FPGA-Accelerated Image Super Resolution Convolutional Neural Network

Report

by

Jakob Essbüchl 01129145
Philipp Lehninger 01327039
Benedikt Morgenbesser 01027440

24th February 2021

Vienna University of Technology
Institute of Computer Technology

Abstract

This project is part of the course System on Chip Design Lab at the Technical University of Vienna. The defined goal was to implement a convolutional neural network (CNN) as a form of digital signal processor (DSP) on a field-programmable gate array (FPGA). After this, a second DSP utilising approximate computing should be designed and both designs should be compared.

The neural network implements a Super Resolution (SR) algorithm for upscaling images. The chosen algorithm is simply called *Super Resolution Convolutional Neural Network (SRCNN)* [5]. To achieve the stated goal, the following was done in order:

1. Implement the neural network in PyTorch to train the network and gain weights for it's nodes.
2. Implement the neural network in NumPy to improve understanding of the algorithm and remove PyTorch dependency.
3. Implement the convolutions in C and wrap them in Python to allow acceleration on the FPGA.
4. Implement parts of the convolutions in the FPGA using VHDL to accelerate the upscaling operation.
    a) One implementation works with any image size but is slower than the purely CPU-run variant without FPGA acceleration.
    b) A second implementation is much faster than the CPU-run variant but is limited in image size.
5. *Create and implement an approximate DSP version of the neural network.*
6. *Compare both approaches (accurate and approximate) in image quality, performance, area, etc.*

Because of time constraints items 5 and 6 could not be completed before the deadline. Unfortunately there is an unsolved bug in the FPGA implementation that makes using larger images impossible. Before that had been solved continuing with the approximate design was not warranted.

Our implementation creates images with good image quality metrics in respect to the algorithms compared in the original paper [5]. In two metrics (SSIM and MSSSIM) it's the best of the compared papers while in three others (PSNR, IFC and NQM) it's in the better half.

As mentioned in the roadmap one FPGA implementation was actually slower than without FPGA acceleration while the other implementation was much faster (over 22 times).

Contents

1. Introduction

1.1. Single Image Super-Resolution

The aim of single image super-resolution (SR) is to recover a high-resolution (HR) image from a single low-resolution (LR) image. HR images are required in a lot of different applications. For example in medical imaging, satellite imaging, high-definition television (HDTV) etc. The problem of up-scaling images is underdetermined: For any low-resolution pixel a variety of different solutions exist. The resolution space has to be constrained to deal with that fact. To do so, prior information is needed. Approaches to get this information can be categorised in four types: prediction models, edge-based methods, image statistical methods and example-based methods. Internal example-based methods exploit the self similarity property. Exemplar patches are generated from the input image. For external example-based methods a mapping between LR & HR patches is taught from external data-sets. The majority of SR algorithms focus on grey-scale or single-channel image super-resolution. To get a coloured image from a single-channel image SR the YCbCr colour space can be used. (See section 1.2.) An example for single image super resolution is shown in figure 1. [1]

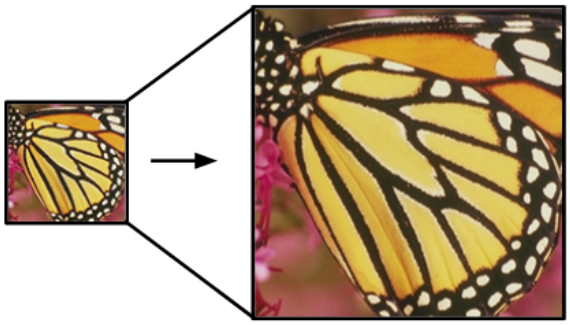


Fig. 1: Example for a single image super-resolution recovery

1.2. YCbCr

In the YCbCr colour space the image is decomposed into three channels: *luminance (Y), blue-difference chroma (Cb) & red-difference chroma (Cr).* Figure 2 shows an image and the associated channels in the YCbCr colour space. The human eye is much more sensitive to changes in luminance than to changes in the chroma channels. Therefore, it is sufficient to apply super-resolution only on the luminance channel, while using computationally cheap methods like bicubic interpolation on the chroma channels to get convincing high-resolution results.

Fig. 2: YCbCr image and channel decomposition. From left to right: Original image, Luminance (Y), Chroma blue (Cb) Chroma red (Cr). Adapted from: [14].

1.3. Neural Networks

Neural networks (NNs) are tools used for efficient optimization. NNs are structures to simulate the method of human thinking based on neurons and activation functions. These massively parallel working networks are designed, similar to the human brain, to store information in the form of patterns. NNs are well suited for image processing.

1.4. SRCNN

For image super-resolution a deep convolutional neural network (SRCNN) can be used as described in []. With this approach a direct end-to-end mapping between a low resolution input image and a high resolution output image is implemented. The only preprocessing step needed for the SRCNN is to upscale the input image to a desired size. This can be done by bicubic interpolation. Thus the 'low-resolution' input image for the neural network has already the same size as the output image. In this context the terms 'low- & high-resolution' actually refer to the image quality and not the image size. The goal of the CNN is to recover an image $\mathbf{X}$ from the input image $\mathbf{Y}$ that is as similar as possible to the ground truth. (The ground truth is the ideal solution). The mapping F($\mathbf{Y}$) consists of three operations: *Patch extraction and representation*, *non-linear mapping*, and *reconstruction*. These operations form the convolutional neural network as shown in figure 3. The purposes of the three steps are:

1. **Patch extraction and representation:**
   Patches from the LR input image are extracted in this step. For each patch a high-dimensional vector that comprises a set of n_1 feature maps is generated. For a single channel input image this layer gives a n_1 channel output.

2. **Non-linear mapping:**
   In this layer the high-dimensional vectors from layer 1 are non-linearly mapped onto other n_2-dimensional vectors. Conceptually, each of these n_2 vectors represent now a high-resolution patch.

3. **Reconstruction:**
   In the last layer the final image is reconstructed from the high-resolution, patch-wise representation from layer 2. The output of this layer is the single channel, high resolution image that ideally fits the ground truth.

Although the layers comprise of apparently different operations they lead to the same form as convolutional layers. The filtering weights and biases of the CNN are subject to optimisation by training. [5].

