)

It is therefore evident that GAN models that are trained with perceptual losses give better results. ESRGAN, for instance has won PIRM challenge 2018 [15] , a competition that assesses model based on perceptual quality. Hence, deployment of models, trained with GAN based setting are discussed further.

2.2.5 Dataset Preparation

Preparing Dataset for super-resolution, unlike other tasks, requires two sets of images, Low resolution image for input and high resolution image for output. Though, in reality, the difficulty of getting both type of data, depends on the application, the state-of-the-art models for super resolution, are trained on datasets where low resolution images are produced by down sampling high resolution images based on bicubic or bilinear interpolation algorithms available in Matlab. Though preparing the datasets is easy in this case, the models that are trained with these datasets learn to invert bilinear or bicubic interpolation algorithms. In other words these datasets are can be termed as datasets with known prior. A prior in statistical sense is the probability of a model to generate the desired data before training and hence the models trained on these datasets can be termed as models with learned prior [12]. As a result, a model may fail to generalise on data in real time as the degradations of the images may not be known.

Therefore, to seek solution to this problem, several strategies of dataset preparation have been put forward. To create organic dataset, in RealSR [31], datasets are prepared by taking several images of a scene by varying focal length. To create application-cum-degradation specific dataset, **Bulat et al** [32], proposed to use GANs to down sample High Resolution images , thereby learning the degradation and use the down sampled images to train a super resolution model. Also, **Ulyanov et.al** [12] found that an architecture of a model contains a lot of information about the prior of an image and produced high resolution image from a random vector, fitting the model only with associated low resolution image.

CHAPTER 3

MODELS FOR EVALUATION

With the review done on various model architectures, losses, and supporting layers in Vitis, it had been decided to test following GAN based models from research, of which one model is constructed with Vitis supported layers and the other that do not contain layers supported by Vitis:

- Residual Dense Network (RDN) – supported by Vitis
- Enhanced Super-Resolution Generative Adversarial Network (ESRGAN) – not supported by Vitis

3.1 RESIDUAL DENSE NETWORK

Residual Dense Network (RDN), Figure 3.1 ,as proposed by **Zhang et.al** [33], is a feed forward network that uses the post up sampling architecture, with dense connections setup within feature extraction layers in the Low Resolution (LR) feature space. The goal of constructing this architecture is to fully use hierarchical features of convolution layers present in LR space using LFF, LRL, GFF and GRL. RDN has four parts:

- Shallow Feature Extractor
- Residual Dense Blocks
- Dense Feature Fusion
- Up Sampling Network

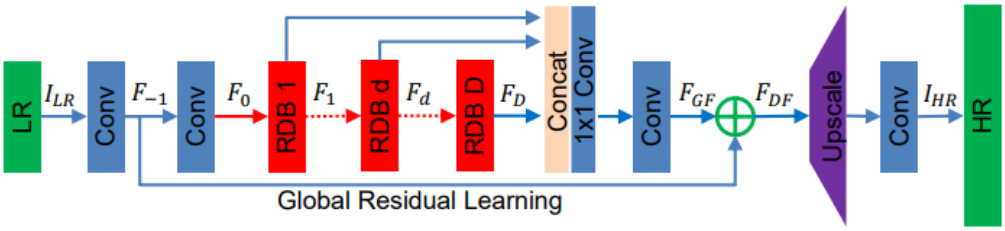

Figure 3.1 Residual Dense Network [33]

3.1.1 Model Description

3.1.1.1 Shallow Feature Fusion

Shallow Feature Extractor (SFF) consists of two Convolution layers, of which, the first convolution layer is used for Global Residual Learning (GRL).

3.1.1.2 Residual Dense Blocks

RDN uses a set of Residual Dense Blocks. A Residual Dense Block (RDB) consist a set of densely connected layers (convolution and activation), Local Feature Fusion (LFF) and Local Residual Learning (LRL). A densely connected layer performs convolution and activation over concatenation of preceding activations. LFF denotes convolution operation with kernel of size 1x1 over concatenation of activations of all densely connected layers of the block. It is given that LFF makes learning easier for larger network sizes. LRL denotes the addition of output of LFF and input of the block. As discussed before, all residual connections do not use Batch Normalisation, as it produces artefacts. The diagrammatic representation is given in Figure 3.2

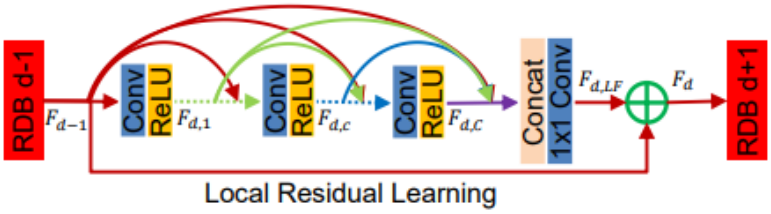

Figure 3.2 Residual Dense Block[33]

3.1.1.3 Dense Feature Fusion

Dense Feature Fusion (DFF) Figure 3.1 consists of Global Feature Fusion (GFF) and Global Residual Learning (GRL). GFF consists of two Convolution layers with kernel size 1x1 and 3x3. The first Convolution layer is similar to LFF where the input of the layer is the concatenation of all RDBs present in the network. GRL, similar to LRL denotes the addition of Shallow Feature Extractor and the output of GFF.

3.1.1.4 Up Sampling Network

Efficient Sub-Pixel Convolutional Neural Network (ESPCN) [17] is used for up sampling the features in LR space. Super-Resolution in Pixel space is the conversion of an image of dimensions H X W X C to an image of dimensions H*r X W*r X C where H, W, C are Height, Width and Channels of an image, r is the up sampling factor. Unlike conventional image interpolation, in Sub-Pixel layer, there are two stages in up sampling, 1) create r^2 channels from input LR feature space, making a LR feature space of dimensions H X W X r^2C 2) shuffle: reshaping the LR feature space into r*H X r*W X C. The authors argue that this methodology do not introduce artefacts.

Keras implementation of Sub-Pixel uses tf.nn.depthtospace() [34], which is not supported in Vitis. Alternatively, ***Francesco Cardinale*** [35] uses UpSampling2D layer [36] for up sampling the LR feature space. Hence, the code base developed by *Francesco Cardinale* is implemented here.

3.1.1.5 Hyper Parameter Description

- Optimiser: Adam (β1, β2)
- Epochs trained
- C (number of Convolution layers in RDB)
- D (number of RDBs)
- G (Growth rate [16]/Number of channels in C Convolution layers)
- G0 (Growth rate [16]/Number of channels in SFF Convolution layers)
- Scale (Up Sampling Factor)
- Learning Rate
- decay_factor (Amount by which learning rate reduces)
- decay_frequency (number of epochs for change in learning rate)

3.1.2 Training Setting

For this study, pre-trained model developed by ***Francesco Cardinale*** is used. The original implementation as proposed by ***Zhang.et.al***, uses input image patch sizes of 32x32. ***Francesco Cardinale*** uses different sizes of image patches of DIV2K dataset [37] for training and Set5 [38], Set14 [39], Urban100 [40] as test datasets, ranging from 16x16 to 500x500. The images are normalized to 0-1 range as a pre-processing step. Though ***Zhang.et.al*** uses L1 loss, ***Francesco Cardinale*** uses Perceptual, Adversarial and L1 losses, making RDN a generator network in a GAN framework, similar to ESRGAN.

3.1.2.1 Hyper Parameters

- Optimiser: Adam β1 = 0.9, β2 = 0.999
- Epochs trained : 219
- C**:** 6
- D**:** 20
- G**:** 64
- G0: 64
- scale:2
- LR**:** 0.0004
- decay_factor: 0.5
- decay_frequency: 30

3.2 RESIDUAL-IN-RESIDUAL NETWORK

Residual-in-Residual Network (RRDN), as proposed by **X.Wang et al** [15], is a generator neural network in ESRGAN. The baseline architecture is inspired from SRResNet used in SRGAN [22] and RRDN is an improved version of the baseline architecture. This improvement corresponds to easy training of the model. The other improvements include 1) residual scaling and smaller initialization 2) use of losses: RaGAN [26] for adversarial loss and perceptual loss.

3.2.1 Model Description

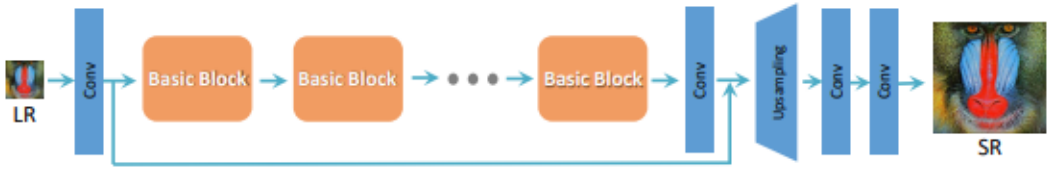

Figure 3.3 Architecture of SRGAN[15]

3.2.1.1 Residual Dense Block

Residual Dense Blocks (RDB) or simply, Dense Blocks used in RRDN follows the same architecture as RDB in Residual Dense Network (RDN) discussed above. The key difference lies in the implementation of LFF (Figure 3.2) Unlike the use of Convolution with kernel size 1x1 in LFF in RDN, a Convolution layer with kernel size 3x3 is used here. Also LRL is absent in this case.

