Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

David Fosca Gamarra

Flare Attenuation Filter

Master's thesis in Embedded Computing Systems
Supervisor: Professor Per Gunnar Kjeldsberg
Co-supervisor: MSc Henrik Sundbeck (Sony Semiconductors EU)
June 2023

NTNU
Norwegian University of
Science and Technology

SONY

David Fosca Gamarra

Flare Attenuation Filter

Master's thesis in Embedded Computing Systems
Supervisor: Professor Per Gunnar Kjeldsberg
Co-supervisor: MSc Henrik Sundbeck (Sony Semiconductors EU)
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

NTNU
Norwegian University of
Science and Technology

Abstract

Lens flare artifacts are undesired visual distortions caused by stray light, which can negatively impact the integrity and quality of an image. These artifacts pose a significant challenge in industrial applications like automotive and surveillance, where the quality and reliability of input images are crucial. Although there are techniques to prevent unwanted light from entering the camera, they are not always effective, requiring image post-processing methods.

In this work, a comprehensive review of image reconstruction algorithms based on deconvolution and stray light characterization through Point Spread Function (PSF) modeling is conducted. These approaches emphasized the significance of accurately modeling the stray light using the PSF specific to a camera system. However, necessary equipment to measure the PSF is unavailable for this work, and attempts to generate a synthetic PSF for evaluating the performance of a deconvolution algorithm resulted in unrealistic lens flare artifacts. Furthermore, literature studies primarily focused on static camera system setups like the ones found in microscopy or astronomy applications (telescope), indicating limited potential for PSF-based lens flare reduction in dynamic industrial settings.

On the other hand, artificial intelligence, particularly deep learning neural networks, have shown promising results in attenuating lens flare despite limited studies in this area. In this work, a synthetic flare dataset is generated, and an iterative training process that includes the evaluation of transfer learning is employed to develop FlareNet, the first compact and lightweight U-Net based model for lens flare reduction. FlareNet architecture, with a small parameter count of less than 150,000 parameters comprising convolutional layers, demonstrates improvement in image quality by reducing flare artifacts on synthetic test images. Furthermore, the model successfully reduces lens flare in real-life images, indicating its potential for achieving visually satisfactory results despite having less than 0.5% of the weights of the state-of-the-art neural architecture used for this same application. Additionally, a quantization-aware approach is applied to assess the impact of reducing the weight representation from float32 to int8, resulting in a 30% lighter model while considering the trade-off in accuracy.

This study serves as a proof-of-concept to understand the resource utilization and performance of implementing a model such as FlareNet, as a hardware-based digital circuit. To this end, the neural network is implemented in C++ using Vitis HLS, with each layer and necessary elements implemented and tested. Synthesis and validation are performed using the VITIS tool, and reports are analyzed while experimenting with HLS optimization directives. However, further work is necessary to optimize the overall design and explore more parallelism potential, making it feasible for deployment in real-time applications. Nevertheless, executing the model on a medium-end GPU demonstrates the possibility of meeting real-time requirements in terms of frames-per-second, but at the cost of higher power consumption, making it less suitable for low-power applications compared to an FPGA implementation.

Acknowledgments

I would like to express my gratitude to Professor Per Gunnar Kjeldsberg from NTNU and MSc Henrik Sundbeck from SONY Semiconductors EU for proposing the subject of my thesis and providing invaluable guidance and supervision throughout its development. I am especially grateful for their insights and feedback, which helped me to refine my research and achieve better results.

I would like to thank the European Master in Embedded Computing Systems (EMECS) program and its coordinators for providing me with the opportunity and support during my time as a student. This program has been an excellent platform for my academic and personal growth, and I am truly grateful for the experience.

Lastly, I would like to express my heartfelt gratitude to my family, without whom none of this would have been possible.

Index

1 Introduction

Photographs of scenes with a strong light source within or near to the optics system's field of view tend to present lens flare artifacts. These artifacts are caused by unwanted light behavior in the optical system, known as stray light. Although lens flare is just one type of stray light phenomenon, it is particularly problematic as it obstructs content in the image, reduces contrast and color diversity, and negatively impacts overall image quality and interpretability [1], as it can be seen in Figure 1. The present project focuses on attenuating lens flare artifacts, which is a challenging area of research in computer vision due to the diversity of lens flare patterns and the multiple factors that can produce them, such as light source characteristics, manufacturing imperfections, and lens wear and tear.

Although this type of artifacts may be trivial to casual photographers, it can severely affect the performance of systems in various domains, such as healthcare, industrial, security, and automotive. For instance, in automotive applications, high-quality images are crucial for distinguishing between pedestrians, vehicles, and traffic lights. Strong light sources, such as the sun or headlights, can generate lens flares, reducing contrast and detail in the image. As computer vision algorithms rely on high-quality input images, lens flare artifacts can corrupt and overwhelm the desired signal, increasing the likelihood of system errors. Flare attenuation can be achieved through three different approaches: modifying the optic system to reduce the amount of unwanted light entering the field of view, modifying the image capture process, or applying digital image post-processing.

This work focuses on post-processing approaches for flare attenuation, with a special emphasis on the potential of deep learning as an efficient solution that can be deployed as a specialized hardware accelerator compared to traditional image restoration. The study and evaluation of these approaches will provide valuable insights into the feasibility and effectiveness of using deep learning for lens flare attenuation.

Figure 1: Example of image without and with lens flare artifact.

1.1 Thesis Objectives

This study aims to explore the feasibility of developing a lens flare attenuation system suitable for real-time applications in industrial settings, with a specific focus on its integration into an embedded system. Therefore, the primary objective of this thesis is to construct a proof-of-concept solution utilizing image post-processing techniques. To achieve this overarching goal, the following specific objectives will be pursued:

- Conduct a comprehensive literature review on lens flare attenuation techniques, focusing on image post-processing approaches.
- Assess image post-processing methods suitable for deployment in embedded systems, specifically for lens flare attenuation in industrial applications like automotive systems.
- Select a viable algorithm from the evaluated approaches and analyze its performance to determine its suitability for implementation as a hardware accelerator.
- Design and implement the chosen solution, as a proof-of-concept, considering key constraints such as real-time application requirements, power consumption limitations, and resource utilization efficiency.
- Define a set of relevant test cases to evaluate the proposed solution, and assess its performance and robustness. Discuss the findings and provide recommendations for potential future enhancements.

1.2 Thesis Contributions

- Comprehensive literature review on existing image post-processing techniques for flare attenuation.
- Evaluation of flare attenuation using deconvolution approaches with Point Spread Function (PSF) modeling.
- Comprehensive literature review on the application of deep learning methods for flare attenuation.
- Generation of synthetic dataset comprising images with and without flare for training and evaluation.
- Exploration and fine-tuning of the first lens flare attenuation lightweight neural network model through iterative training and hyper-parameter optimization.
- Exploration of the potential of using transfer learning for flare attenuation.
- Discussion on the highlights and considerations to have during training a memory lightweight model for flare attenuation.
- Exploration of using quantization aware techniques to optimize the deep learning model for efficient inference.
- Evaluation of the performance and effectiveness of the trained model on real-life images affected by flare and analysis on how to improve the results.
- Implementation and analysis of the convolutional neural network layers in C++ programming language required to build the U-Net based architecture and synthesize it using VITIS HLS.
- Synthesis and optimization of neural network layers using HLS optimization directives, analyzing their impact on resource utilization and performance for future improvements and depoyment in FPGA.
- Functional verification of the Register Transfer Level (RTL) implementation through VITIS Co-Simulation.
- Deployment of the deep learning model on two devices with GPUs (RTX3060 and Jetson Nano Maxwell) to assess its performance on an embedded system.

It is worth mentioning that the entirety of the project was accomplished during the spring semester, without any prior projects or studies forming the basis of this assignment.

1.3 Thesis Organization

The thesis report is structured as follows:

Chapter 2 provides the theoretical foundation behind lens flare attenuation utilizing image post-processing techniques, specifically image reconstruction using deconvolution algorithms and deep learning. The chapter starts by elucidating the phenomenon behind lens flare artifacts and explaining how characterizing stray light can assist in mitigating this undesired effect. Furthermore, it introduces deep learning as a promising approach to address this issue, with a primary focus on convolutional neural networks (CNNs), while also discussing the crucial components necessary for its implementation. Then, it presents state of the art work on flare attenuation using deep learning and provides a brief introduction to common deep learning hardware accelerators such as GPUs and FPGAs. Finally, the chapter explores the role of High-Level Synthesis (HLS) in enabling the implementation of complex algorithms, including deep learning models, into hardware designs.