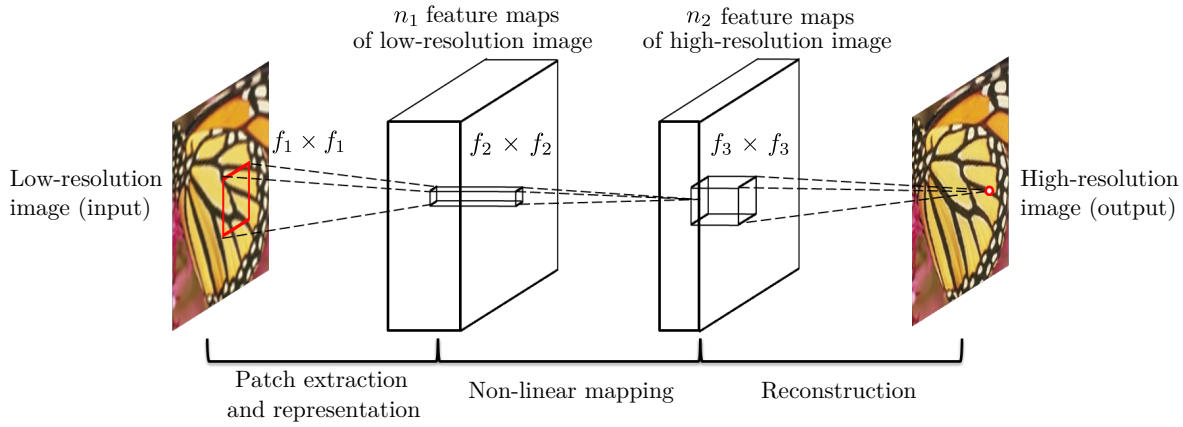
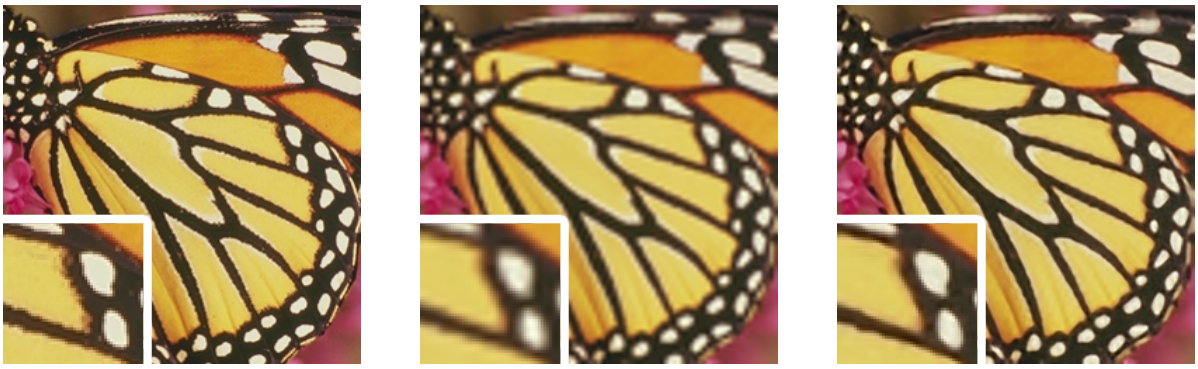


Fig. 3: Convolutional layers of the SRCNN. Source: [5].

An example for a SR image is shown in figure 4. The right-most image is the ground truth, i.e. the ideal solution. The image on the left side was upscaled with a simple bicubic interpolation. In the centre the output of the super-resolution deep convolutianal network is shown.



(a) Ground truth (b) Bicubic interpolation x3 (c) SRCNN x3

Fig. 4: Comparison of ground truth, bicubic interpolation and SRCNN.

2. Implementations

Three implementations using different Python modules and sometimes different hardware were done and will be described in the following sections.

All three implementations follow the same general program structure:

1. Load weights & biases
2. Open image
3. Downsize image to the closest multiple of the chosen scale factor. This becomes the *ground truth image (GT)* used as a comparison with the upscaled images.
4. Downsize the GT image by using the chosen scale factor.
5. Upscale the downsized image by the chosen scale factor using bicubic interpolation. This is the *bicubic interpolation image*.
6. Convert the image from RGB colour space to YCbCr colour space, extract the Y channel and scale the pixel values from 0-255 to 0-1.
7. Upscale the Y channel by the chosen scale factor using the neural network. Roughly speaking this means calculating a lot of 2D convolutions which themselves consist of a lot of multiplications and additions.
8. Scale the pixel values back from 0-1 to 0-255
9. Combine the Y channel with the Cb and Cr channels from before and save the image. This is the upscaled *SRCNN image*.
10. Calculate the image metrics PSNR and SSIM as well as execution time.

2.1. PyTorch

For the first implementation of the SRCNN PyTorch was used. PyTorch is an open-source machine learning library that allows an easy and efficient implementation of neural networks. A key feature of PyTorch is that calculations can be accelerated using the CUDA Cores of Nvidia GPUs. However, there are two main drawbacks:

First, most of the calculations when upscaling an image happen inside the functions of the PyTorch library. In this way it acts similarly to a black box. This would not allow accelerating parts of the calculations on the FPGA since there is no easy access. While possible in theory it would mean reading into and understanding the inner workings of the very complex PyTorch library.

Secondly, PyTorch is a large library with a size of several GB and does not offer builds for ARM CPUs, which the ZedBoard is using. This would require building the whole library on the ZedBoard itself which would take hours due to the slow CPU and is prone to errors because of missing dependencies.

In the end PyTorch was used not only as a first implementation, but also for playing around with different network sizes, training the network and extracting weights and biases to use with the other two implementations.

2.2. NumPy

Since PyTorch is such a large library and the convolution calculations are hidden inside the library functions it wasn't feasible to use it together with FPGA acceleration. Easiest access to the FPGA hardware would be with pure C code but we wanted to keep as much of code in python as possible to make development easier and faster.

Thus, the decision was made to remove the PyTorch dependency and try to implement the convolutions in NumPy. This was both for understanding how to actually 'manually' calculate the two-dimensional convolutions as well as serve as a stepping stone to the FPGA hardware. The plan was to possibly port the large amount of multiplications that the convolutions consist off over to C and go from there to the FPGA hardware.

In practice it turned out that the NumPy implementation was *very* slow. An image that took about 3 seconds in PyTorch took 14 minutes in NumPy (see Table 3). Part of this is because of for loops which are notoriously slow in Python. Of four nested for loops, two could be removed by utilising NumPy's vectorised operations. These are executed in heavily optimised C (which NumPy is mostly written in in the back-end). Two loops remained nonetheless and are probably the root cause for the slow execution time.

This implementation could likely be optimised much further, but since we needed to switch to C anyway we decided to leave it as is.

2.3. Cython

The Cython implementation is a direct derivative of the NumPy implementation. The 2D convolutions were completely rewritten in C while NumPy was kept for type conversions and scaling of pixel values prior to and after the convolutions. The convolutions are imported into the python code as an external library using the ctypes module.