3.2.1.2 Residual-in-Residual Dense Block

Residual-in-Residual Block (RRDB), termed as basic block in the Figure 3.3 is a series of Residual connections, whose residues are RDBs with residual scaling. This can be attributed to Local Residual Learning in RDN. As the network uses Residual Dense Network as Residues, hence the name Residual-in-Residual Dense Network.

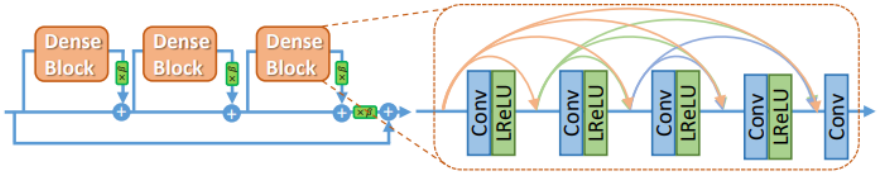

Figure 3.4 Residual-in-Residual Dense Block[15]

3.2.1.3 Up sampling Network

Efficient Sub-Pixel Convolutional Neural Network (ESPCN) [no] is used for up sampling the features in LR space. Vitis does not support this layer.

3.2.1.4 Network Interpolation

To remove unpleasant noise generated by GAN based methods, a new strategy is proposed in ESRGAN. The model (say M1) is firstly trained using Per-Pixel based losses (e.g.: L1) and Metrics (PSNR). The model M1 is then trained using the GAN based setting and the model with new parameters is derived (say M2). An interpolated model is termed as weighted average of M1 and M2. This strategy gives an option between PSNR/Distortion and Perceptual Quality as the weight of weighted average can be adjusted to suit the application.

3.2.1.5 Hyper Parameter Description

- Optimiser: Adam (β1, β2)
- Epochs trained
- C (number of Convolution layers in RDB)
- D (number of RDBs in a RRDB)
- T (number of RRDBs)
- G (Growth rate [16]/Number of channels in C Convolution layers)
- G0 (Growth rate [16]/Number of channels in SFF Convolution layers)
- Scale (Up Sampling Factor)
- Learning Rate
- decay_factor (Amount by which learning rate reduces)
- decay_frequency (number of epochs for change in learning rate)

3.2.2 Training Setting

For this study, pre-trained model developed by ***Francesco Cardinale*** is used. The original implementation, as proposed by **X.Wang** uses input image patch sizes of 96x96, 128x128, and 192x192. ***Francesco Cardinale*** uses different sizes of image patches of DIV2K dataset [37] for training and Set5 [38], Set14 [39], Urban100 [40] as test datasets, ranging from 16x16 to 500x500. The images are normalized to 0-1 range as a pre-processing step. As described in [15], the training is a two stage process. Firstly, RRDN is trained in PSNR oriented approach using L1 loss. This PSNR driven model is then used as the initiation in the GAN based setting. This two stage approach is used for 1) avoiding local optima 2) faster training 3) Better quality as opposed to training a model with random initialisation.

3.2.2.1 Hyper Parameters

- Optimiser: Adam β1 = 0.9,β2 = 0.999
- Epochs trained : 299
- C:4
- D:3
- G:32
- G0:32
- T:10
- Scale:4
- LR: 0.0004
- decay_factor: 0.5
- decay_frequency:30

CHAPTER 4

APPLICATION DESIGN

The Application design focusses on deployment and therefore training a model is not discussed here. A pretrained model developed by ***Francesco Cardinale*** [35], using keras framework with tensorflow backend, is tested in this implementation.

4.1 KERAS MODEL DEPOLYMENT

As discussed in chapter 2, for deploying a deep neural network in Zynq Ultrascale+ ZCU102, Vitis AI provides a set of tools and the following steps are to be followed:

- Train a model constructed with Vitis supported layers (discussed below) using Keras with Tensorflow backend. Versions Supported: v1.12.0 and v.1.15.0
- Generate Model Checkpoints and freeze the graph
- Prune the model using Vitis AI Pruner tool. This step is optional and can be used for larger models, if needed
- Quantize the model using Vitis AI Quantisation tool. This is a mandatory step, without which, the model cannot be deployed.
- Compile the model to obtain a .elf file. Compilation is the process of creating instructions for DPU. The .elf file contains model information and the operations that are needed to be performed by the DPU
- For debugging purposes, Vitis AI profiler can be used to get layer wise information after compilation. Profiler can also be used while runtime
- Convert the .elf file to a .so Shared library file using GCC toolchain for cross compilation
- Copy the shared library file to /usr/lib in the hardware platform

Xilinx/vitis-ai:tools-1.0.0-cpu docker image is used. The outline of the usage of these tools is discussed here. For overall understanding, refer [8].

4.1.1 Creating Checkpoints

As the model is created using Keras with Tensorflow backend, there are two formats to save a model 1) Saved Model 2) H5 and the model is saved using H5 format. Therefore, from H5 format, checkpoint files are created using Tensorflow saver object, keras backend session object and the graph from the session. The following python code outlines the process.

```
import tensorflow.keras as K #v2.2.4-tf ot v2.2.5
import tensorflow as tf
K.backend.set_learning_phase(0)
```

```
loaded_model=k.models.load_model(keras_h5) #keras_h5 is
the path to h5 model file
saver=tf.train.Saver() #creating saver object to save
graph
tf_session=K.backend.get_session() # fetching the session
from backend to access variables
input_graph_def=tf_session.graph.as_graph_def()#accessing
the graph
saver.save(tf_session,ckpt_dir)#save the variables,
metadata from the given graph and session, initialise
ckpt_dir before calling this function
tf.io.write_graph(input_graph_def,tfgrapgh_path,tfgraph_f
ilename)# save protobuf file to be used for freezing and
visualisation purposes
```

4.1.2 Freeze Graph

The checkpoint that is created contains all information about the model, for example, the losses, metrics, hyper parameters used, which are not required for inference. Also the weight parameters are in variable scope. Therefore, to remove unwanted information and to change the scope of weights from variables to constants, Tensorflow has a tool freeze_graph which can be accessed in command line. freeze_graph tool inputs protobuf file of the model, checkpoint files , output graph name and node name. An example command is shown below:

```
freeze_graph –input_graph=./tf_infer_graph.pb –
input_chechpoint=./tf_chkpt.ckpt –input_binary=true –
output_graph=./frozen_graph.pb –
output_node_name=nodename/operation
```

Sometimes, the above command might report an error, in which case, [41] can be used.

4.1.3 Pruning

Pruning is a process of removing unused / least priority weights in the graph. Pruning is not done in this implementation, as it might lead to inconsistent results for Super-Resolution.

4.1.4 Quantization

Quantisation is the process of converting weights from 32 bit Floating Point values to 8 bit Integer representation. This is a mandatory step as DPU accepts only values of 8 bit Integer representation. To accomplish this task, Vitis AI provides a tool `vai_q_tensorflow` which can be accessed in command line. The tool inputs frozen graph, input and output node names, calibration function etc and results a protobuf model for compilation. A sample command in shown below:

```
vai_q_tensorflow quantize --input_frozen_graph=
/path_to_frozen_graph --input_nodes= input_node_name --
input_shape= dynamic_shape_of_input_tensor --output_nodes
=output_node_name -input_fn=calib.calib_input --
calib_iter=2 --output_dir=/path_to_output_directory -
method=0
```

Quantisation requires a calibration function. This function collects training dataset (100 – 1000), pre-processes them and inputs it to the frozen model. The tool, thereby estimates the weight distribution of the model for effective quantisation. A sample code for calibration function is given below:

```
from PIL import Image
import os
import numpy as np
import glob
import random

calib_image_dir="/path/to/dataset/" #assume dataset
contains 100 data points
calib_batch_size=50
image_files=[f for f in
glob.glob(calib_image_dir+"*.png")]

def calib_input(iter):
     images=[]
     for index in range(0,calib_batch_size):
         curimg=image_files[iter* calib_batch_size+index]
         im=Image.open(curimg)
         image=np.asarray(im)
         image=image/255.0 #Pre-processing
         images.append(image.tolist())
     return{"LR":images} #assume LR to be the input node
```

here, calib_iter * calib_batch_size = number of data points

The tool generates *deploy_model.pb* and *quantize_eval_model.pb* files to the specified output directory. *deploy_model.pb* is used for compilation and *quantize_eval_model.pb* is used for simulation purposes.
Note: The models were quantised with method 0 (Non-Saturation Quantisation). Models quantised with method 1(Min-diff) produced random speckles (see chapter 5)

4.1.5 Compilation

Compilation is the process of generating instructions (.elf) to DPU to execute the given model in FPGA. Vitis AI provides `vai_c_tensorflow` tool for this purpose that can be used in command line. This tool inputs *deploy_model.pb* generated by quantize tool, architecture of DPU, output directory and user defined neural network name. A sample command is given below:

```
vai_c_tensorflow --frozen_pb=/path_to_deploy_model.pb
--arch=
/opt/vitis_ai/compiler/arch/dpuv2/ZCU102/ZCU102.json
--output_dir=/path_to_directory –net_name=network_name
```

As the model developed by ***Francesco Cardinale*** is trained with a wide range of input image sizes, it was planned to test images with incremental increase in patch sizes. The patch sizes tested were: 44x44, 84x84 and 124x124. Beyond the patch size of 124 x124, the Vitis AI tools could not freeze or compile the model.