Chapter 3 begins by examining previous endeavors in assessing the effectiveness of deconvolution algorithms for flare attenuation. It draws conclusions on why a restoration approach based on characterizing stray light using the Point Spread Function (PSF) is not recommended for this project. Furthermore, the chapter introduces the implementation of the proposed deep learning solution, encompassing multiple facets. These include the generation of synthetic datasets, defining the structure of the neural network, the training process, and implementing the convolutional layers in C++ for High-Level Synthesis (HLS). Additionally, the chapter provides comprehensive details on the quantization procedure and the deployment of the solution on an embedded device with AI acceleration capabilities, specifically the Jetson Nano GPU.

Chapter 4 showcases the evaluation and results of the deep learning model. It provides comprehensive insights into the training outcomes, quantization effects, and performance during inference on both synthetic and real-life flared images. The chapter highlights significant observations made during the training process and conducts a thorough analysis of the HLS synthesis results with focus on the impact of optimization directives. Furthermore, it evaluates the performance of deploying the neural network solution on the resource-constrained Jetson Nano.

Chapter 5 concludes the thesis work, summarizing the main contributions, key findings, and their implications.

Chapter 6 offers recommendations for future work and suggests potential improvements based on the findings and analysis conducted throughout the study.

The **Appendix** includes a detailed description of the project's GitHub repository, outlining the important files and providing instructions on how to execute them. It also includes technical guidance on replicating the training procedure of the neural network and configuring the Jetson Nano to run a C++ inference application.

2 Theoretical Background

This chapter aims to provide a concise introduction to the theoretical concepts necessary for comprehending the subsequent design and implementation chapter.

2.1 Flare Attenuation through Deconvolution

2.1.1 Stray light

Stray light is an unwanted electromagnetic radiation that can adversely affect the quality of data captured by optical systems, such as cameras [2]. This phenomenon can be caused by a direct or indirect light source in front of the camera's field of view, as well as by light scattering on the optics surfaces due to dust and other imperfections, or by light reflecting on mechanical mounting surfaces within the optical system [3]. Stray light can be classified as either specular or scatter [4]. Specular stray light follows a physical model and has a deterministic behavior. An example of specular stray light are "ghost reflections", which occur when light from a source in the image field undergoes several unwanted reflections before reaching the image sensor array. In contrast, scatter light or flare has a more unpredictable behavior and is usually generated from light that gets scattered inside the optical system due to imperfections in the lens surfaces. An example of how lens flare is generated can be seen in Figure 2.

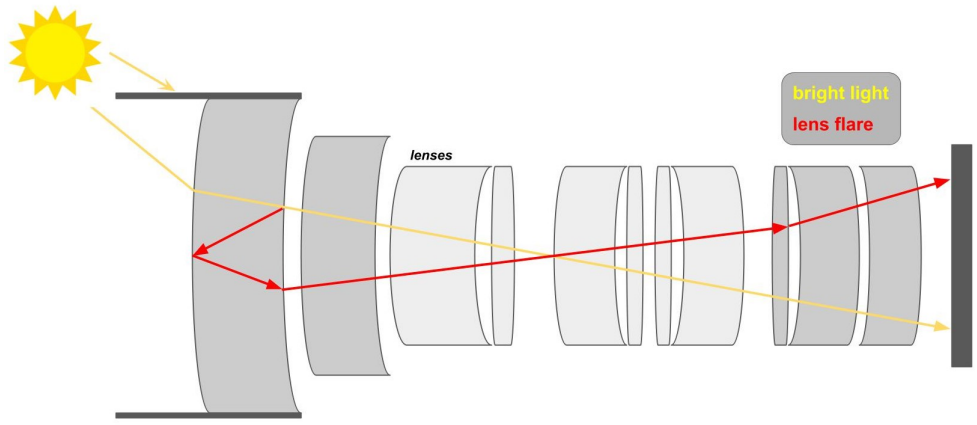

Figure 2: Example of how lens flare artifacts are generated in an optical system.

There are several mechanisms for mitigating stray light in general, and these can be broadly classified into three types of approaches: i) modifying the optical system [5], ii) modifying the procedure of capturing an image [3], and iii) applying image post-processing techniques [6]. The first approach involves physically modifying the optical system to prevent unwanted light from entering through the lenses. This can be achieved through the use of baffle lenses, which block stray light from entering the optical system. The second approach involves modifying the procedure of capturing an image in order to

minimize the impact of stray light. For instance, in [3], the authors use a high frequency occlusion mask between the scene and the camera. The resulting occluded scene along with the unoccluded one are shot with multiple exposure times. Using the occluded photos, the authors propose a method of estimating the stray light and then subtracted it from the unoccluded photo. However, this approach requires a considerable amount of images of a static scene before attempting the attenuation process. The third approach is the most challenging one due to the different types of stray light artifacts that could appear depending on the scene and the optical system. To reduce the complexity of the problem, image post-processing algorithms often focus on specific types of stray light, either specular or scatter, to attenuate them as accurately as possible. The present work focuses particularly on lens flare attenuation. Examples of the two different types of stray light artifacts that were discussed are illustrated in Figure 3.

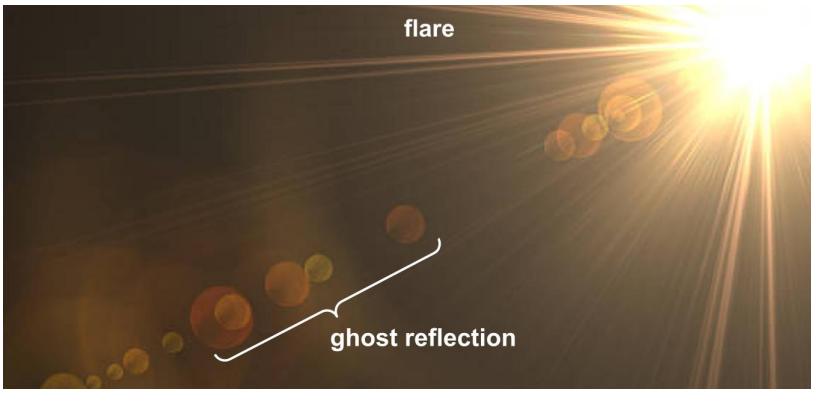

Figure 3: Example of types of stray light: specular (ghost reflection), or scatter (flare).

2.1.2 Modeling Stray light

In a theoretical scenario, a three-dimensional point within the camera's field of view should individually influence the level of color intensity in its corresponding two-dimensional pixel representation. However, in reality, light from neighboring pixels within and outside the optical system's field of view contaminate the rest of the image, especially when it comes to a strong light source. This behavior can be modeled mathematically by defining the intensity in a given pixel as the linear weighted combination of the intensities in all points in an underlying ideal image [7]. For this purpose, a two-dimensional convolution operation is defined.

Two-dimensional (2D) convolution is a common mathematical operation used in signal and image processing. It involves sliding a small 2D matrix or kernel, as seen in Figure 4, over an image represented as another larger 2D matrix, and computing the sum of element-wise products at each position. This operation can be used for tasks such as blurring, edge detection, and feature extraction in image processing applications.