All of the convolution operations as well as weights and biases were converted from 32-bit floating point number format to 32-bit Q6.26 fixed-point integer format. This was done because it's much easier to implement fixed-point arithmetic in the FPGA compared to floating point arithmetic. Also, with pixel values originally having 256 possible values between 0 and 255, there is not much dynamic range needed, making a fixed-point implementation a good fit.

The Cython implementation has three sub-implementations:

1. All calculations run on the CPU, single-threaded
2. 2D Convolutions run on the FPGA, the rest runs on the CPU. Works with any image size but is slow (referred to as FPGA1)
3. Several improvements upon 2. Only works with images up to a certain size but is much faster (referred to as FPGA2)

These will be described in detail in the following section.

Both implementations with HDL designs on the FPGA fill the streams to the FPGA concurrently with child processes.

2.4. FPGA Implementations

2.4.1. Number Representation

The data written to and read from the design on the FPGA is in the 32-bit Q6.26 fixed point format. This means that the last 26 bits are fractal digits whereas the highest 6 bits represent an integer. Note that the MSB is a sign bit, negative values are expressed in two's complement.

The C code accessing the hardware and the pure software implementation (Cython) also use this number representation, the Python script converts all floating point numbers to 32-bit Q6.26 fixed point representations.

2.4.2. General Overview

The core task of the hardware design is to apply a kernel multiplication to a selected image/feature patch. Therefore, the patch and kernel data pairs have to be acquired concurrently via two separate streams.

The kernel multiplication multiplies each kernel value with its corresponding value from the patch, the results are accumulated to one pixel value. For example, the kernel multiplication of a 5x5 kernel (kernel size 5) with a 5x5 patch will execute 25 fixed point multiplications, add the products up and yield one fixed point number. This behaviour is described in the VHDL file *patch_mult.vhdl*.

The main difference between the two HDL implementations is, that the *patch_multi plier_pipeline*, also referred to as 'FPGA Implementation 1', just performs this kernel multiplication, however, the second HDL design (*feature_convolution_pipeline*, 'FPGA Implementation 2'), also has to do the patch selection and the padding at the edges of the feature/image.

2.4.3. FPGA Implementation 1

The first FPGA implementation reads patch and kernel data from two separate FIFOs which are filled by two separate asynchronous Xillybus streams (xillybus_write_patch_32 and xillybus_write_kernel_32). As the basic Xillybus IP core, that is contained in the basic xillydemo project has only one 32-bit write stream (downstream, CPU to FPGA), a custom Xillybus IP Core had to be generated.

The relevant interfaces of this custom IP core are:

- xillybus_write_patch_32: 32-bit asynchronous stream for patch data
- xillybus_write_kernel_32: 32-bit asynchronous stream for kernel data
- xillybus_read_32: 32-bit asynchronous stream reading back the results

The multiplier has been generated via the Vivado IP Catalog. It has two pipeline stages and a Clock-Enable pin for stall functionality. This IP Core is configured to use the integrated multipliers on the FPGA instead of LUTs. A combinatorial adder and a register form the accumulator and therefore the third pipeline stage of the *patch_mult* entity. The output is ready, when a counter has counted through all kernel entries.

The result of the kernel multiplications is written into a FIFO that is connected to the output stream. All three FIFOs are single-clock non-FWFT (First Word Fall Through) 32-bit FIFOS with a depth of 512 values.

2.4.4. FPGA Implementation 2

The second implementation contains several improvements. At first, a new custom Xillybus IP core has been generated:

- xillybus_config: 32-bit synchronous stream for storing config data
- xillybus_command: 8-bit synchronous stream for control flow commands
- xillybus_write_feature_32: 32-bit asynchronous stream for feature data
- xillybus_write_kernel_32: 32-bit asynchronous stream for kernel data
- xillybus_read_32: 32-bit asynchronous stream reading back the results

The config stream allows to set the height and width of the image, which is a requirement to be able to write a whole feature instead of patches to the FPGA logic. This is necessary because the logic design has to iterate through all pixel positions of a feature and do the patch extraction by itself. Moreover, the kernel size is not a VHDL generic anymore but can also be set via the config stream.

The feature and kernel data is still acquired via two separate asynchronous streams, however, the data is not written into FIFOs but into BRAM. For the patch extraction and padding logic the data has to be addressable and accessible multiple times. Counters

calculate the addresses and check if a whole feature and kernel have been written. When all necessary data is present, the convolution of this one feature can start. As for padding the read addresses of the feature RAM have to be altered and multiplications in address calculations have to be avoided in general, so the address calculation is non-trivial and surely the most complex part of this hardware implementation.

The patch extraction logic will yield a value of a patch and its corresponding kernel value from the memory and pass these two values to the *patch_mult* entity from the first implementation. The results will again be written into an output FIFO.

These changes promise a faster execution but also lead to new limitations:

- memory capacities limit the maximum size of features and kernels
- data flow is more sporadic, DMA buffers have to be flushed regularly
- synchronization of hardware accesses gets a bit more complicated

To enable to send control commands to the hardware the synchronous 8-bit command stream has been added to the Xillybus IP core. In the current implementation it has been used to skip feature re-transmissions. Thereby, the loops for output and input channels can be interchanged in the software and each feature has to be sent only once. The idea was, that the software can send a skip command instead of re-sending the whole feature.

However, this functionality still has synchronisation problems and the resulting images are damaged (refer to Section 7.4.1 for details).

3. Development platform

3.1. ZedBoard

For this project a ZedBoard development board was used. It includes the Zynq®-7000 All Programmable SoC with a programmable logic and an ARM®-based processing system. The Ethernet port was used for communication via SSH. The SD card includes the boot files and the Linux file-system of the embedded Xillinux. A photo of the board is shown in figure 5.

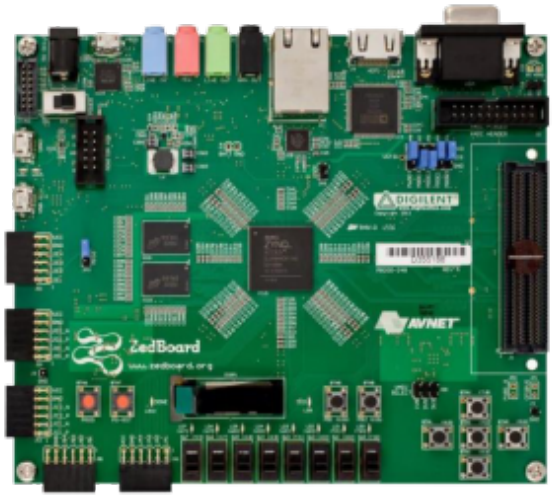


Fig. 5: The ZedBoard development board. Source: [15].