Compilation results for RDN models with input sizes 44x44, 84x84 and 124x124 is shown here. Results regarding RRDN models are shown in Chapter 5: Results

Table 4.1 Compilation of Models with 3 input patch sizes

Kernel ID	0	0	0
Name	RDN_44	RDN_84	RDN_124
Kernel Name	RDN_44	RDN_84	RDN_124
Kernel Type	DPUkernel	DPUkernel	DPUkernel
Codes Size	0.85 MB	2.07 MB	3.35 MB
Param Size	15.59 MB	15.59 MB	15.59 MB
Workload MACs	63265.70 MOPS	230579.91 MOPS	502465.53 MOPS
IO memory space	7.35 MB	26.75 MB	33.88 MB
Mean Value	0,0,0	0,0,0	0,0,0
Total Tensor Count	289	289	271
Boundary Input Tensor (Dimensions)	LR:0 (44*44*3)	LR:0 (84*84*3)	LR:0 (124*124*3)
Boundary Output Tensor (Dimensions)	SR_convolution:0 (88*88*3)	SR_convolution:0 (168*168*3)	SR_convolution:0 (248*248*3)
Total Node Count	288	288	270
Input Node (Dimensions)	F_m1_convolution (44*44*3)	F_m1_convolution (84*84*3)	F_m1_convolution (124*124*3)
Output Node (Dimensions)	SR_convolution (88*88*3)	SR_convolution (168*168*3)	SR_convolution (248*248*3)

4.1.6 Cross Compilation

With the generation of executable .elf file, a library file should also be created, to make the file compatible with programming languages like C++ / Python.

A sample command is shown below:

```
aarch64-linux-gnu-gcc
--sysroot=/opt/vitis_ai/petalinux_sdk/sysroots/aarch64-
xilinx-linux -fPIC -shared <name of .elf file>
-o <name of output library file>
```

Name of the output library file should be in the form libdpumodel*modelname*.so for example, for model RDN_44 (as the name generated by the compilation tool) the library name is libdpumodelRDN_44.so. This file should be copied to /usr/lib/ directory in ZCU102

4.2 APPLICATION DEVELOPMENT

This section discusses some key API functions and explains the functions of the code[1].

4.2.1 Overall Design

Figure 4.1 Application flow

As the diagram depicts, an input image is first split into patches of size compatible with the input shape of the model. Then iteratively, each patch is set to the global memory buffer / IO memory space. Along the iteration, the model is run for each patch, generates the output and stores in IO memory space. Along the iteration, the generated patches are recombined to give the super-resolved image.

[1] https://github.com/gkrislara/Image-super-resolution-FPGA/blob/master/RDN_SR.cpp

4.2.2 API Functions

This section focusses on some important API functions that are needed to be used in a C++ application. These API come under the header file dnndk/n2cube.h. Though there is a set of python APIs available, C++ suits the production requirements.

`dpuOpen()`: attaches and opens the DPU file /dev/dpu

`dpuLoadKernel()`: Loads the specified DPU kernel including DPU instructions, weights, biases

`dpuCreateTask()`: allocates memory buffer to DPU for the corresponding kernel

`dpuRunTask()`: launches and executes the specific operation given by the instructions

`dpuDestroyTask()`: deallocates resources of the DPU for the specific task

`dpuDestroyKernel()`: releases the specific kernel and it resources

`dpuClose()`: detaches the DPU

4.2.3 Function Description

4.2.3.1 Splitting Images into Patches

For a given image as cv::Mat Matrix, Patch size, pixel overlap, this function splits the image into overlapping patches.

```
static int oheight;
static int owidth;

std::vector<cv::Mat> patchify(cv::Mat img,int opatch_size,int
padding_size=2)
{
int patch_size=opatch_size;
std::vector<cv::Mat> cpatches;
cv::Mat patch;
cv::Mat image= img;
int width=image.cols;
int height= image.rows;
int w_rem= width % patch_size;
int h_rem = height % patch_size;
int w_extend = patch_size-w_rem;
int h_extend = patch_size-h_rem;
cv::Mat ext_image;

cv::copyMakeBorder(image,ext image,0,h extend,0,w extend,cv::BORDER R
EPLICATE);

cv::copyMakeBorder(ext_image,ext_image,padding_size,padding_size,padd
ing_size,padding_size,cv::BORDER_REPLICATE);

oheight=ext_image.rows;
owidth=ext_image.cols;
int w_left,w_width,h_top,h_height;
```

```
for(int i=padding_size;i<ext_image.cols-padding_size;i+=patch_size)
{
  for(int j=padding_size;j<ext_image.rows-padding_size;j+=patch_size)
       {
         w_left= i-padding_size;
         h_top = j-padding_size;
         w_width = patch_size + 2*padding_size;
         h_height =  patch_size + 2*padding_size;
         cv::Rect crop(w_left,h_top,w_width,h_height);
         patch=ext_image(crop);
         cpatches.push_back(patch);
 }
}
return cpatches;
}
```

4.2.3.2 Setting Input Patches into Memory Buffer

For a given input image patch, task pointer, input Node name, this function sets the patch to the DPU IO memory buffer.

```
int dpuSetInputImage(DPUTask *task, const char* nodeName, const
cv::Mat &image,int idx=0)
{
    int value;
    int8_t *inputAddr;
    unsigned char *resized_data;
    cv::Mat newImage;
    float scaleFix;
    int height, width, channel;

    height = dpuGetInputTensorHeight(task, nodeName, idx);
    width = dpuGetInputTensorWidth(task, nodeName, idx);
    channel = dpuGetInputTensorChannel(task, nodeName, idx);


    if (height == image.rows && width == image.cols) {
        newImage = image;
    }
    else{
        std::cout<<"Required image size
"<<height<<"x"<<width<<"x"<<channel<<"\n";
        return -1;
    }
    resized_data = newImage.data;

    inputAddr = dpuGetInputTensorAddress(task, nodeName,
idx);
    scaleFix = dpuGetInputTensorScale(task, nodeName, idx);


    for (int idx_h=0; idx_h<newImage.rows; idx_h++) {
     for (int idx_w=0; idx_w<newImage.cols; idx_w++) {
      for (int idx_c=0; idx_c<3; idx_c++) {
                  inputAddr[idx_h*newImage.cols*3+idx_w*3+idx_c] =
newImage.at<cv::Vec3f>(idx_h, idx_w)[idx_c]* scaleFix;
               }
            }
        }
    return scaleFix;}
```

4.2.3.3 Executing Feed Forward Operation

This function calls `dpuSetInputImage()` and accordingly executes the Feed Forward Deep Neural Network operation and results a super-resolved patch

```
cv::Mat runPatch(cv::Mat patch,DPUTask *taskConv)
{
int height=dpuGetOutputTensorHeight(taskConv,OUTPUT_NODE,0);
int width=dpuGetOutputTensorWidth(taskConv,OUTPUT_NODE,0);
int channel=dpuGetOutputTensorChannel(taskConv,OUTPUT_NODE,0);
int total_size=height*width*channel;
cv::Mat reimg=
cv::Mat(height,width,CV_32FC3,cv::Scalar(0.0,0.0,0.0));

int8_t outdata[total_size]={0};
float scaleFix=0.0;

cv::Mat image= patch;
cv::cvtColor(image, image, cv::COLOR_BGR2RGB);
        cv::normalize(image,image,0,1,cv::NORM_MINMAX,CV_32F);

scaleFix=dpuSetInputImage(taskConv,INPUT_NODE,image);//setimage--Flag
issue

dpuRunTask(taskConv);

dpuGetOutputTensorInHWCInt8(taskConv,OUTPUT_NODE,outdata,total_size);
//get output
//tensor to cv::mat

for (int idx_h=0; idx_h<height; idx_h++) {
 for (int idx_w=0; idx_w<width; idx_w++) {
  for (int idx_c=0; idx_c<channel; idx_c++){
   reimg.at<cv::Vec3f>(idx_h, idx_w)[2-
idx_c]=outdata[idx_h*width*channel+idx_w*channel+idx_c];
                }
            }
        }
     cv::normalize(reimg,reimg,0,255,cv::NORM_MINMAX,-1);
     return reimg;
}
```