Therefore, the intensities in the observed image are the result of a two-dimensional convolution operation between an ideal image, referring to an image without the effects of

lens flare artifacts, and a filter as seen in Equation (1). Where Obs(x,y) is the function of the observed image intensity, Img(x',y') is the ideal non contaminated image and PSF(x,y;x',y') is the Point Spread Function (PSF) of the entire system. From this mathematical representation, it is theoretically possible to retrieve the ideal image by means of applying the inverse operation of convolution, also known as deconvolution. A later section will further explain the deconvolution operation used for image restoration along with the most commonly used algorithms. However, the quality of the deconvolution results are highly dependant on how well the PSF has been modeled.

$$Obs(x,y) = \sum_{x'} \sum_{y'} PSF(x,y;x',y') \times Img(x',y') \tag{1}$$

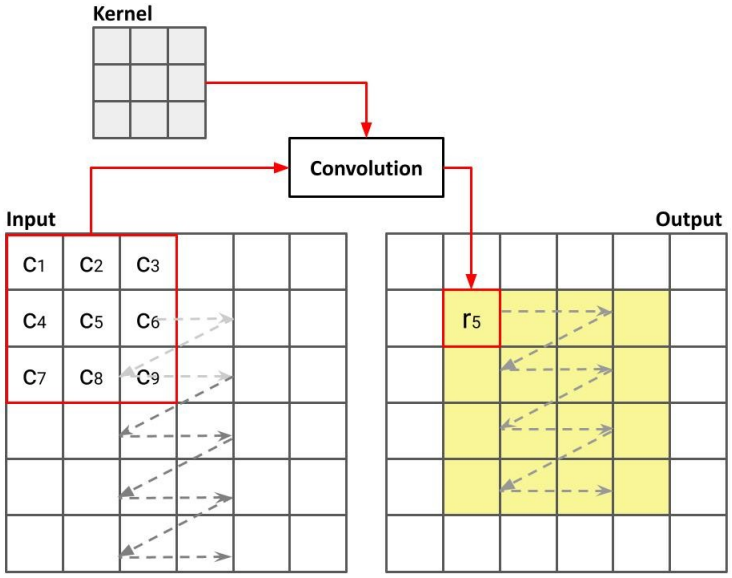

Figure 4: Example of sliding window through image.

2.1.3 Point Spread Function (PSF)

The PSF is the impulse response of an optical system, characterizing the system's response to an individual point of light source. In other words, it describes how much power is leaking between neighboring pixels according to their position. The PSF is usually represented as a 2D function that describes how the point of light spreads out over a certain distance from its original position. Therefore, the presented PSF of the entire system, represented by PSF(x,y;x',y'), is the collection of individual PSFs in every pixel of the image. The shape of the PSF depends on various factors, such as the quality of the optical system, the wavelength of the light, and the aperture size. As mentioned before, estimating the PSF of an optical system is a crucial step in stray light attenuation. According to [8], the process requires capturing several measurements of individual PSF in each point of the camera's field of view. This can be done by using a small light source in a black scene that resembles one pixel at maximum intensity levels and the amount

of light it spreads over its neighboring pixels. For this purpose, a small LED on a black surface is placed far enough from the camera so that it represents the size of a single pixel as seen in Figure 5. In order to avoid light reflections, it is recommended to conduct the experiment in a dark room. It is desirable to measure the PSF for every pixel of the image, but it is impractical due to the excessive amount of time it would take. Therefore, a series of PSF measurements are taken from different pixels and the remaining PSFs are estimated through interpolation techniques. Moreover, capturing measurements at various exposure times is crucial to obtain a precise approximation of the PSF and avoid pixel saturation.

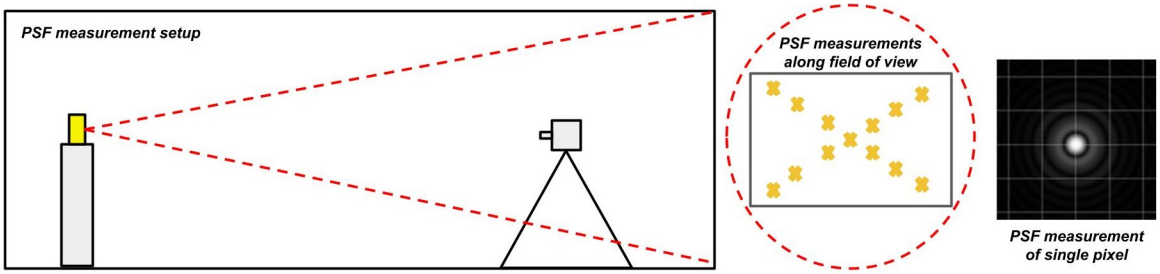

Figure 5: Experimental setup used to measure the PSF of a camera.

Rather than relying on traditional interpolation techniques to estimate the PSF for all remaining pixel points, a more accurate approach involves utilizing a parametric model that considers the behavior of stray light within an optical system. This advanced solution takes into account the specific characteristics and properties of the optical system, allowing for a more precise estimation of the PSF. Examples of this is [8], a model derived from laws of optical physics. Additionally, in order to estimate the parameters of the parametric model, an error function must be defined and optimized by linear solvers using individual PSF measurements as input. However, using a parametric model such as the one used in [8] has certain limitations. Firstly, it struggles with smaller local maxima and is unable to account for ghost reflections. Secondly, it does not consider light that enters the system from outside the field of view of the camera.

Additionally, it is worth noting that the PSF, as shown in Figure 6, is a tensor that depends on both the position of the light source and the neighboring pixels that are affected by it. As a result, there is a convolution kernel for each pixel, which leads to a complexity of $O(MxNxnxm)$ for the deconvolution process. Where M and N are the dimensions of the image, while n and m are the dimensions of the PSF kernel. This means that the computational cost of the deconvolution operation grows rapidly as the size of the image and the PSF kernel increase. For example, if we double the size of the image or the PSF kernel, the computational cost will increase by a factor of 8. This can make deconvolution computationally expensive and time-consuming. However, previous research has shown that using a shift-invariant representation of the original PSF can also achieve acceptable quality enhancement of the image while considerably reducing the complexity of the deconvolution operation [8]. The shift-invariant PSF is obtained by evaluating the shift-variant PSF at the center of the image.

It is important to note that a significant portion of the research on image restoration using PSF has primarily focused on setups involving still cameras or optical systems, where the scenes are highly controlled and static. For instance, applications in fields such as microscopy, astronomy (telescopes), and healthcare.

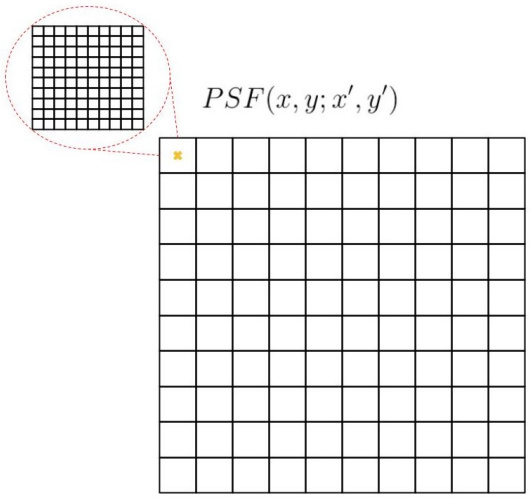

Figure 6: Shift-variant PSF kernel

2.1.4 Deconvolution

Deconvolution is a mathematical technique used to recover the original image from its degraded version through the use of a restoration filter which is designed based on the known properties of the degradation process, such as the PSF. In practice, this inverse filter is often unstable or amplifies noise, and thus, additional techniques are used to balance the trade-off between noise reduction and quality preservation.

Several algorithms have been developed and grouped based on their approach to achieve image restoration. One way is based on whether the algorithm is solved in an iterative or non-iterative way. Another way is based on the level of prior knowledge the algorithm has about the PSF kernel: blind, semi-blind, or non-blind. Two important algorithms are the Wiener and Richardson-Lucy deconvolution.

The Wiener filter is a non-iterative linear space-invariant filter known for fast computation, it takes into account the frequency spectrum of the original image, the PSF, and the noise in the degraded image. By using a statistical model, it estimates the optimal linear filter that minimizes the mean squared error. The filter can be implemented in the frequency and spatial domain using the Fourier Transform or Convolution respectively. Nonetheless, the Wiener filter is sensitive to noise and can produce negative results, leading to ringing effects in the restored image [9].

On the other hand, Richardson-Lucy is an iterative algorithm based on the Bayes theorem that also assumes that the observed degraded image is a convolution of the true image

and the PSF, corrupted with noise. In each iteration, an estimate of the ideal image is updated by convolving the current estimate with the PSF and then scaling it to match the degraded image. The scaled estimate is then used to update the current estimate in the next iteration as seen in Equation (2). Where I^{n+1} is the restored image in the next iteration, I is the current image which is being restored, B is the initial flared image, and the convolution operation is denoted by $*$. The algorithm stops iterating when the quality of the image estimate reaches a satisfactory level based on a stopping criterion or when a predetermined number of iterations is reached. Moreover, Richardson-Lucy has the advantage of producing non-negative results and being robust against variations in PSF estimation. However, the algorithm is computationally intensive and the amount of time required for convergence to a satisfactory result is not known before run-time. The quality of the restored image depends on the number of iterations and the stopping criterion used. If the stopping criterion is not chosen properly, the algorithm may introduce artifacts in the restored image [10].

$$I^{n+1} = I^n \cdot \left(PSF * \frac{B}{PSF * I^n} \right)^{\beta} \tag{2}$$

Blind deconvolution is a type of image restoration algorithm that aims to recover the original ideal image as well as the PSF through an iterative process. Unlike non-blind deconvolution, which assumes a known PSF, blind deconvolution does not require any knowledge of the PSF. Instead, it estimates both the original image and the PSF simultaneously, based on the degraded image alone. However, blind deconvolution is a challenging problem due to the fact that its solution is non-unique and depends on assumptions and constraints defined to regularize the solution. Blind deconvolution has shown promising results in deblurring astronomical imaging, however it is still an active area of research and further developments are needed to improve its accuracy and reliability in a wider range of applications.