3.2. Xillinux

Xillinux is a Linux distribution for different FPGA boards including the ZedBoard. It is based upon Ubuntu LTS 16.04 for ARM. The embedded OS uses the SD card as hard disk and supports plug & play for various devices, like a USB mouse or keyboard. Xillinux also provides a graphical user interface. A computer screen can be connected via VGA or HDMI. The relevant software can be downloaded from: http://xillybus.com/xillinux. It includes an SD card image for the operating system and a boot partition kit. An introduction on how to build Xillinux (generate bitstream, load SD image..) is given in the Getting started with Xillinux for Zynq-7000 document.

3.3. Xillybus IP

For the data transport between Xillinux and the FPGA Xillybus IP cores were used. Xillybus consists of an FPGA IP core and a Linux driver. Xillybus uses direct memory access (DMA) and the AXI bus. The communication between the user application and the IP core happens through standard FIFO's and BRAM. A custom Xillybus IP core can be configured in the IP core factory. A simplified block diagram for the Xillybus IP core is shown in figure 6.

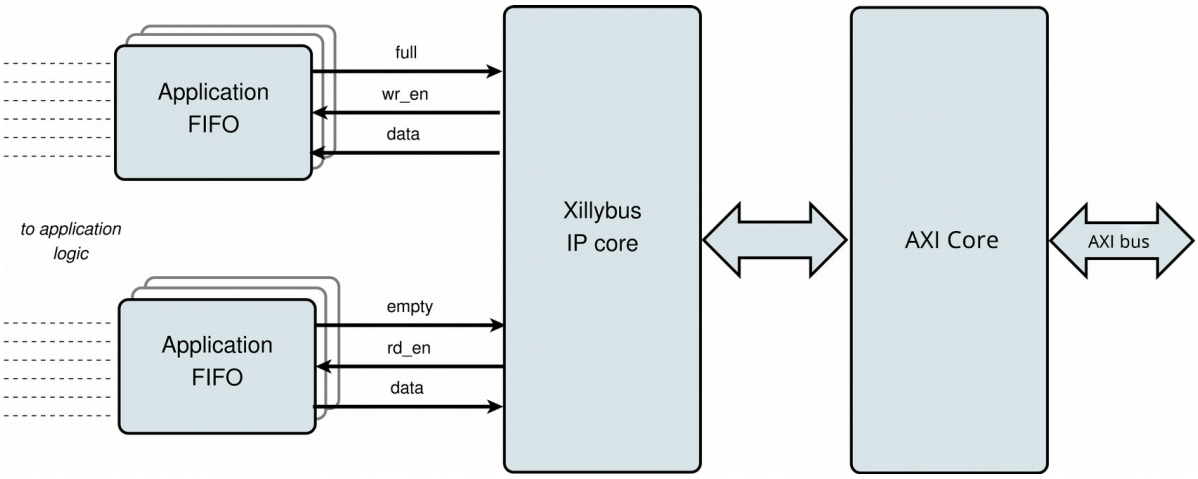


Fig. 6: Simplified block diagram for the Xillybus IP core. Adapted from: [12].

Note: Xillinux and Xillybus have several license variants. A no-fee educational license is granted for student projects. The free license is only applicable, if no commercial interest is followed.

4. Evaluation - Image Quality

For the evaluation of the quality of the upscaled image, the result has to be compared with the ground truth. The ground truth is the ideal solution. For a real world usage scenario the ground truth is naturally not known. To evaluate the performance of the algorithm, the original image is sampled down. The LR image is then used as input. The original image acts as ground truth and can be compared with the output image. Both scenarios are illustrated in figure 7.

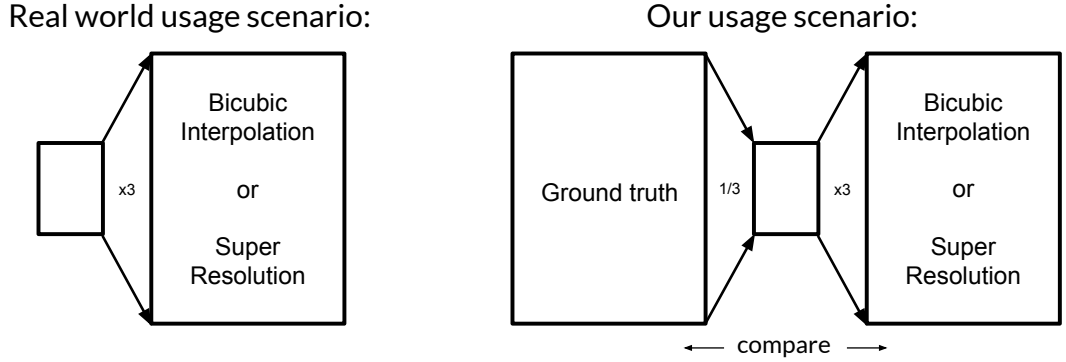


Fig. 7: Left: Real world usage scenario. An input image is fed to the network and upscaled. Right: The original image acts as ground truth. The image is scaled down and up again.

4.1. Metrics

There are different metrics to evaluate the difference between ground truth and output image. The two most important metrics are PSNR and SSIM. The approaches are described in section 4.1.1 & 4.1.2. Other metrics used to evaluate the image quality are discussed in section 4.1.3.

4.1.1. PSNR

The peak signal to noise ratio (PSNR) between two images can be used as a straightforward image quality measure. The ratio in dB can be computed using equation 1.

$$PSNR = 10 \cdot log_{10}\left(\frac{R^2}{MSE}\right) \tag{1}$$

R is the maximum fluctuation of the input image data type. Hence, for a pixel value between 0 and 255 $R = 255$. MSE is the mean squared error between the ground truth and the output image. For M rows and N columns the MSE can be calculated as seen in equation 2.

$$MSE = \frac{\sum_{M,N}[I_1(m,n) - I_2(m,n)]^2}{M \cdot N} \tag{2}$$

Colour images are usually converted to the YCbCr space and the PSNR is calculated for the luminance channel. (See section 2). Naturally, a high PSNR is desired. [3]

4.1.2. SSIM

The structural similarity index (SSIM) was developed as an improved image quality measure. While the PSNR calculates the absolute deviation between two images, SSIM is a method that also takes the human perception into account. The SSIM combines three components, a luminance comparison (l), a contrast comparison (c) and a structure comparison (s). (see equation 3). The relative importance of the terms can be set with the parameters: α, $\beta \,\&\, \gamma$. The variables $x \,\&\, y$ refer to the two images.

$$SSIM(x,y) = [l(x,y)]^{\alpha} \cdot [c(x,y)]^{\beta} \cdot [s(x,y)]^{\gamma} \tag{3}$$

In [16] the formula for the structural similarity index is derived to equation 4. With the average value μ and the variance σ of the images. The coefficients $C_1 \,\&\, C_2$ serve as stabiliser for a small denominator.

$$SSIM(x,y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)} \tag{4}$$

The structural similarity index ranges from 0 to 1. A SSIM of 1 implies two identical images, a SSIM of 0 two complete different ones. [16]

4.1.3. Other Metrics

Other metrics used to compare the image quality are MSSSIM, IFC & NQM.