4.2.3.4 Reconstructing Image from Patches

The following functions are used to remove the padding / overlap and concatenating the patches to form a complete image.

```
std::vector<cv::Mat> unpad(std::vector<cv::Mat> patches,int pad){
  cv::Mat patch;
  std::vector<cv::Mat> uppatches;
  for(auto it=patches.begin();it!=patches.end();++it)
  {
    patch=*it;
    uppatches.push_back(patch(cv::Rect(2*pad,2*pad,patch.cols-
2*pad,patch.rows-2*pad)));
  }
  return uppatches;
}
```

The function requires the super resolved patches, height and width of resulting image and padding size (usually twice the size of padding given to patchify() function)

```
cv::Mat depatchify(std::vector<cv::Mat> cpatches, int op_width, int
op_height,int padding_size=4)
{
   cv::Mat image(2*oheight,2*owidth,CV_32FC3,cv::Scalar(0,0,0));
   std::vector<cv::Mat> patches=unpad(cpatches,padding_size);
   int patch_size=patches[0].cols;
   int patches_per_col = 2*oheight/patch_size;

   int col=-1,row=0;
   for(int i=0;i<patches.size();i++)
    {
       if (i % patches_per_col == 0)
        { ++col;
          row=0;
         }

patches[i].copyTo(image(cv::Rect(col*patches[i].cols,row*patches[i].r
ows,patches[i].cols,patches[i].rows)));
    row++;}
   std::cout<<"rows:"<<image.rows<<"\n";
   std::cout<<"cols:"<<image.cols<<"\n";
   return image(cv::Rect(0,0,op_width,op_height));
}
```

4.2.3.5 Putting All Things Together

This is the function that is depicted in the Figure 4.1 It inputs an image, processes it with the help of DPU and saves according to patch size and pixel overlap.

```
void runRDN_SR(std::string imgpath,DPUTask *taskConv,int patch_size){
    assert(taskConv);
    std::vector<cv::Mat> patches,patchesx2;
    cv::Mat patchx2;

    cv::Mat image= cv::imread(imgpath);
    int imwidth=image.cols;
    int imheight=image.rows;
    int errsize;
    std::cout<<"\n\nPROCESS STARTED\nProcessing "<<imgpath<<"\n";
    std::cout<<"width:"<<imwidth<<"\n";
    std::cout<<"height:"<<imheight<<"\n";

    int overlap=0;
    patches=patchify(image,patch_size,overlap);

    for(auto it=patches.begin();it!=patches.end();++it)
    {
        patchx2=runPatch(*it,taskConv);
        errsize=patchx2.cols;
        if(errsize!=2*patch_size)
          break;
        patchesx2.push_back(patchx2);
    }

if(patchesx2.size()==patches.size()){
        std::cout<<"Processed "<<patches.size()<<" patches\n";
```

```
        std::cout<<"conversion complete\n";
        std::cout<<"oheight:"<<oheight<<"\n";
        std::cout<<"owidth:"<<owidth<<"\n";
        cv::Mat
HR=depatchify(patchesx2,2*imwidth,2*imheight,2*overlap);
        std::string name = imgpath.substr(imgpath.find_last_of("/") +
1);
        std::string
save="HR"+std::to_string(patch_size)+"_"+std::to_string(overlap)+"_"+
name;
        cv::imwrite(save,HR);
        std::cout<<"image written successfully as "<<save<<"\n";
    }
    else{
        std::cout<<"inconsistent patches!!\n output patch size
"<<errsize<<"\n";
    }
    patches.clear();
    patchesx2.clear();
}
```

4.2.3.6 Compiling the Application

This application is intended to run in arm processor core and hence g++ complier compatible with arm gcc or g++ toolchain should be used.
The following is a sample compilation command

```
g++ -std=c++11 -03 –I. –o RDN_SR RDN_SR.cpp –ln2cube –
lhineon –lopencv_imgcodecs –lopencv_highgui –
lopencv_imgproc –lopencv_core –pthread –lxrt_core
```

CHAPTER 5

EXPERIMENTS, RESULTS AND OBSERVATION

5.1 RDN

20 Images from a mix of DIV2K [37] (Training Dataset), Set5 [38]. Set14 [39], Urban100 [40] (test dataset) are used for evaluation. To obtain the Low Resolution Images, High Resolution Dataset were down sampled by a factor of 2 using Matlab resize function with bicubic interpolation As mentioned before, images were split into three patch sizes: 44x44, 84x84, and 124x124.

5.1.1 Visual Results

This section shows results of some of the datasets mentioned above.

5.1.1.1 Training Dataset

Image Name: 0010 Dataset: DIV2K

High Resolution Image:

Figure 5.1 0010-High Resolution

Super Resolved Images:

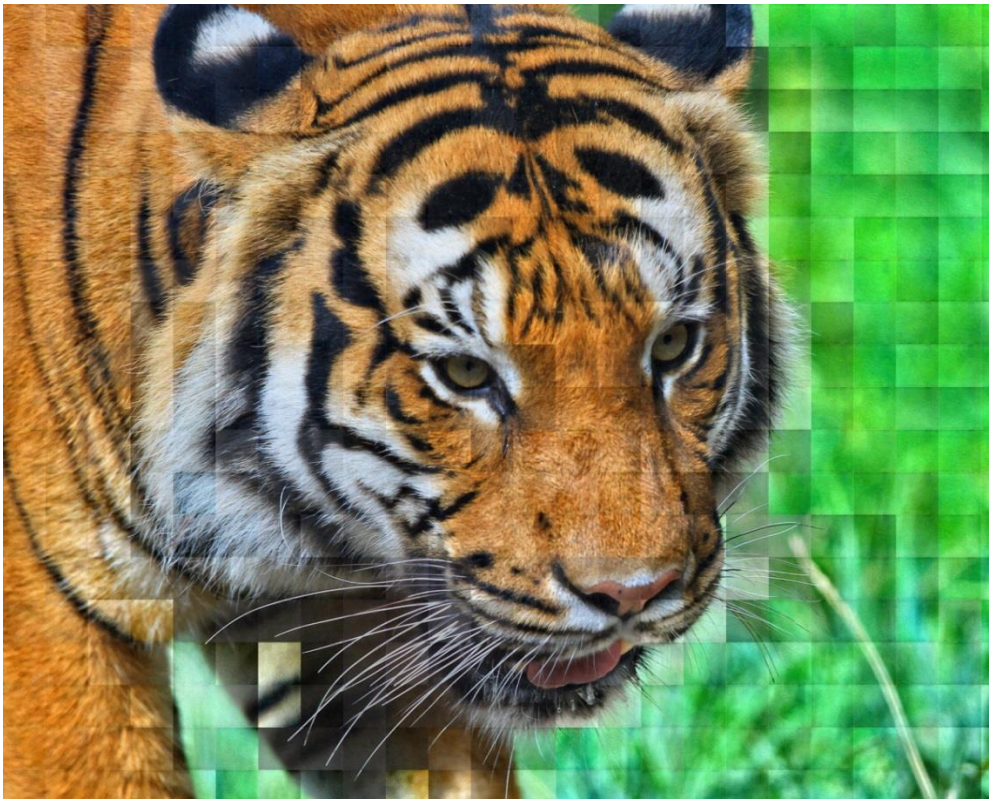

Figure 5.2 0010 – Super Resolution Image Patch size:44x44

Figure 5.3 0010 – Super Resolution Image Patch size:84x84

Figure 5.4 0010 – Super Resolution Image Patch size:124x124

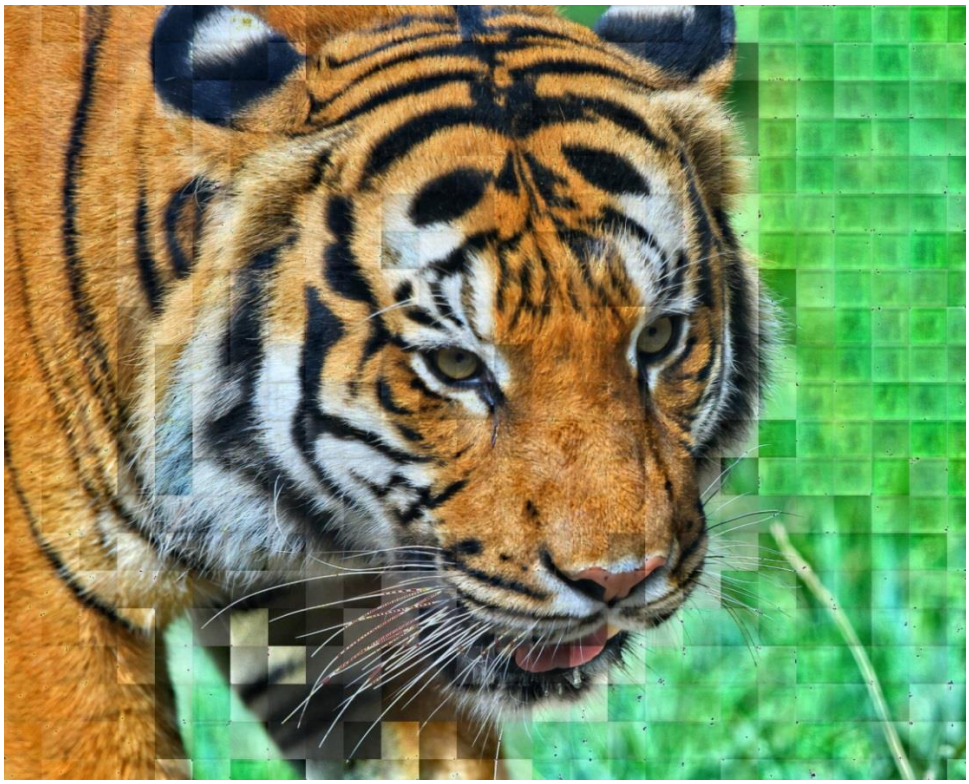

Figure 5.5 0010 – Super Resolution Image, Patch size:44x44, Model quantised with method 1 (Random Speckles can be observed)