2.2 Genetic Algorithms

Genetic Algorithms (GAs) are optimization algorithms that use adaptive heuristics inspired by natural selection and genetics [11]. They are widely used to solve complex search and optimization problems. In the context of this work, GAs will be employed to search for the optimal weights for a convolutional kernel that can model the behavior of lens flare artifacts on an image, as it has been used before in other applications that required optimization of kernel weights [12]. The GA follows a set of standard steps that are repeated through a defined amount of iterations, also known as generations, that simulate time effect during an evolution process. This process is depicted in Figure 7, and each step is explained below:

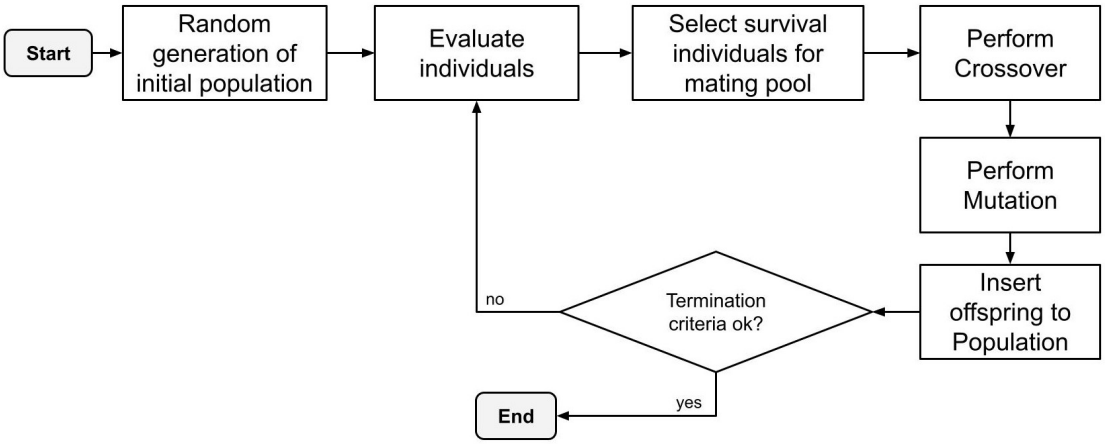

Figure 7: Diagram showing the main steps in a Genetic Algorithm.

- Initialization: a group of chromosomes is randomly generated, also known as population. Here, each chromosome represents a potential solution to the search problem. Each value inside the chromosome, also referred as gene, will be modified during the iteration process.

Figure 8: Structure of a chromosome, also known as individual.

- Evaluation: each chromosome in the population is evaluated by using an objective function, also known as fitness function, to rate how close a chromosome is to reach a solution. The fitness function must be carefully defined based on the search objectives, as it is the main way to guide the evolution of the chromosomes and measures the quality of the solution.

- Selection: mimics the idea of "survival of the fittest" in nature, as it selects only the chromosomes with highest fitness values. There are various selection approaches employed to choose parents for reproduction, such as: tournament and roulette.
- Reproduction: also known as crossover, involves combining genetic information from two chromosome parents to create offspring. This process usually involves exchanging or recombining segments of the genetic material between parents to generate new chromosomes. There are several approaches employed for mating, such as: one-point, one-point arithmetic, complete arithmetic, among others. For instance, one-point crossover involves randomly selecting a single point along the chromosome and exchanging the genetic material between the parents at that point as seen in Figure 9. On the other hand, the arithmetic crossovers combine the genetic material of both parents using a formula as seen in Figure 10.

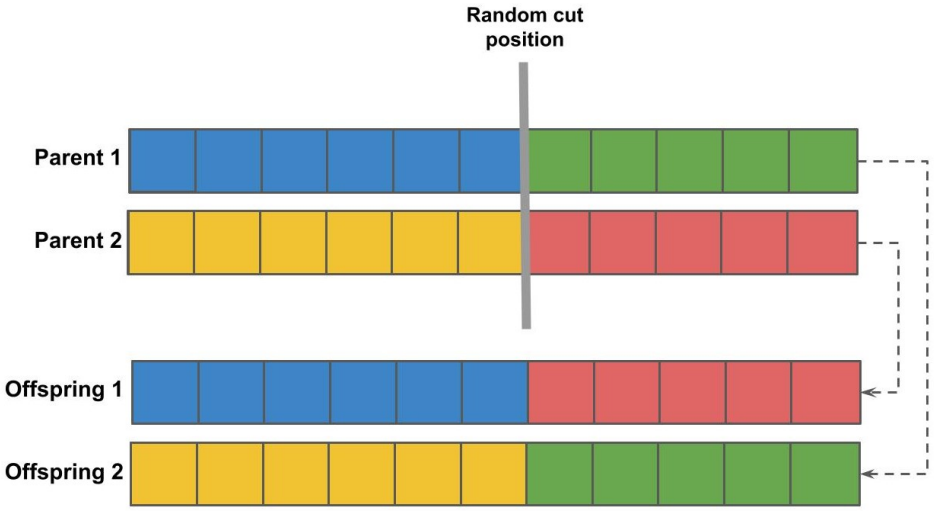

Figure 9: One-point crossover

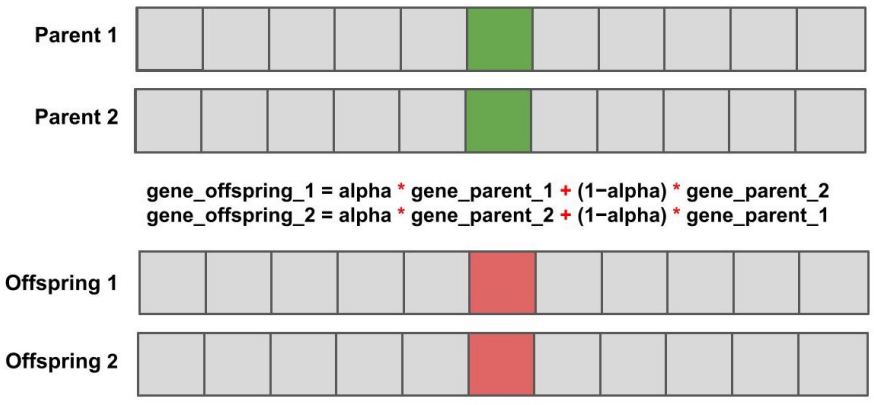

Figure 10: One-point arithmetic crossover.

- Mutation: involves random changes done to the offspring's genetic information. This step helps to maintain diversity in the population and allows for exploration of new areas of the solution space. For instance, two common types of mutation approaches are: all genes mutation or single gene mutation as shown in Figure 11. For the first type, all the genes can potentially mutate based on the "mutation probability" which will define if p_mut takes the value of 0 or 1. In case p_mut is 1, then the gene will be incremented by a ratio (beta) of its current value. However, if the mutation probability is 0, the gene will remain unchanged. Alternatively, in the single gene mutation approach, only one randomly selected gene in the offspring will undergo mutation based on the mutation probability. By increasing or decreasing the mutation probability the algorithm can control the degree of exploration in the solution space by applying drastic or subtle mutations in the genes.

Figure 11: Single gene mutation.

- Replacement: the chromosome parents with the lowest fitness values will be usually replaced by the new offsprings, before starting the next generation.
- Termination: can include reaching a maximum number of generations, finding a satisfactory solution, or running out of computational resources.