1. The multi-scale structural similarity index (MSSSIM) is a more advanced form of the structural similarity index. The method incorporates details of the image at different resolutions. The image is therefore iteratively low-pass filtered and downsampled. The MSSSIM index is a combination of the measures at these different scales. The optimal coincidence is given by a MSSSIM index of 1. [11]

2. The information fidelity criterion (IFC) analyses the image fidelity in an information-theoretical framework. The mutual information between the input (ground truth) and the output (upscaled image) is quantified statistically. A higher score indicates a better similarity. [8]

3. The noise quality measure (NQM) takes variation in contrast sensitivity, variation in the local luminance mean, contrast interaction between spatial frequencies and contrast masking effects into account. The NQM assesses additive noise in images. It is given in dB. [4]

4.2. Results

Table 1 compares the different metrics for the image quality of our SRCNN implementation, with the one from [5], the bicubic interpolation and various other image enhancing algorithms. These are:

1. *SC* - sparse coding based method [13],
2. *NE+LLE* - neighbour embedding + locally linear embedding method [2],
3. *KK* - A method developed by Kim, K.I. and Kwon, Y. [6],
4. *ANR* - Anchored Neighbourhood Regression method [10], and
5. *A+* - Adjusted Anchored Neighbourhood Regression method [9].

The values in table 1 are the average values for the images from the Set5 dataset. The set of images is attached in Appendix A, the individual scores are listed in Appendix B. For IFC & NQM the A+ algorithm shows the best results. The SRCNN implementation by Dong shows the best PSNR. Our implementation of the SRCNN achieves the best index for SSIM. For MSSSIM these three approaches share the highest achieved score. The relative improvement of the SRCNN compared to a bicubic interpolation is shown in table 2.

The results confirm the qualitative impression that the algorithm performs in particular well with images with sharp edges as it is the case in the 'butterfly.bmp' (figure 10) image. For images that consist mostly of lightly textured surfaces like the 'baby.bmp' (figure 11) the improvement is smaller.

Metric	Bicubic	SC [13]	NE+LLE [2]	KK [6]	ANR [10]	A+ [9]	SRCNN (Dong) [5]	SRCNN (Ours)
PSNR	29.56	31.42	31.84	32.28	31.92	32.59	**32.75**	31.92
SSIM	0.871	0.882	0.896	0.903	0.897	0.909	0.909	**0.913**
IFC	3.49	3.16	4.40	4.14	4.52	**4.84**	4.58	4.41
NQM	27.93	27.29	32.77	32.10	33.10	**34.48**	33.21	33.04
MSSSIM	0.975	0.980	0.984	0.985	0.984	**0.987**	**0.987**	**0.987**

Table 1: The average results of PSNR (dB), SSIM, IFC, NQM (dB), and MSSSIM on the Set5 dataset.

Metric	baby.bmp	bird.bmp	butterfly.bmp	head.bmp	woman.bmp
PSNR	39 %	88 %	149 %	25 %	86 %
SSIM	2 %	3 %	11 %	3 %	5 %
IFC	15 %	25 %	46 %	16 %	29 %
NQM	-3 %	343 %	412 %	257 %	361 %
MSSSIM	0.7 %	0.9 %	2.1 %	0.9 %	1.3 %
Average	11 %	92 %	124 %	60 %	97 %

Table 2: Relative improvement of image quality metrics between SRCNN and bicubic interpolation.

5. Evaluation - Execution Time

Besides the image quality, the execution time was also investigated. The different implementations of the SRCNN are compared in table 3. The measurement of the execution time is done in Python. The basis for the execution times are the original ('butterfly.bmp') image from the Set5 dataset (256 x 256 px size) and smaller version (99 x 99 px size) of the same image.

5.1. Results

Table 3 shows execution time for the different implementations of the SRCNN for the two image sizes. A visualization of this data is shown in figure 8. As expected, the PyTorch-based algorithm shows the best performance. This is because calculations are accelerated on the GPU. PyTorch is an optimised library and the utilised hardware is much faster than the one on the ZedBoard. The second fastest execution time is achieved by the FPGA2 implementation which shows the acceleration potential of specialised hardware. It is over 22 times faster than the pure CPU variant running on the ZedBoard.

Even with the strong desktop hardware, the NumPy implementation is very slow (up to 350 times slower than GPU-accelerated PyTorch and up to 33 times slower than the purely CPU-using C variant). This is due to the inefficient implementation and could be vastly improved, as discussed in section 2.2.

Implementations with very short execution times (PyTorch and Cython on the desktop CPU) show very little difference in execution times between image sizes. This is because of (mostly) static set up times. The program does not only measure the time the 2D convolutions take but also all of the prior and subsequent actions of the program. These change very little to not at all between image sizes and implementations.

		Execution time			
		256 x 256 px		99 x 99 px	
Implementation	Device	s	min	s	min
PyTorch (CPU & GPU)	Desktop[1]	2.64	0.04	2.43	0.04
NumPy (CPU)	Desktop	924	14.4	559	9.3
Cython (CPU)	Desktop	29	0.5	17	0.3
Cython (CPU)	ZedBoard[2]	631	10.5	195	3.2
Cython (FPGA 1)	ZedBoard	1413	23.6	227	3.8
Cython (FPGA 2)	ZedBoard	N/A	N/A	8.67	0.14

Table 3: Execution times for a 256 x 256 px & a 99 x 99 px image 'butterfly.bmp'.

The FPGA2 implementation does not have an execution time for the 256 x 256 px sized image because of an unsolved bug with larger images. For details on this bug see section 7.4.1.