Image Name: 0002 Dataset: DIV2K

Figure 5.6 0002 – High Resolution Image

Figure 5.7 0002 – Super Resolution Image Patch size:44x44

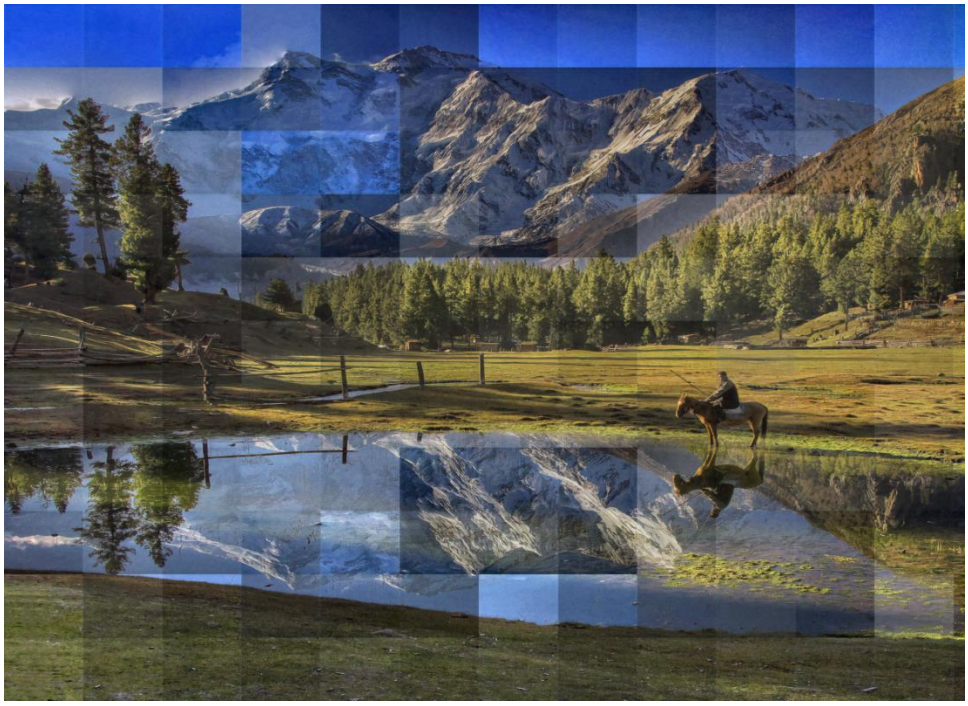

Figure 5.8 0002 – Super Resolution Image Patch size:84x84

Figure 5.9 0002 – Super Resolution Image Patch size:124x124

5.1.1.2 Test Dataset

Image Name: Monarch Dataset: Set14

High Resolution Image:

Figure 5.10 Monarch – High Resolution Image

Figure 5.11 Monarch – Super Resolution Image Patch size: 44x44

Figure 5.12 Monarch – Super Resolution Image Patch size: 84x84

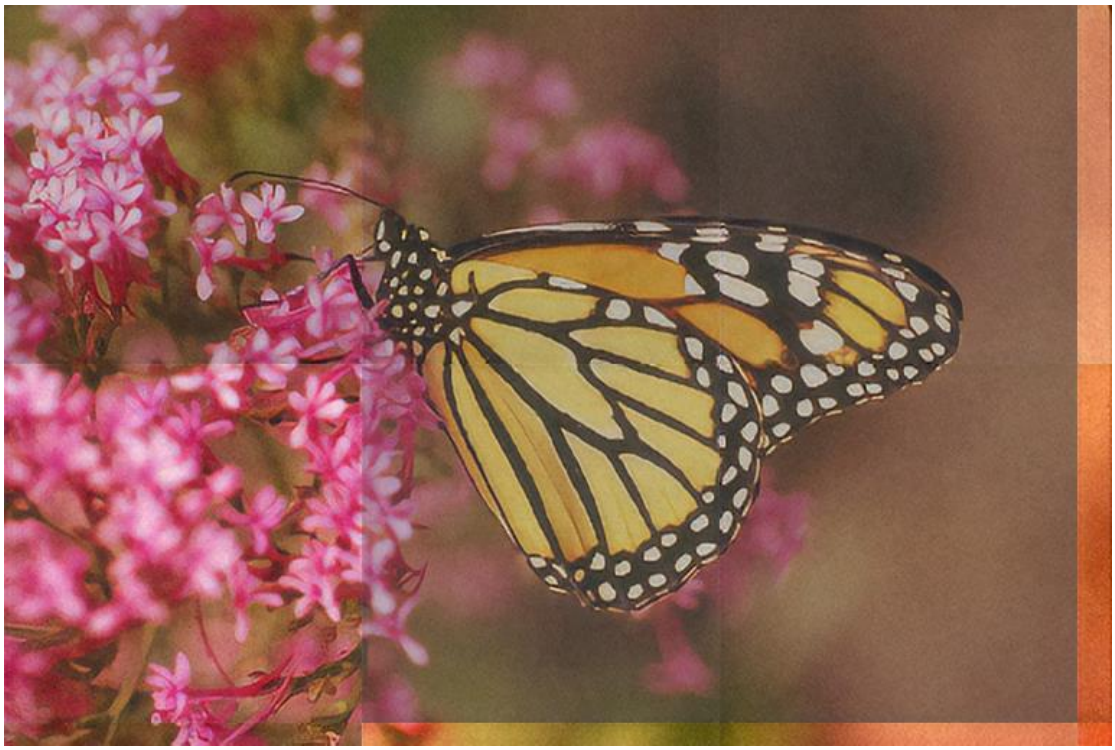

Figure 5.13 Monarch – Super Resolution Image Patch size: 124x124

Image Name: Zebra Dataset: Set14

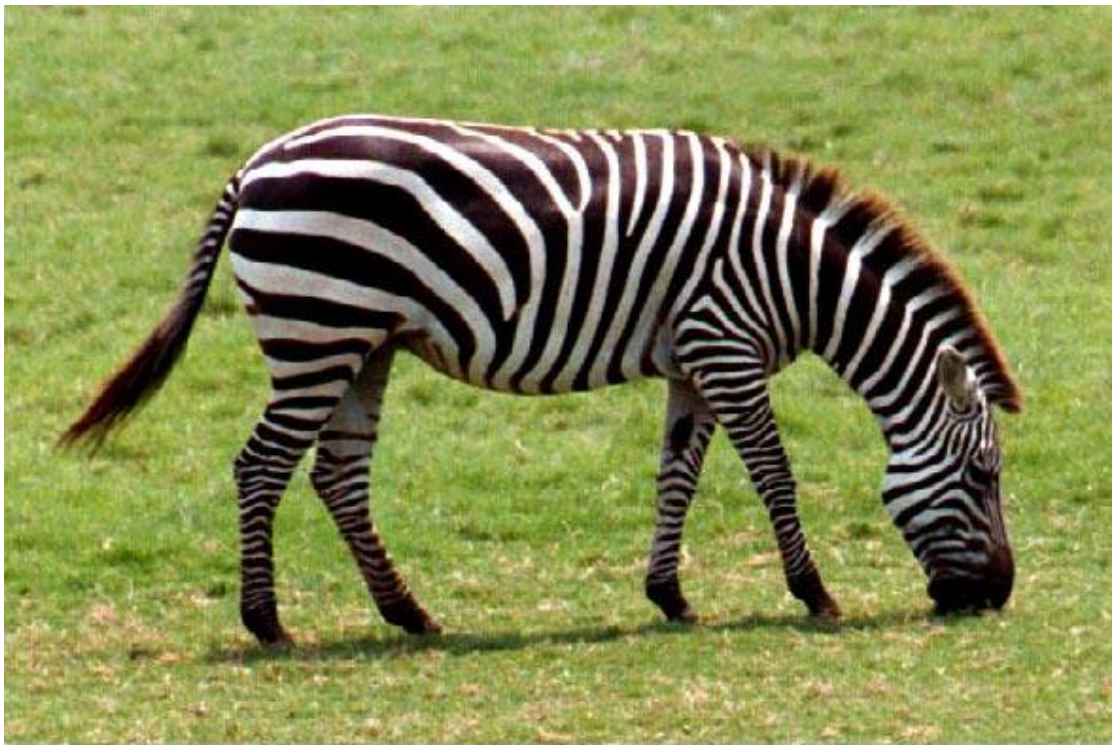

Figure 5.14 Zebra – High Resolution Image

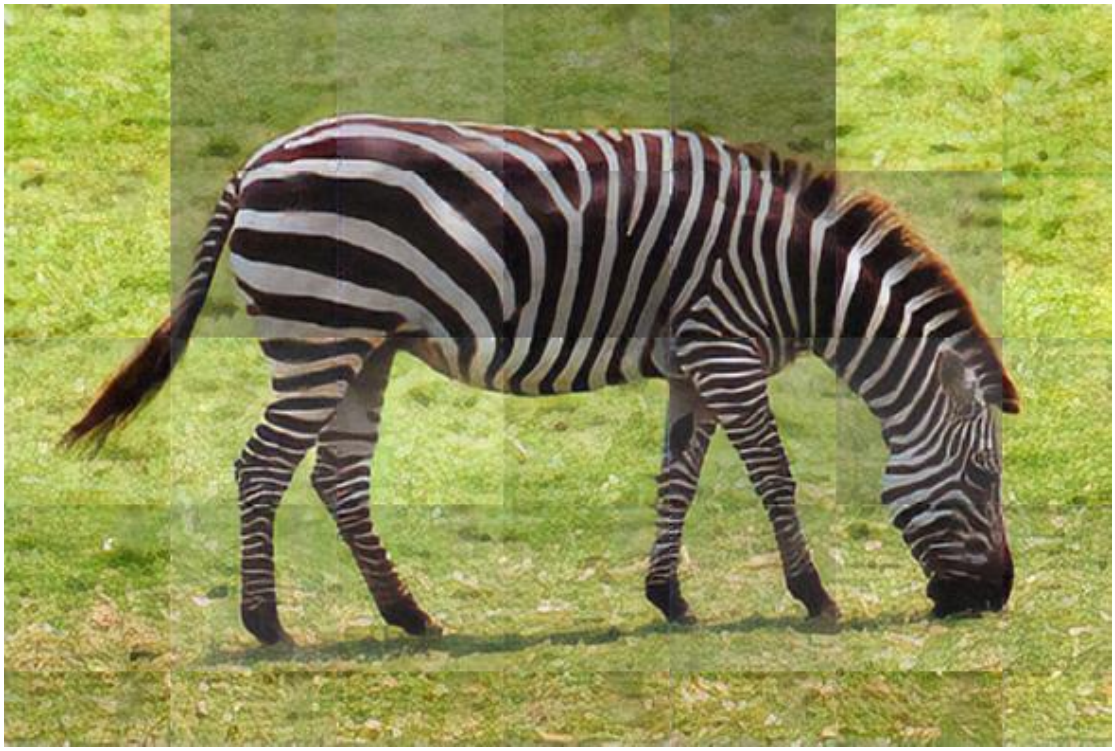

Figure 5.15 Zebra – Super Resolution Image Patch size: 44x44

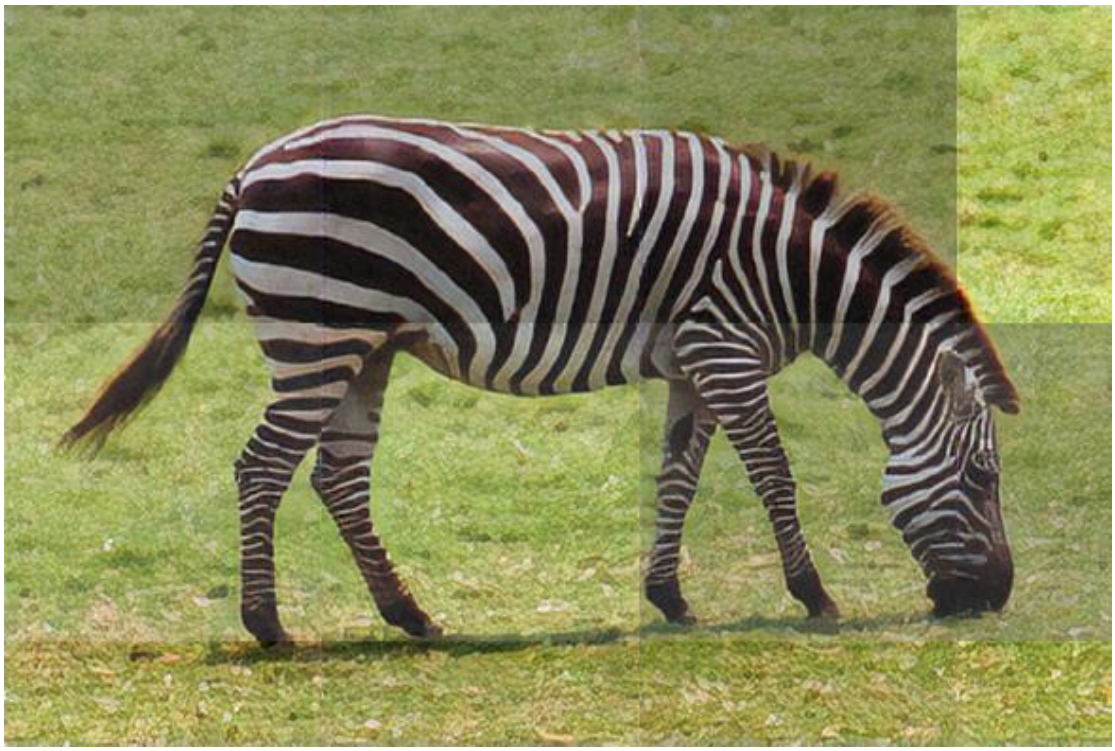

Figure 5.16 Zebra – Super Resolution Image Patch size: 84x84

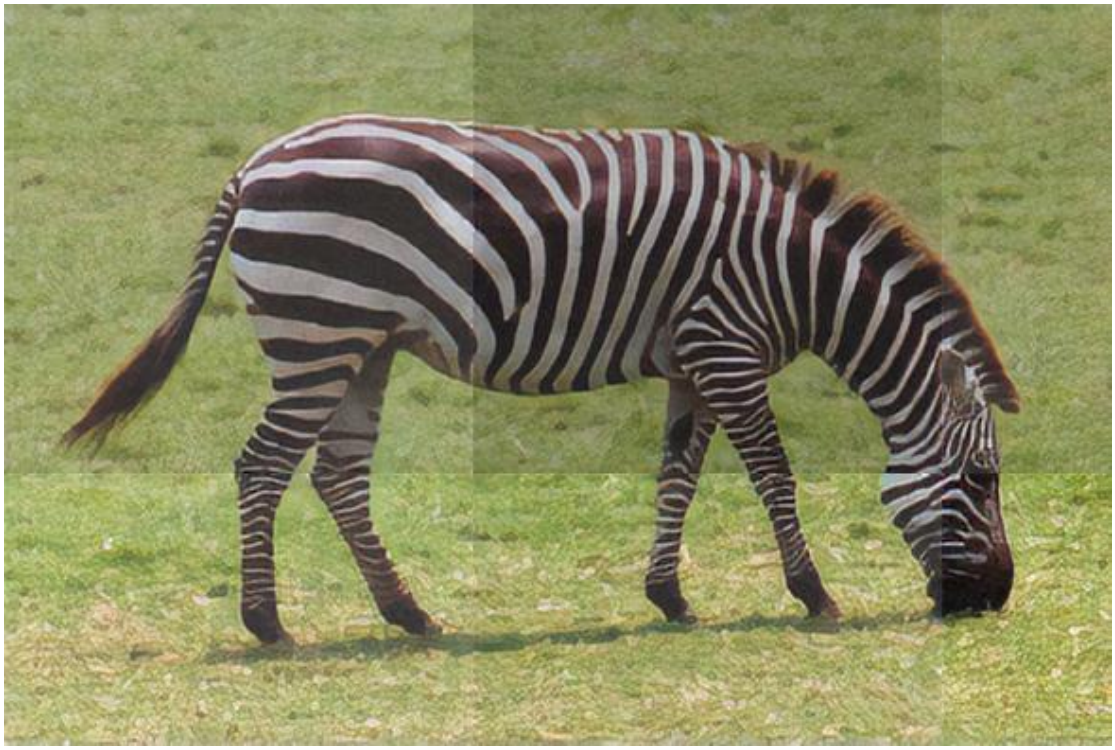

Figure 5.17 Zebra – Super Resolution Image Patch size: 124x124

From the above pictures it can be seen that the image reconstruction has resulted in patches that have uneven tone and high saturation. The High saturation gradually decreases with the increase in patch size. It is also evident that the tone change occurs when there is a transition between the distributions of patches of an image.

Patch wise comparison

Figure 5.18 Comparison of High Resolution and Super Resolution

It can be seen that the Super Resolution image misses the focal details that are present in High Resolution Image and produces a Super Resolution with smudgy appearance.