2.3 Deep Learning

Artificial Intelligence (AI) has gained significant popularity across various fields due to its ability to tackle complex problems. Machine learning (ML), a branch of AI, focuses on the development of algorithms that can adjust their response to input stimuli through an iterative learning process that requires exposure to data. These algorithms excel at recognizing patterns in data to make predictions without being explicitly programmed. In ML, there are two fundamental tasks: classification and regression. Classification involves assigning input data into predefined classes, where the output of a classification model is either a discrete class label or a probability distribution over the classes. On the other hand, regression models aim to predict a continuous numerical value or a set of values, commonly representing a quantity or a score. This type of model seeks to find a function that best fits the data by minimizing the difference between predicted and actual ground-truth values [13].

Deep learning (DL) is a sub-field of machine learning, and it draws inspiration from the structure of the human brain to develop artificial neural networks. These extensive networks can perform predictive analysis when trained with a substantial amount of data specific to the application. Deep learning excels at automatically learning and extracting relevant features from various types of raw data, such as images, text, or sound, without the need of feature engineering. Additionally, it has the ability to adapt and improve with the amount and diversity of available data, which will increase the robustness and accuracy of its predictions on real-life cases [14]

2.3.1 Neural Network - Perceptron

As the building block of artificial neural networks lies the "perceptron", illustrated in Figure 12 and represented by a mathematical equation based on various learning parameters like weights and biases [14] as seen in Equation (3). Here y represents the output of the perceptron, wi are the weights associated with the input variables xi, b is the bias term.

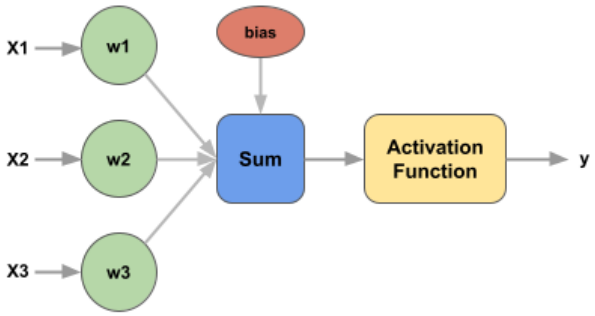

Figure 12: Perceptron model.

$$y = activation(\sum_{i=1}^{n} w_i \cdot x_i + b) \tag{3}$$

This equation shows how the perceptron takes multiple inputs and multiplies them by a corresponding weight. These weighted inputs are then accumulated along with a bias term. The resulting sum is passed through an activation function, which determines the non-linear output behaviour of the perceptron. There are several activation functions used in artificial netural networks and each has its own characteristics and application which depends on the specific task and the properties desired in the model's output [15]. The most common ones are explained below.

- Sigmoid: known as logistic function, maps the input to a value between 0 and 1. It is useful in models where the output needs to be a probability.
- Rectified Linear Unit (ReLU): returns the input value if it is positive or returns zero otherwise. It has gained popularity due to its simplicity and ability to mitigate the vanishing gradient problem by only allowing positive values to move forward in the network computation.
- Hyperbolic Tangent (Tanh): maps the input to a value between -1 and 1. It is often used in internal hidden layers of neural networks.
- Softmax: takes a vector of real numbers as input and outputs a probability distribution over multiple classes, with a sum equal to one. It is commonly used in multi-class classification tasks.

2.3.2 Convolutional Neural Network

Convolutional Neural Networks (CNNs) are among the most important deep learning architectures used for feature extraction from images and other multi-dimensional data. A CNN, presented in Figure 13, stacks multiple layers of filters with different learning parameters to extract and recognize increasingly complex features from the input data. The key component of a CNN is the convolutional layer. This layer is conformed by a set of filters with weights that are adjustable through back-propagation during the learning process. Each filter performs a mathematical operation called convolution, which requires the filter to slide across the input image while computing dot products between the filter weights and the corresponding input values. The result is a set of feature maps that highlight different patterns or recognize abstract features in an image, such as: texture, color, edges, among others [16]. CNNs typically consist of multiple convolutional layers followed by activation functions, typically ReLU, to introduce non-linearity into the network. In addition, pooling layers, such as max pooling or average pooling, are usually included between hidden layers which downsample the feature maps, reducing their spatial dimensions while learning to retain the most important information. Depending on the neural network application, the final layers can be fully connected layers or another convolutional layer with dimensions that match the output data dimensions. Overall, CNNs have revolutionized the field of computer vision and have become the go-to choice for many visual recognition tasks, including image classification, object detection, and semantic segmentation [17].

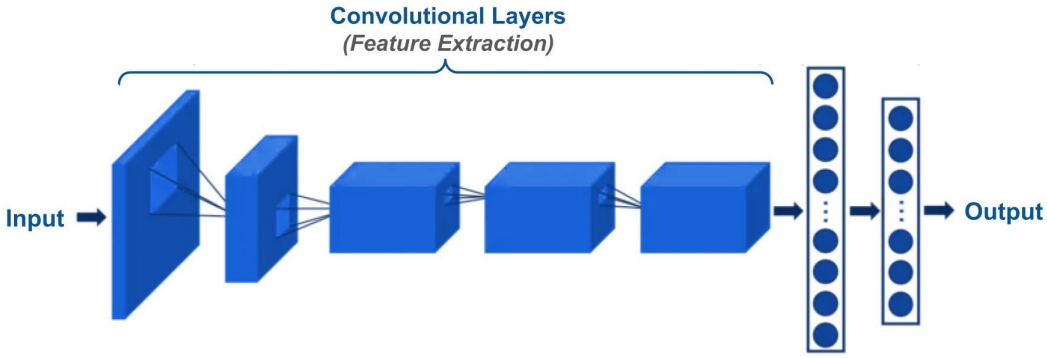

Figure 13: Convolutional Neural Network (CNN)

2.3.3 Training and Inference

During the training process of a neural network, a large dataset is used, typically consisting of input data paired with corresponding output ground-truth values. The ground-truth represents the desired prediction that the model should learn to reproduce when given a specific input. The neural network adjusts its internal parameters, which include the weights and biases of each perceptron, using a technique called backpropagation [18]. Backpropagation involves propagating errors backward through the network to compute gradients, which are then used to update the weights and biases of the neurons. This iterative process of forward propagation, loss calculation, backward propagation and weights updates continues until the model gradually improves its predictions and achieves higher accuracy on unseen data, as portrayed in Figure 14.

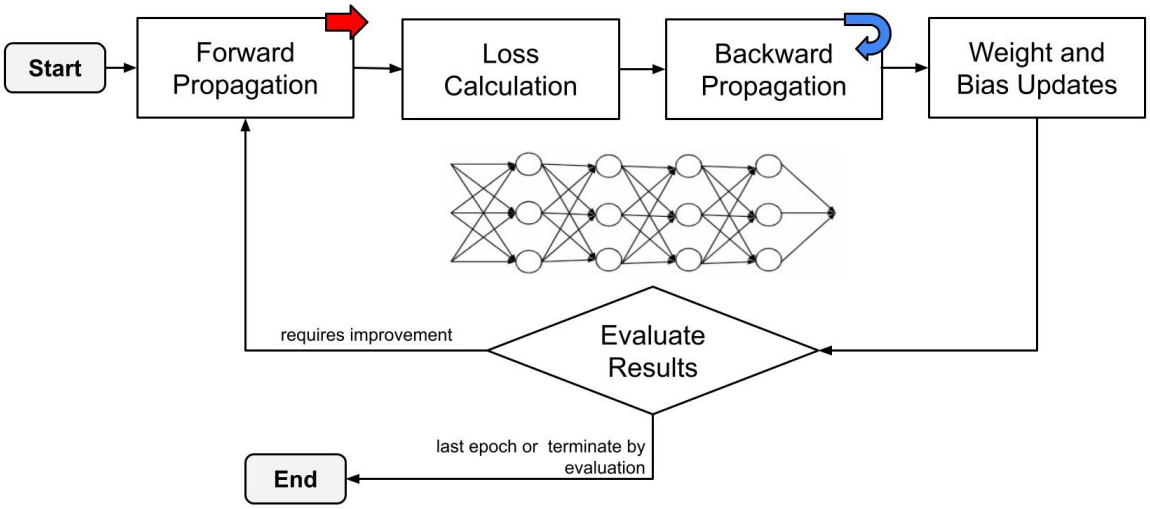

Figure 14: Deep learning training process.

The dataset is a crucial component in developing a deep learning solution as its quality and diversity greatly impact the results. A diverse and representative dataset enables the neural network to generalize its knowledge and make accurate predictions on real-world data. Typically, the dataset is divided into three subsets: training, validation, and testing.