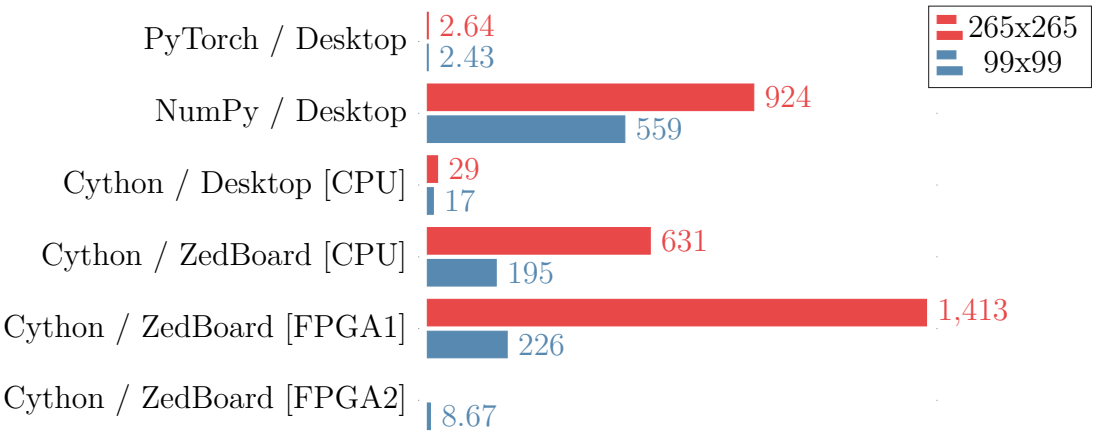


Fig. 8: Visualization of the execution times in seconds of table 3.

[1]CPU: Intel Core i5-8400 2.80GHz GPU: Nvidia Geforce GTX 1070TI
[2]CPU: Zynq-7000 SoC XC7Z020-CLG484-1; Dual ARM Cortex-A9 MPCore 667 MHz

6. Evaluation - Space

		FPGA1		FPGA2	
Resource	Available	Utilization	%	Utilization	%
LUT	53200	3484	6.6	4233	8.0
LUTRAM	17400	271	1.6	233	1.3
FF	106400	3927	3.7	4191	3.9
BRAM	140	4	2.9	100	71.1
DSP	220	4	1.8	4	1.8
IO	200	81	40.5	81	40.5
BUFG	32	2	6.3	2	6.3
PLL	4	1	25.0	1	25.0

Table 4: Resource utilization of the FPGA for FPGA implementation 1 & 2.

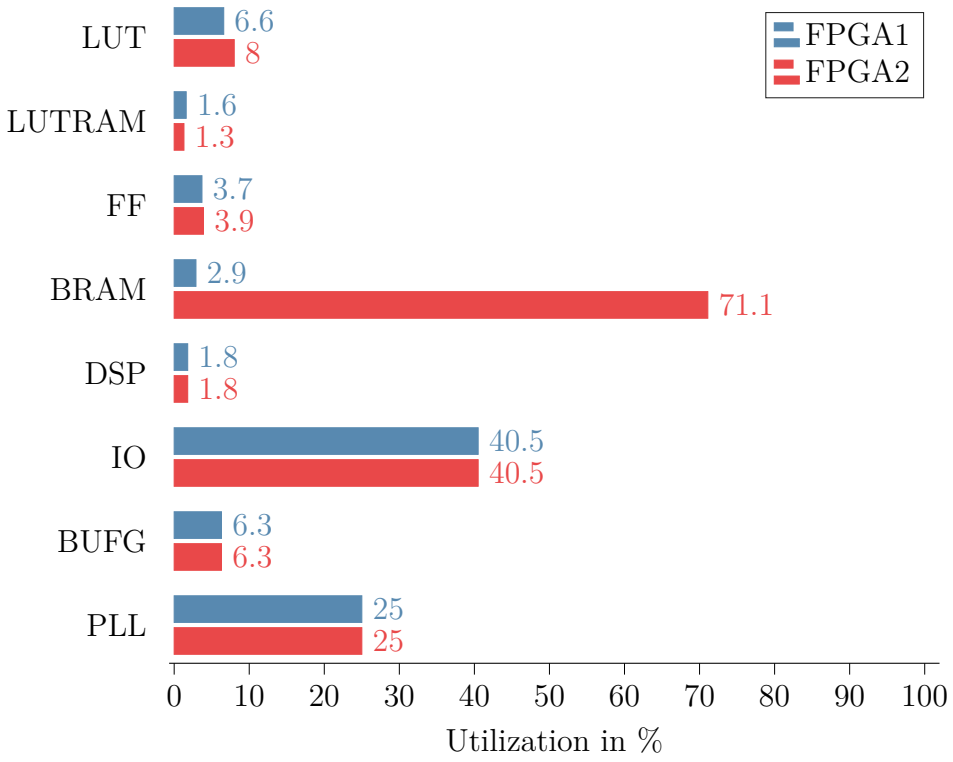


Fig. 9: Visualization of the resource utilization of the FPGA in % of available space.

7. Discussion

7.1. Image Quality

As seen in table 1 our implementation of the SRCNN shows good results with regard to the quality of the upscaled image. The implementation achieves the highest score for two of the five metrics, SSIM and MSSIM (for a detailed look refer to section 4. The algorithm is particularly suitable for images with hard edges. With textured surfaces it only gets a small qualitative improvement over bicubic interpolation. To achieve better results in this area, a larger and much more complex neural network would be needed.

The largest relative improvement compared to bicubic interpolation is achieved for the butterfly image (see table 2). The butterfly (see figure 10) has a lot of sharp edges which suits the algorithm. The improvement is less visible for example for the image of the baby (see figure 11), an image where the relatively untextured skin dominates. The lack of edges and structure naturally results in a lack of features for the algorithm.

7.2. Thoughts on Parallelization

The idea to accelerate the image convolution with a large amount of hardware multipliers soon turned out to be very difficult to implement. The main problem is that the data arrives sequentially on the FPGA design, therefore the design tends to be pipelined but not fully parallel. Multi-port RAM would allow to partially parallelize the design, but eventually only dual-port RAM could be implemented with the on-chip BRAM.
It is possible to get more ports if some identical RAM blocks are mirrored, but this multiplies the amount of used BRAM and the limitation on the maximum achievable image sizes is hard enough. As all multiplications of a kernel multiplication have to be accumulated, the accumulator would become the bottleneck. A tree of adders, like a *Wallace Tree Adder* [7], would be a possibility to optimise such a design, so that it could make use of parallel multiplications.

With the use of dual-port BRAM, the throughput of the HDL design could theoretically be nearly doubled, however, it has to be considered that also the complexity of the patch extraction and padding logic would increase significantly.

7.3. Thoughts on Approximation

As the second FPGA implementation has been added at a very late stage of this project and still has some bugs, approximation of the multiplier or the accumulator has not been considered anymore.

The whole design runs with the 100 MHz bus_clk of the Xillybus IP core. Therefore, a speed-up of either of those logic blocks would only be interesting, if clock cycles / pipeline stages could be saved, but not in order to increase the possible maximum clock frequency.