5.1.2 Quantitative Results

The following are the quantitative results based on FR metrics: PSNR, SSIM and NR metric NIQE. It can be observed that

- Deployed Model produces less natural image / images with low perceptual quality (in terms of NIQE)
- On an average, as the patch size increases, the image becomes less distorted (in terms of PSNR and SSIM)
- Perceptual Quality of Test data is lesser than Training data

Table 5.1 PSNR, SSIM, NIQE Results for images with input patch size 44x44

Name	Dataset	PSNR	SSIM	NIQE SR	NIQE HR
0001	DIV2K	29.1093	0.79208	4.8444	3.6858
0002	DIV2K	14.2514	0.6745	6.7145	5.4203
0010	DIV2K	16.7815	0.79411	5.0161	3.4753
0070	DIV2K	19.226	0.80755	6.2358	5.3119
0800	DIV2K	22.3656	0.71079	5.1843	3.5008
Baboon	Set14	22.1703	0.74919	7.3683	6.3862
Baby	Set5	23.595	0.81414	7.9708	3.6322
Barbara	Set14	21.6319	0.8051	5.8627	5.0425
Bird	Set5	17.9942	NA	6.1355	5.1477
Butterfly	Set5	17.598	NA	10.5584	9.5716

Face	Set14	18.3988	NA	9.6196	9.8626
Img_012	Urban100	18.7874	0.80056	6.3732	4.4676
Img_045	Urban100	17.8128	0.85171	9.444	9.4225
Img_067	Urban100	16.4783	0.70131	7.6275	5.9088
Img_078	Urban100	14.6047	0.63541	6.1589	4.1388
Img_100	Urban100	18.9525	0.85617	9.2197	8.4111
Monarch	Set14	21.353	0.80669	5.0297	4.2552
Pepper	Set14	26.4627	0.76069	6.4423	7.3509
Woman	Set5	19.2023	NA	6.1254	5.1936
Zebra	Set14	21.4209	0.78753	7.3752	4.2207

Table 5.2 PSNR, SSIM, NIQE Results for images with input patch size 84x84

Name	**Dataset**	**PSNR**	**SSIM**	**NIQE SR**	**NIQE HR**
0001	DIV2K	20.9021	0.86095	5.26	3.6858
0002	DIV2K	16.7123	0.77823	6.9395	5.4203
0010	DIV2K	21.2759	0.86797	5.5155	3.4753
0070	DIV2K	20.3345	0.84114	6.4365	5.3119
0800	DIV2K	22.7079	0.80121	5.5653	3.5008
Baboon	Set14	16.0322	0.74476	8.0793	6.3862
Baby	Set5	23.5478	0.81651	7.8415	3.6322
Barbara	Set14	22.1495	0.80588	6.4821	5.0425
Bird	Set5	NA	NA	6.1804	5.1477
Butterfly	Set5	NA	NA	11.4785	9.5716
Face	Set14	NA	NA	11.3155	9.8626
Img_012	Urban100	20.1839	0.79853	6.4067	4.4676
Img_045	Urban100	23.2521	0.85943	8.8549	9.4225
Img_067	Urban100	10.0316	0.68821	8.4158	5.9088
Img_078	Urban100	16.2113	0.6945	6.1667	4.1388
Img_100	Urban100	15.9917	0.82466	9.6759	8.4111
Monarch	Set14	27.7769	0.84488	5.11	4.2552
Pepper	Set14	22.168	0.78607	7.3997	7.3509
Woman	Set5	NA	NA	7.1419	5.1936
Zebra	Set14	20.8071	0.79308	8.837	4.2207

Table 5.3 PSNR, SSIM, NIQE Results for images with input patch size 124x124

Name	**Dataset**	**PSNR**	**SSIM**	**NIQE SR**	**NIQE HR**
0001	DIV2K	17.5108	0.88147	5.3755	3.6858
0002	DIV2K	17.1008	0.82556	7.1751	5.4203
0010	DIV2K	24.5804	0.89394	5.4078	3.4753
0070	DIV2K	21.8752	0.86665	6.6059	5.3119
0800	DIV2K	24.3612	0.81116	5.379	3.5008
Baboon	Set14	NA	0.73638	7.7951	6.3862
Baby	Set5	24.4424	0.81893	8.0869	3.6322

Barbara	Set14	18.9751	0.82031	6.7348	5.0425
Bird	Set5	NA	NA	6.1804	5.1477
Butterfly	Set5	NA	NA	10.3627	9.5716
Face	Set14	NA	NA	10.9811	9.8626
Img_012	Urban100	19.0361	0.79669	6.4546	4.4676
Img_045	Urban100	22.4798	0.85714	8.9528	9.4225
Img_067	Urban100	10.871	0.67687	8.2583	5.9088
Img_078	Urban100	17.6372	0.70588	6.2117	4.1388
Img_100	Urban100	17.1011	0.81644	9.6936	8.4111
Monarch	Set14	24.3686	0.85508	5.2198	4.2552
Pepper	Set14	20.1985	0.77817	7.4931	7.3509
Woman	Set5	NA	NA	8.1251	5.1936
Zebra	Set14	NA	0.77448	9.0255	4.2207

Table 5.4 Latency Figures for three deployed models

Patch_Size	Average Latency per Patch (seconds)	Average Latency per Image (seconds)
44	0.099414	14.47958
84	0.305316	14.10561
124	0.556706	12.72073

It is evident from the fact that models that input large patch sizes take more time to process models that input smaller patch sizes. On the other hand, model with input Patch size 124x124, has less latency per image than patch sizes 44x44 and 88x88.

5.2 RRDN

First of all, RRDN contains layers that do not support Vitis: 1) Lambda layer for scalar multiplication 2) depthtospace function for subpixel layer and hence cannot be deployed

The following work around was tried for deployment

5.2.1 Creating custom layers for both scalar multiplication and depthtospace:

```
class Beta(Layer): #Scalar Multiplication
  def __init__(self,beta=0.2,**kwargs):
    super(Beta,self).__init__(name="Beta",trainable=False,**kwargs)
    self.beta=beta
  def call(self,inputs):
    beta=self.beta
    return inputs*tf.constant(beta)
  def compute_output_shape(self,input_shape):
    return tuple(input_shape)
  def get_config(self):
    base_config=super(Beta,self).get_config()
    conf={"beta":self.beta,}
    return dict(list(base_config.items()) + list(conf.items()))
```

The following custom layer developed for SubPixel in [42] is used

```
class SubpixelConv2D(Layer):
    def __init__(self, upsampling_factor=4, **kwargs):
        super(SubpixelConv2D, self).__init__(**kwargs)
        self.upsampling_factor = upsampling_factor

    def build(self, input_shape):
        last_dim = input_shape[-1]
        factor = self.upsampling_factor * self.upsampling_factor
        if last_dim % (factor) != 0:
            raise ValueError('Channel ' + str(last_dim) + ' should be
 of '
                             'integer times of upsampling_factor^2: '
 +
                             str(factor) + '.')
    def call(self, inputs, **kwargs):
        return tf.nn.depth_to_space( inputs, self.upsampling_factor )

    def get_config(self):
        config = { 'upsampling_factor': self.upsampling_factor, }
        base_config = super(SubpixelConv2D, self).get_config()
        return dict(list(base_config.items()) + list(config.items()))
    def compute_output_shape(self, input_shape):
        factor = self.upsampling_factor * self.upsampling_factor
        input_shape_1 = None
        if input_shape[1] is not None:
            input_shape_1 = input_shape[1] * self.upsampling_factor
        input_shape_2 = None
        if input_shape[2] is not None:
            input_shape_2 = input_shape[2] * self.upsampling_factor
        dims = [ input_shape[0],
                 input_shape_1,
                 input_shape_2,
                 int(input_shape[3]/factor)
               ]
        return tuple( dims )
```

5.2.2 Splitting the model and deploy Vitis supported layers:

As Subpixel layer and Lambda layers are not supported, the models were split into three sections namely section1, section2 and section3. Section1 and Section3 are parts that contain Vitis supported layers and Section 2 (subpixel) was planned to be implemented in Processing System and Scalar multiplications were removed in section1. Compiling Section1 resulted in errors during compilation.

Hence, both the methodologies resulted in failure and RRDN model could not be deployed.

CHAPTER 6

CONCLUSION AND FURTHER IMPROVEMENTS

Deep Learning Processing Unit has a capacity to run larger models with low latency. Despite these benefits, it is evident from the results that the deployment of Super resolution models did not yield intended results as compared to High Resolution images. Quantisation plays a huge role in deployment of Super-Resolution deep Learning models and reduction in performance can be seen.

Quantisation can be the reason for the uneven tone of patches. One way to solve the problem is to constraint the weights of layers to a specified bound [43]. Convolution layers in Keras have an option of weight constrains where this can be achieved. Another advantage of using weight constraints is that it reduces overfitting of data and thus could be a good alternative to batch normalisation [44].