During the training phase, the model learns from the training data to improve its performance. The validation data is used to assess whether the model is overfitting to the training data and if it can generalize well to new data. Overfitting is one major problem during training, and it happens when the model only learns how to predict on input data that is the same or very similar to the train dataset, performing poorly with unseen data. By monitoring the loss values of both the training and validation sets, it is possible to determine if the model is learning effectively without overfitting. If both the training and validation losses decrease in a similar ratio over iterations, assuming a balanced dataset, it can indicate that the model is not overfitting. On the other hand, the testing data provides an unbiased assessment of how well the final model generalizes to unseen data.

Inference in a neural network refers to the process of making predictions using a trained model with set weights. Data enters through the input layer in the same format as it did during training, and is propagated forward through the network with the activations and weights of the neurons/perceptrons influencing the flow of information. Each neuron in the network applies its activation function to the weighted sum of its inputs, producing an output that serves as input for the next neuron. This process is repeated for all neurons in each layer until the output layer is reached, where the final prediction is made.

2.3.4 Hyper-parameters

In deep learning, hyper-parameters are parameters that are defined before the training process begins and determine the behavior and performance of the model. Unlike the model internal parameters (weights and biases), which are learned during training, hyper-parameters are not updated based on the data but rather selected by the developer. The following are some of the most important and common hyper-parameters:

- Learning rate: controls the ratio at which the model parameters will be influenced by the error loss correction after back-propagation. A high learning rate may cause the model to converge quickly but risks missing the optimal solution, while a low learning rate may lead to slow convergence or getting stuck in sub-optimal solutions. A common starting point for the learning rate is around 0.01.
- Optimizers: play a crucial role in determining how the network learns from the data and converges to the optimal solution. Popular optimizers include: Stochastic Gradient Descent (SGD), Adaptive Moment Estimation (Adam), Nesterov Adam (Nadam), Root Mean Square Propagation (RMSProp), among others [19]. For example, Adam uses the historical gradients to compute adaptive learning rates for each parameter. It considers two moving average estimates: i) first moment estimate (mean of the gradients) and ii) second moment estimate (mean of the squared

gradients). The level of contribution that past gradients have on the estimates are controlled by using decay rates (beta1 and beta2). On the other hand, Nadam modifies the update procedure by incorporating Nesterov momentum, which allows the optimizer to anticipate the upcoming gradient descent and make more accurate adjustments to the parameters.

- Epochs: determines how many times the model will iterate over the entire training dataset. The choice of the number of epochs depends on factors such as the size of the dataset, the complexity of the task, and the convergence behavior of the model through the analysis of its performance with a validation dataset.

- Number of hidden layers: defines the depth of the neural network, which determines the complexity and capacity of the model. Increasing the architectural depth by adding more hidden layers can enable the model to learn more complex feature representations but risking the possibility of overfitting.

- Activation functions: introduces non-linearity to the model. Choosing the appropriate activation functions, specially for hidden layers, such as ReLU, sigmoid, or tanh, can impact the model's ability to capture complex relationships in the data [20].

- Batch size: refers to the number of training examples processed before the model's parameters are updated within an epoch. The number of times the parameters are updated is defined by the number of elements in the train dataset divided by the batch size. In addition, it affects the speed and stability of the training process [20]. Larger batch sizes can lead to faster training but require more memory capacity to store activations and gradients in the development environment, while smaller batch sizes can introduce more randomness into the training process, helping the model to explore a wider range of examples and potentially improve its generalization ability. However, smaller batches will require more parameter updates within an epoch, leading to increased convergence time.

- Regularization techniques: are used to prevent overfitting by introducing randomness during training. Dropout is one regularization technique commonly used to prevent overfitting and improve the models generalization ability [21]. It involves randomly "turning off" or setting to zero a portion of the neurons in a layer during each training epoch. This process encourages the network to learn more robust and generalized features since no single neuron can rely solely on the presence of other specific neurons.

In addition to these hyper-parameters, there are some others that are unique to a Convolutional Neural Networks (CNNs) an are mentioned below:

- Number and size of filters: determine the capacity of the network to learn spatial patterns at different scales. Increasing the number of filters can capture more complex features, but at the cost of computational resources.
- Stride: determines the step size at which the filters move across an input image or tensor. A stride larger than one reduces the spatial dimensions of the output feature maps, speeding up the inference process but sacrificing detailed spatial information.
- Padding: adds extra border pixels to the input image, allowing the convolutional filters to process pixels at the edges of the image. It prevents reduction in dimensions during convolutional operations.
- Pooling: down-sample the feature maps by evaluating local neighborhoods [22]. Common evaluations include calculating the maximum or average value between neighbors. The stride of the pooling operation determine the spatial reduction. A pooling operation with a stride of 2 will half the dimensions.
- Kernel initializer: technique used to initialize the weights of the kernels in a neural network. Different initialization methods can be used [23], two of the most common ones are: i) Xavier Initialization: It initializes the weights with random values drawn from a distribution that is scaled based on the number of input and output units and works best for activation functions such as sigmoid or tanh, ii) He Initialization: scales the weights based only on the number of input units, without considering the number of output units and is specifically designed for ReLU activation function.

2.3.5 Image Processing Metrics (MSE, MAE, SSMI)

In image processing, Mean Squared Error (MSE), Mean Absolute Error (MAE), and Structural Similarity Index (SSIM) are commonly used metrics to compare two images.

MSE measures the average squared difference between the original image and the processed image. It is calculated as the average of the squared differences between corresponding pixel values of the two images. MSE is a measure of the overall image quality, and it penalizes large deviations from the original image more than small deviations. In Equation (4), n is the number of data points, y_i is the true value of the i-th data point, and $\hat{y}_i$ is the predicted value of the i-th data point.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{4}$$

MAE, on the other hand, measures the average absolute difference between the original image and the processed image. It is calculated as the average of the absolute differences between corresponding pixel values of the two images. Unlike MSE, MAE is less sensitive to outliers and large deviations from the original image. In Equation (5), n is the number of data points, y_i is the true value of the i-th data point, and $\hat{y}_i$ is the predicted value of the i-th data point.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{5}$$

SSIM is a metric that takes into account the luminance, contrast, and structure of the images being compared. It measures the similarity between the two images on a scale from 0 to 1, where 1 indicates perfect similarity. SSIM is more perceptually meaningful than MSE or MAE because it models the sensitivity of the human visual system to changes in the image [24]. In Equation (6), x and y are the two images being compared, where μ_x and μ_y are the means, σ_x and σ_y are the standard deviations, σ_{xy} is the covariance, and c_1 and c_2 are constants to avoid division by zero.

$$SSIM(x, y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)} \tag{6}$$

2.3.6 Data Augmentation

Data augmentation is a common technique used to overcome the limited amount of available training data by increasing its diversity through the generation of new artificial images [25]. It is essential to ensure that these artificially generated images are consistent with real-world conditions, enabling the model to generalize well to unseen data [26]. For instance, taking several pictures of the same scene with slightly rotated angles in the horizontal plane, or varying light exposure levels through image attenuation, are examples of data augmentation techniques.

Figure 15: Examples of image augmentation through horizontal flipping.

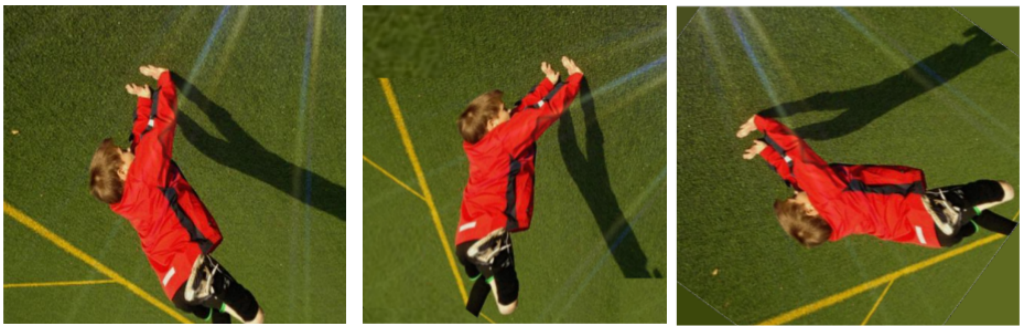

Figure 16: Examples of image augmentation through rotation.

Figure 17: Examples of image augmentation through light attenuation.