For the multiplier, the target for an approximated version would have to be to fit in just one pipeline stage. In case of the adder, only the speed-up of an adder tree (e.g. Wallace tree adder [7]) would make sense, this would require an at least partially parallelized design reading the data for the pipeline from multi-port RAM.

Therefore, future work could include a check if a dual-read-port BRAM is applicable and then possibly the approximation of an adder tree.

Another thing to consider is whether to approximate globally or just parts of the calculations. While global approximation would yield the largest possible time/area savings the detrimental effect on image quality might be too strong. A compromise would be to not approximate but there, where approximation errors have the smallest impact on image quality. Of course this introduces a lot of additional complexity into development and it's questionable whether it's worth the effort.

7.4. Limitations and Solutions

Especially the second FPGA implementation seems to be promising, but has still some bugs or rather room for optimisation. Bugs and ideas for improvements are listed here:

7.4.1. Dark image bug

When running the FPGA2 implementation on images with sizes above a certain size (more than 99x99 pixels but less than 256x256 pixels), the resulting images will be extremely dark. The original image is still recognisable but colours are shifted. Very bright spots turn completely black.

After this bug has occurred once, all future results will show this bug, even if the image size is small. A system reboot will be necessary to get correct results again. Up to now, it has not been determined from where this problem originates from. It's not even clear if it's a software memory bug or a bug of the HDL implementation.

This bug is the reason why no execution speed for the 256x256 pixel butterfly image is listed. The implementation calculates the resulting image very quickly (over 22 times faster than without acelleration), but the result will not be correct.

7.4.2. Feature Re-Transmission Skip

As already mentioned in section 2.4.4, the second HDL implementation should also support a functionality named 'feature re-transmission skip'. The software could send a skip command instead of re-sending the whole feature. Thus, the loops for output and input channels can be interchanged in the software and each feature has to be sent only once.

Due to the fact that there seem to be still synchronization problems, this functionality has not been used for measurements and is currently not used in the C code. If this would be fixed, this should allow a further speed-up.

7.4.3. Handling BRAM Limitations

Even if the dark image bug was fixed, the maximum image sizes would be limited by the BRAM size. A 300x300 pixel image occupies 71 % of BRAM. However, bigger images could still be upscaled if a stitching algorithm was implemented. The algorithm would split the image into smaller padded sub-images before passing them to the convolution algorithm and recombines them afterwards.

A. Appendix: Set5 Images

(a) Ground truth (b) Bicubic interpolation x3 (c) SRCNN x3

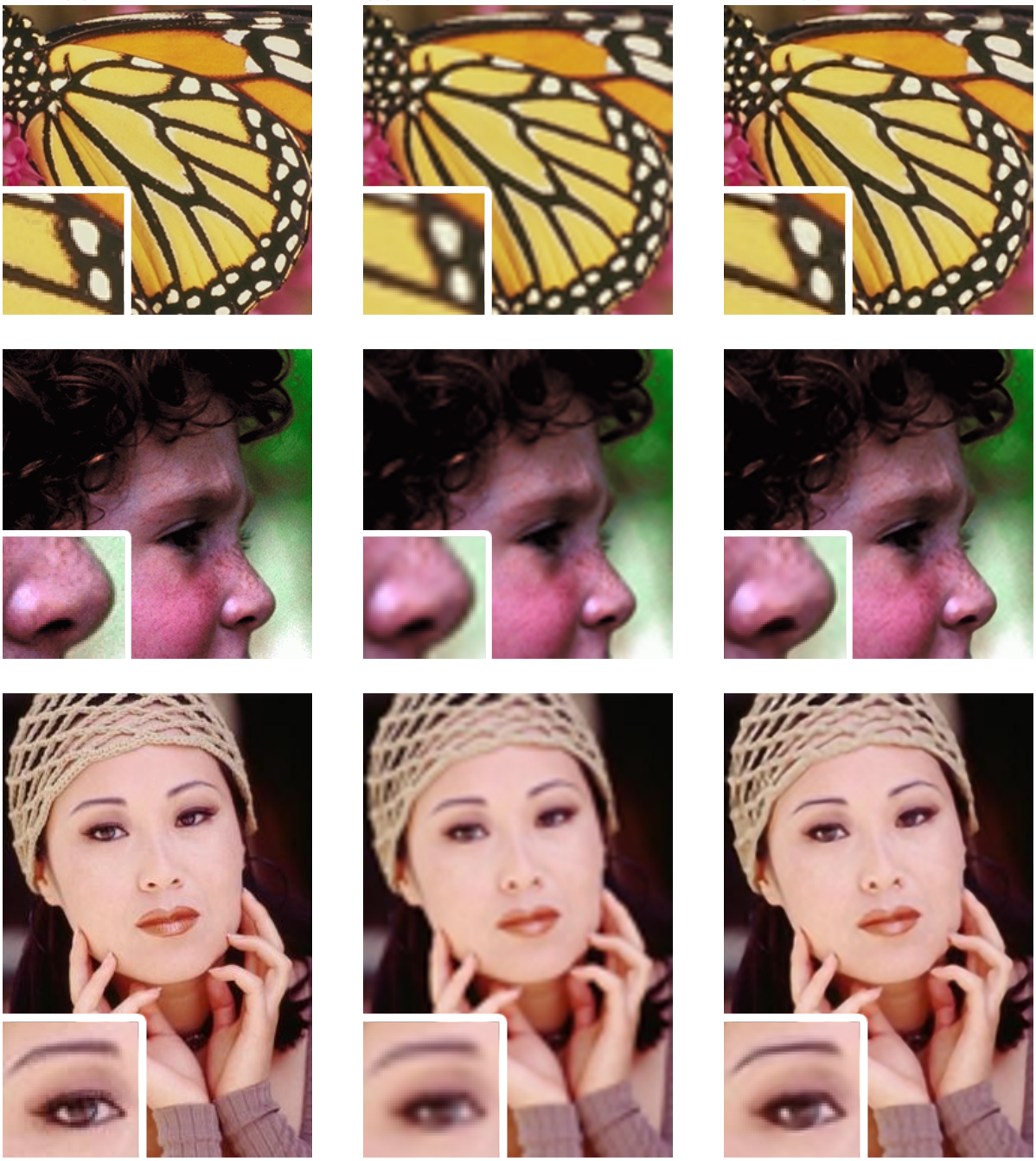


Fig. 10: Comparison of ground truth, bicubic interpolation and SRCNN of Set5 images 'butterfly', 'head' and 'woman'

(a) Ground truth (b) Bicubic interpolation x3 (c) SRCNN x3

Fig. 11: Comparison of ground truth, bicubic interpolation and SRCNN of Set5 images 'bird' and 'baby'.