Considering Real time Application (say for video streaming), Latency has to be further reduced to per second processing. To achieve this, two possible solutions can be done: 1) Using all three DPUs, Split and deploy a model accordingly. This increases parallelism and thus reduces latency 2) Performing Knowledge distillation of super resolution models [45][46]. Knowledge distillation uses a student-teacher training framework where the teacher network with higher number of parameters teaches a student network with lesser number of parameters. As a result of parameter reduction, the number of operations reduces and hence reduces the latency

Another advantage of Knowledge Distillation is that non-deployable models like ESRGAN can be taken as teacher network and a model with Vitis supported layers can be used as student networks and thus non-deployable models can be deployed in this way.

As sub Pixel network cannot be deployed, and PixelTCN network supports Vitis layers, PixelTCN can be used as an alternative instead of Sub Pixel layer for deployment.

References

[1] A. C. Ian Goodfellow, Yoshua Bengio, “Deep Learning Book,” *Deep Learn.*, 2015, doi: 10.1016/B978-0-12-391420-0.09987-X.

[2] B. Csáji, “Approximation with artificial neural networks,” *MSc. thesis*, 2001, doi: 10.1.1.101.2647.

[3] H. Greenspan, “Super-resolution in medical imaging,” *Computer Journal*. 2009, doi: 10.1093/comjnl/bxm075.

[4] Y. Blau, R. Mechrez, R. Timofte, T. Michaeli, and L. Zelnik-Manor, “The 2018 PIRM challenge on perceptual image super-resolution,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2019, doi: 10.1007/978-3-030-11021-5_21.

[5] P. Rasti, T. Uiboupin, S. Escalera, and G. Anbarjafari, “Convolutional neural network super resolution for face recognition in surveillance monitoring,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2016, doi: 10.1007/978-3-319-41778-3_18.

[6] “Zynq UltraScale+ Device Technical Reference Manual UG1085 (v2.1).” https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf (accessed May 11, 2020).

[7] “Zynq DPU v3.1 Product Guide PG338 (v3.1),” 2019. Accessed: May 11, 2020. [Online]. Available: www.xilinx.com.

[8] “Vitis AI User Guide,” 2019. Accessed: May 11, 2020. [Online]. Available: www.xilinx.com.

[9] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding,” in *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, 2016.

[10] M. Abadi *et al.*, “TensorFlow: A system for large-scale machine learning,” in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016*, 2016.

[11] Y. Jia *et al.*, “Caffe: Convolutional architecture for fast feature embedding,” in *MM 2014 - Proceedings of the 2014 ACM Conference on Multimedia*, 2014, doi: 10.1145/2647868.2654889.

[12] V. Lempitsky, A. Vedaldi, and D. Ulyanov, “Deep Image Prior,” in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2018, doi: 10.1109/CVPR.2018.00984.

[13] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2016, doi: 10.1109/CVPR.2016.90.

[14] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” in *32nd International Conference on Machine Learning, ICML 2015*, 2015.

[15] X. Wang *et al.*, “ESRGAN: Enhanced super-resolution generative adversarial networks,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2019, doi: 10.1007/978-3-030-11021-5_5.

[16] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, “Densely connected convolutional networks,” in *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, 2017, doi: 10.1109/CVPR.2017.243.

[17] W. Shi *et al.*, “Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network,” in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2016, doi: 10.1109/CVPR.2016.207.

[18] L. Xu, J. S. J. Ren, C. Liu, and J. Jia, “Deep convolutional neural network for image deconvolution,” in *Advances in Neural Information Processing Systems*, 2014.

[19] H. Gao, H. Yuan, Z. Wang, and S. Ji, “Pixel Transposed Convolutional Networks,” *IEEE Trans. Pattern Anal. Mach. Intell.*, 2020, doi: 10.1109/TPAMI.2019.2893965.

[20] I. J. Goodfellow *et al.*, “Generative adversarial nets,” in *Advances in Neural Information Processing Systems*, 2014.

[21] A. Radford, L. Metz, and S. Chintala, “DCGAN,” *4th Int. Conf. Learn. Represent. ICLR 2016 - Conf. Track Proc.*, 2016.

[22] C. Ledig *et al.*, “Photo-realistic single image super-resolution using a generative adversarial network,” in *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, 2017, doi: 10.1109/CVPR.2017.19.

[23] M. S. M. Sajjadi, B. Scholkopf, and M. Hirsch, “EnhanceNet: Single Image Super-Resolution Through Automated Texture Synthesis,” in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, doi: 10.1109/ICCV.2017.481.

[24] J. Johnson, A. Alahi, and L. Fei-Fei, “Perceptual losses for real-time style transfer and super-resolution,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2016, doi: 10.1007/978-3-319-46475-6_43.

[25] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” in *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, 2015.

[26] A. Jolicoeur-Martineau, “The relativistic discriminator: A key element missing

from standard GaN," in *7th International Conference on Learning Representations, ICLR 2019*, 2019.

[27] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: From error visibility to structural similarity," *IEEE Trans. Image Process.*, 2004, doi: 10.1109/TIP.2003.819861.

[28] C. Ma, C. Y. Yang, X. Yang, and M. H. Yang, "Learning a no-reference quality metric for single-image super-resolution," *Comput. Vis. Image Underst.*, 2017, doi: 10.1016/j.cviu.2016.12.009.

[29] A. Mittal, R. Soundararajan, and A. C. Bovik, "Making a 'completely blind' image quality analyzer," *IEEE Signal Process. Lett.*, 2013, doi: 10.1109/LSP.2012.2227726.

[30] Y. Blau and T. Michaeli, "The Perception-Distortion Tradeoff," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2018, doi: 10.1109/CVPR.2018.00652.

[31] J. Cai, H. Zeng, H. Yong, Z. Cao, and L. Zhang, "Toward real-world single image super-resolution: A new benchmark and a new model," in *Proceedings of the IEEE International Conference on Computer Vision*, 2019, doi: 10.1109/ICCV.2019.00318.

[32] A. Bulat, J. Yang, and G. Tzimiropoulos, "To learn image super-resolution, use a GAN to learn how to do image degradation first," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2018, doi: 10.1007/978-3-030-01231-1_12.

[33] Y. Zhang, Y. Tian, Y. Kong, B. Zhong, and Y. Fu, "Residual Dense Network for Image Super-Resolution," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2018, doi: 10.1109/CVPR.2018.00262.

[34] "tf.nn.depth_to_space | TensorFlow Core v2.2.0." https://www.tensorflow.org/api_docs/python/tf/nn/depth_to_space (accessed May 11, 2020).

[35] Francesco Cardinale, "ISR," 2018. https://github.com/idealo/image-super-resolution.

[36] "tf.keras.layers.UpSampling2D | TensorFlow Core v2.2.0." https://www.tensorflow.org/api_docs/python/tf/keras/layers/UpSampling2D (accessed May 11, 2020).

[37] E. Agustsson and R. Timofte, "NTIRE 2017 Challenge on Single Image Super-Resolution: Dataset and Study," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, 2017, doi: 10.1109/CVPRW.2017.150.

[38] M. Bevilacqua, A. Roumy, C. Guillemot, and M. L. A. Morel, "Low-complexity single-image super-resolution based on nonnegative neighbor

embedding," in *BMVC 2012 - Electronic Proceedings of the British Machine Vision Conference 2012*, 2012, doi: 10.5244/C.26.135.

[39] R. Zeyde, M. Elad, and M. Protter, "On single image scale-up using sparse-representations," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2012, doi: 10.1007/978-3-642-27413-8_47.

[40] J. Bin Huang, A. Singh, and N. Ahuja, "Single image super-resolution from transformed self-exemplars," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2015, doi: 10.1109/CVPR.2015.7299156.

[41] "freeze_graph." https://gist.github.com/morgangiraud/249505f540a5e53a48b0c1a869d370bf#file-medium-tffreeze-1-py (accessed May 12, 2020).

[42] "fengwang/subpixel_conv2d: Sub-Pixel Convolutional Layer with Tensorflow/Keras." https://github.com/fengwang/subpixel_conv2d (accessed May 11, 2020).

[43] "Module: tf.keras.constraints | TensorFlow Core v2.2.0." https://www.tensorflow.org/api_docs/python/tf/keras/constraints (accessed May 12, 2020).

[44] "How to Reduce Overfitting Using Weight Constraints in Keras." https://machinelearningmastery.com/how-to-reduce-overfitting-in-deep-neural-networks-with-weight-constraints-in-keras/ (accessed May 12, 2020).

[45] Q. Gao, Y. Zhao, G. Li, and T. Tong, "Image Super-Resolution Using Knowledge Distillation," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2019, doi: 10.1007/978-3-030-20890-5_34.

[46] Z. Hui, Y. Yang, X. Gao, and X. Wang, "Lightweight image super-resolution with information multi-distillation network," in *MM 2019 - Proceedings of the 27th ACM International Conference on Multimedia*, 2019, doi: 10.1145/3343031.3351084.