Some other more advanced data augmentation techniques used in [27] and [28] that will be used later in the implementation chapter are introduced and explained below:

- Remove DC background: removes the mean pixel intensity from the entire image.
- Affine transform: applies linear transformation that preserves the straightness of lines while allowing for translation, rotation, scaling, and skewing of objects. An affine transformation is represented by a matrix of size 3x3.
- Normalize and Random white balance: technique used to add variability to the color temperature of an image by randomly adjusting the color channels, previously normalized.
- Random blur and DC offset: adds variability to the image by applying a random amount of blur and adds a random offset to the pixel values of the image.
- Additive gaussian noise: common type of noise with gaussian distribution (zero mean and a known standard deviation) that is added to images to simulate the effects of random variations in the image pixel values.
- Random digital gain: randomly scales the pixel channel values of an image by a certain factor to simulate variations during image acquisition.

2.3.7 Transfer Learning

Transfer learning is a popular and powerful method in machine learning that allows the reuse of a pre-trained model for a different task within a similar domain. For instance, a pre-trained ResNet-50 neural network, which has been trained on a large dataset to classify images between 1000 different classes, can be re-used for a personalized application that requires classification between 38 completely new classes [29]. Compared to training a new model from scratch, transfer learning saves a significant amount of time, data, and hardware resources.

The key idea behind transfer learning is to use a pre-trained model that has already learned to extract many general feature maps that are common to information in a similar domain. This is especially useful in building convolutional-based models, where a pre-trained model can be used as the "backbone" to recognize general features such as texture, corners, shapes, among others. The output of these features can then be fed into a new stage in charge of learning how to use the given features to predict values close to the ground-truth ones. Figure 18 illustrates how a section of a pre-trained model can be exported and used in a new model.

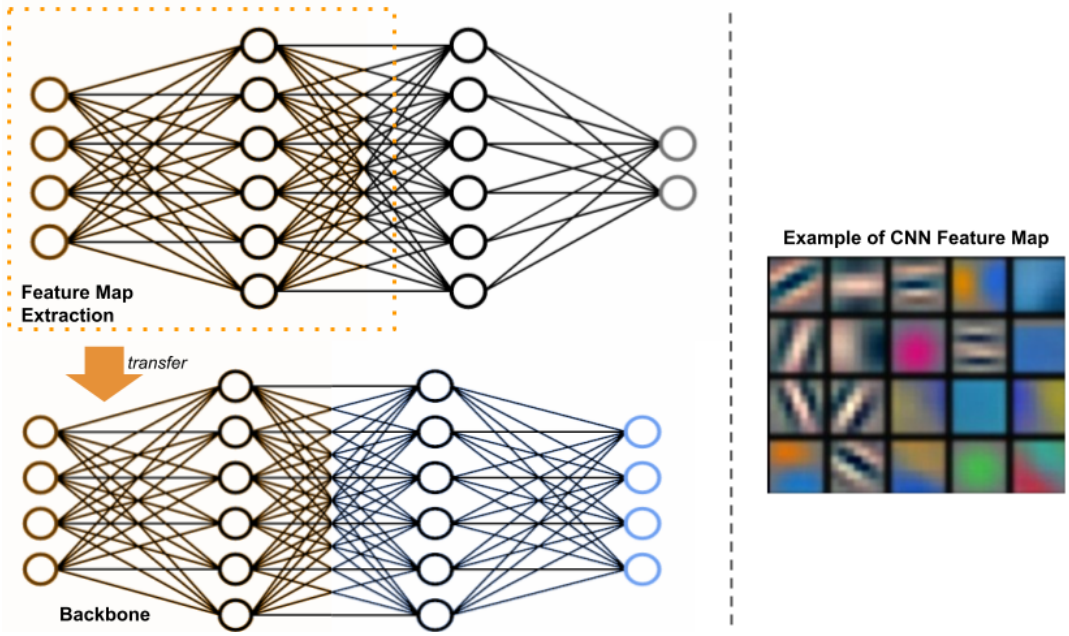

Figure 18: Visual explanation of transfer learning and example of CNN feature map.

2.3.8 Model Quantization

As introduced in Section 2.3.1, the basic building block of any neural network is the perceptron, which contains learning parameters that are adjusted during training. After the learning process, the model can be used for inference as a set of interconnected static parameters. The learning parameters are stored and called from memory every time a process needs to run inference on some input. The standard representation of the values is 32-bit floating-point, which allows for a high accuracy. However, a deep learning model can have millions of parameters, leading to a significant amount of memory usage that may not fit on certain hardware devices. For instance, a well known computer vision network, ResNet-50, contains around 26 million weights and 15 million activations, which requires around 150 MB of space in memory. So, a model that runs effectively on a development environment, such as a server or a desktop computer, might not be able to run effectively on a lower-end device with hardware constraints.

To address this issue, quantization can be used to optimize a model for a target device. Quantization reduces the required memory space and also has a positive impact on latency and power consumption while having little effect on model accuracy. Studies have shown that using 8-bit integer precision values for weights and activations does not significantly affect accuracy [30][31]. This is particularly important for low-end devices that lack floating-point functional units for arithmetic operations.

There are several post-training quantization approaches, as explained in [32], but the conservative approach is known as “dynamic range quantization”. This approach statically quantizes weights from 32-bit floating point to 8-bit precision integer, mapping the min-max values of the floating point tensor to the min-max values of the 8-bit integer. Sta-

tistical clipping techniques are also useful for avoiding outliers being considered as min or max values [33]. However, certain neural networks, such as MobileNet [34], require quantization-aware training to preserve accuracy as much as possible [35]. Quantization-aware training is a technique that involves simulating the effects of quantization during the training process. By introducing quantization effects, such as reducing the precision of numerical values to fewer bits, the model learns to handle the resulting loss of precision while still striving for the highest possible accuracy.

2.3.9 Deep Learning Frameworks

When it comes to implementing deep learning models, there are various software libraries, commonly known as frameworks, that offer high-level interfaces for tasks such as model design, data pre-processing, training, evaluation, and deployment in specific hardware target environments. Two of the most widely used frameworks are TensorFlow and PyTorch.

TensorFlow [36], developed and maintained by Google, is an open-source framework that provides a versatile architecture for building and training neural networks through its rich set of tools and APIs, making it suitable for a wide range of applications. On the other hand, PyTorch [37], developed and maintained by Facebook, is more popular among researchers. It is known for its dynamic computational graph feature, which allows for more flexibility in model design and debugging.

Both TensorFlow and PyTorch are extensively documented and supported by a community of developers. In addition, both are able to work with multiple programming languages, including Python and C++. Therefore, the choice of framework depends more on the specific requirements of the project and the developer's familiarity with the framework.

2.4 Deep Learning for Flare Attenuation

Deep learning has achieved remarkable success in various domains. However, applying it to flare attenuation tasks has been limited by the scarcity of real datasets available for training such models. Acquiring a large number of aligned images with and without flare from real scenarios is impractical due to the need for consistent photographic conditions (lighting, camera angle, exposure time, and other factors) between images that cannot be controlled in real-world scenarios. To overcome this challenge, several research works, such as [28], [38], and [39], have addressed the problem by leveraging high-quality synthetic datasets that incorporate a wide diversity of flare images merged with ground truth clean scenes.

In the study supported by Google Research, [28] works on a synthetic dataset generated by following the principles of optical physics to create flares that closely resemble real-world cases. The process involves mathematically modeling the scatter and reflective components generated by light diffraction in real lenses, resulting in a diffraction pattern, to then generate a vast amount and variety of synthetic images of flares. Additionally, the dataset includes real flares captured from different angles by rotating a camera around a light source in a dark room. This combination of synthetic and real flare samples enhances the diversity and realism of the dataset.

Similarly, [38] focuses on generating a dataset specifically for veiling glare, which is another type of artifact caused by a light source within or near the camera's field of view. However, unlike [28], they do not employ mathematical modeling of lens systems in their dataset generation process.

In another work, [39] proposed the first nighttime dataset for flare attenuation, considering both scattering and reflective components. Like the previous study, their dataset consists of synthetically generated samples without incorporating real flares.