B. Appendix: Quality Measures

Metric	baby.bmp	bird.bmp	butterfly.bmp	head.bmp	woman.bmp
PSNR [dB]	33.30	31.15	23.15	32.81	27.37
SSIM	0.907	0.919	0.823	0.824	0.884
IFC	3.65	3.85	3.42	3.23	3.33
NQM [dB]	41.58	24.77	19.57	30.32	23.38
MSSSIM	0.981	0.983	0.969	0.969	0.975

Table 5: Results for the images from dataset5 for **bicubic** interpolation.

Metric	baby.bmp	bird.bmp	butterfly.bmp	head.bmp	woman.bmp
PSNR [dB]	34.72	33.89	27.12	33.78	30.07
SSIM	0.927	0.949	0.914	0.851	0.925
IFC	4.21	4.79	5.00	3.73	4.32
NQM [dB]	41.45	31.24	26.67	35.84	30.02
MSSSIM	0.988	0.992	0.989	0.977	0.988

Table 6: Results for the images from dataset5 for our **SRCNN** implementation.

Team

A flat hierarchy was used. Neural networks were new to all team members, so everyone was participating in research, coding, etc. Nonetheless various areas were appointed to each member. They ultimately take main responsibility for the respective areas.

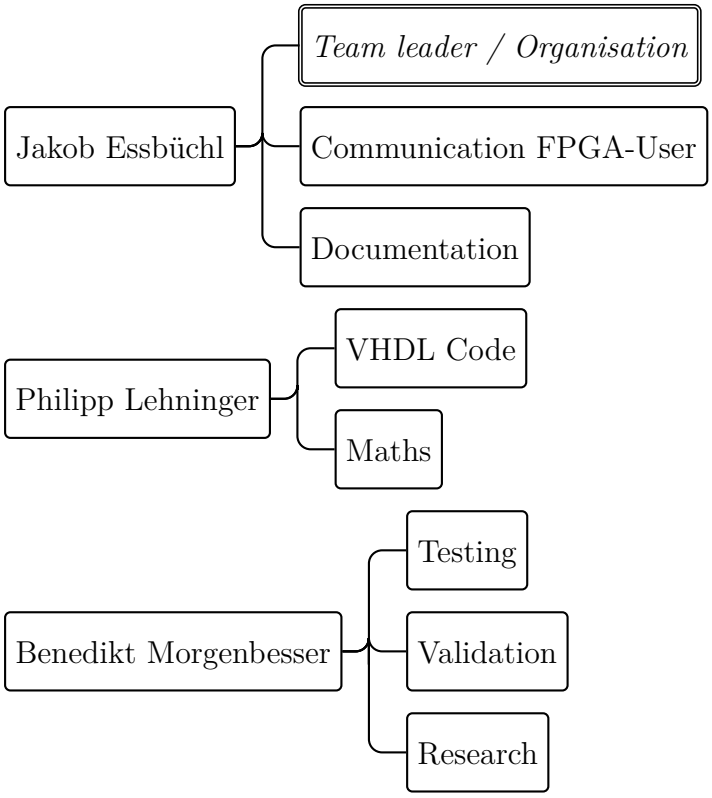


Fig. 12: Assigned areas for each team member. Team members did not work solely on their assigned areas but had the main responsibility for them.

References

[1] Fathi E Abd El-Samie, Mohiy M Hadhoud and Said E El-Khamy. *Image super-resolution and applications*. CRC press, 2012.

[2] Hong Chang, Dit-Yan Yeung and Yimin Xiong. 'Super-resolution through neighbor embedding'. In: 1 (2004).

[3] *Compute peak signal-to-noise ratio (PSNR) between images*. (Online; accessed 17.Feb.2021). URL: `https://de.mathworks.com/help/vision/ref/psnr.html`.

[4] Niranjan Damera-Venkata et al. 'Image quality assessment based on a degradation model'. In: *IEEE transactions on image processing* 9.4 (2000), pp. 636–650.

[5] Chao Dong et al. 'Image super-resolution using deep convolutional networks'. In: *IEEE transactions on pattern analysis and machine intelligence* 38.2 (2015), pp. 295–307.

[6] Kwang In Kim and Younghee Kwon. 'Single-image super-resolution using sparse regression and natural image prior'. In: *IEEE transactions on pattern analysis and machine intelligence* 32.6 (2010), pp. 1127–1133.

[7] Sakshi Sharma and Pallavi Thakur. 'Design of high speed power efficient wallace tree adders'. In: *Available at SSRN 2777887* (2016).

[8] Hamid R Sheikh, Alan C Bovik and Gustavo De Veciana. 'An information fidelity criterion for image quality assessment using natural scene statistics'. In: *IEEE Transactions on image processing* 14.12 (2005), pp. 2117–2128.

[9] Radu Timofte, Vincent De Smet and Luc Van Gool. 'A+: Adjusted anchored neighborhood regression for fast super-resolution'. In: *Asian conference on computer vision*. Springer. 2014, pp. 111–126.

[10] Radu Timofte, Vincent De Smet and Luc Van Gool. 'Anchored neighborhood regression for fast example-based super-resolution'. In: *Proceedings of the IEEE international conference on computer vision*. 2013, pp. 1920–1927.

[11] Zhou Wang, Eero P Simoncelli and Alan C Bovik. 'Multiscale structural similarity for image quality assessment'. In: *The Thrity-Seventh Asilomar Conference on Signals, Systems & Computers, 2003*. Vol. 2. Ieee. 2003, pp. 1398–1402.

[12] *Xillibus IP core product brief*. (Online; accessed 12.Feb.2021). 2020. URL: `http://xillybus.com/downloads/xillybus_product_brief.pdf`.

[13] Jianchao Yang et al. 'Image super-resolution via sparse representation'. In: *IEEE transactions on image processing* 19.11 (2010), pp. 2861–2873.

[14] *YCbCr — Wikipedia, The Free Encyclopedia*. (Online; accessed 11.Feb.2021). URL: `https://de.wikipedia.org/wiki/YCbCr-Farbmodell#/media/Datei:Barns_grand_tetons_YCbCr_separation.jpg`.

[15] *ZedBoard*. (Online; accessed 12.Feb.2021). URL: `http://zedboard.org/product/zedboard`.

[16] Hamid Rahim Sheikh Zhou Wang Alan Conrad Bovik and Eero P. Simoncelli. 'Image Quality Assessment: From Error Visibility to Structural Similarity'. In: *IEEE transactions on image processing* 13.4 (2004), pp. 600–612.