To validate the use of synthetic datasets, all of these studies trained deep neural networks and evaluated the performance of different architectures. The work done in [28] found that the U-Net architecture [40] produced the best results for flare attenuation. During training, they employed two types of loss calculations: image loss and flare loss using MSE and MAE metrics respectively. The former encourages the predicted image to closely resemble the ground truth, while the latter encourages the predicted flare to resemble the ground truth to avoid introducing artifacts in the predictions. The training process involved approximately 60 epochs on a dataset of 20,000 samples, using the Adam optimizer with a fixed learning rate of 0.0001.

To measure the quality of the reconstructed images, Structural Similarity Index (SSIM) was used as a metric, comparing the predictions to a baseline defined by the input image (with flare) and ground-truth (without flare) from the synthetic dataset. They achieved a Structural Similarity Index (SSIM) of 0.994, surpassing the baseline SSIM of 0.843, indicating that the predicted image closely resembles the input image. By specifically addressing and mitigating the flare artifacts, the model effectively preserves the overall

similarity between the input and predicted images. In addition, they also conducted an evaluation on a real dataset visually showing the effectiveness of the model at attenuating lens flare [28].

2.4.1 U-Net

U-Net is a state-of-the-art deep learning auto-encoder architecture used for image segmentation [40]. An auto-encoder is a special type of neural network that copies its input value to the output by learning how to compress the input data while minimizing reconstruction error. Originally developed for biomedical image segmentation, U-Net gained popularity in different domains due to its combination of speed and precision. The neural network model shown in Figure 19 integrates two clear sections based on convolutional layers: encoder and decoder.

The encoder follows a typical contraction path where in each block the input image is passed through two 3x3 kernel size convolutions, each followed by a rectified linear unit (ReLU). In addition, the spatial information is reduced through a 2D max-pooling operation with stride 2, which is represented in Figure [19] by the red arrows pointing down. This process generates a bottle neck from which the model is able to increase its feature map knowledge. The feature channel dimension doubles in each block, with a filter size increasing from 64 to 512 across the blocks. The encoding process is repeated three times before reaching the bottom of the network, where two more convolutional layers are applied without max-pooling. It is a common practice to utilize existing CNN architectures like ResNet-101 [41], VGG16 [42], or MobileNet [34] as a backbone for the encoder section. These architectures come with the advantage of pre-trained weights on datasets such as ImageNet [43], allowing for transfer learning and achieving high accuracy. However, despite their accuracy, these encoders are complex with multiple layers and harder to work with for inference on edge devices with limited resources [44]. For instance, when employing the MobileNet architecture for transfer learning, the model is divided into several "blocks," each comprising a minimum of five layers. These layers typically include at least two normal convolutions, two depthwise convolutions, batch normalization, and concatenation layers to merge information from previous layers [34].

The decoder section is an extensive pathway that sequentially takes as input the concatenation between the up-convolutional features and the skip-connection as high-resolution features from the contracting path. Every block in the decoder consists of a transposed convolution for up-sampling the feature map followed by a 2D convolution that halves the dimension of the feature channels. In addition, the transposed convolution is concatenated with the corresponding cropped feature map from the encoder shown as the gray arrows connecting the decoder and encoder sections in Figure 19, to add original information from the contracting path and avoid loss of information. Finally, each block also includes two 3x3 convolutions with ReLU activation's functions and the process is repeated until reaching the output layer, where a 1x1 convolution with an output filter size of three is applied to map the feature channel information to the three output classes.

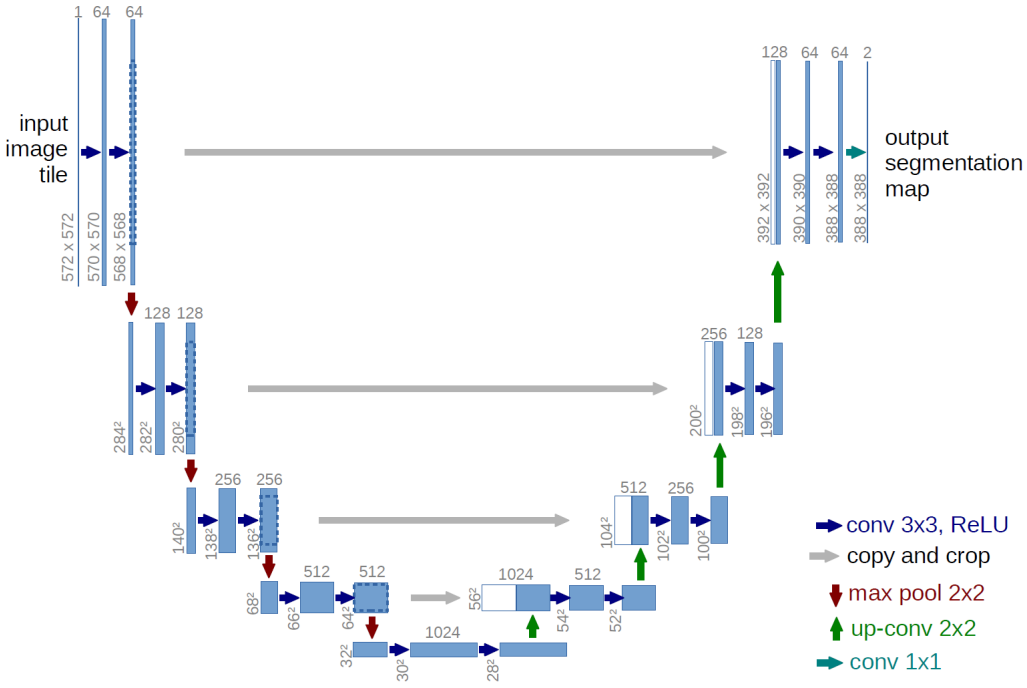

Figure 19: U-Net architecture [40].

2.4.2 Compact U-Net architectures

Despite the impressive performance of U-Net architectures, their pursuit of increased accuracy often leads to complex designs with millions of trainable parameters. This complexity demands extended training times and substantial computational resources, making them impractical for low-power devices with limited memory. To enable U-Net inference on edge devices, it is crucial to have models with low complexity, a small number of trainable parameters, while maintaining accuracy in the results [44].

Researchers have made efforts to reduce the complexity of U-Net models. For instance, the Squeeze U-Net model proposed in [45] achieves similar accuracy to the original U-Net while having only 2.59 million trainable parameters compared to the original model [40] with 30 million. Squeeze U-Net achieves this reduction by using fewer and simpler layers, primarily employing 2D convolutional layers, 2D transposed layers, and concatenation layers with ReLU activation functions. However, even with this reduction, Squeeze U-Net is still relatively large for edge device inference.

Another approach to reducing complexity is presented in [46] and [44], where the authors propose two U-Net-like architectures called C-UNet and C-UNet++. These models reduce complexity by removing convolutional stages compared to the original architecture. Additionally, they utilize separable depthwise convolutional layers, which significantly reduce the number of parameters while maintaining the filtering capability. The depth of the filters is also reduced, with the largest depth being 32, compared to the original U-Net's layers with 128, 256, 512, and 1024 depth filters. The authors find a good balance

between size and accuracy, with the C-UNet model having 51,113 parameters and the C-UNet++ model having as few as 9,129 parameters with only 4 types of convolutional layers (2D convolution, 2D depthwise separable convolution, 2D transpose convolution and 2D max-pooling), which can be seen in Figure 20.

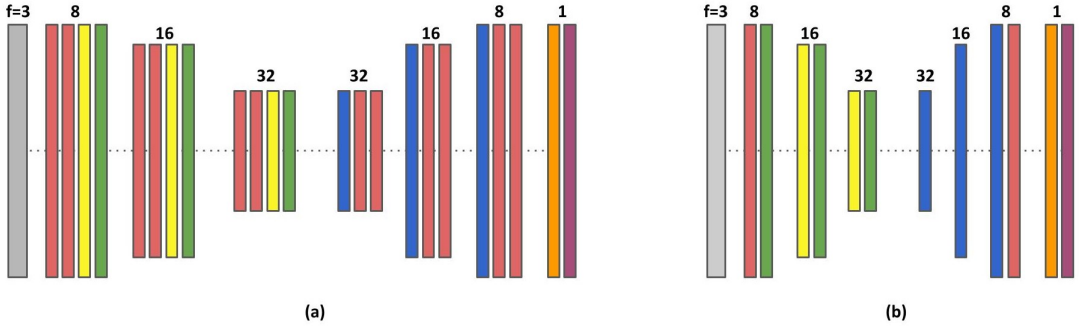

Figure 20: Compact U-Net based architectures proposed by [44]: (a) C-UNet and (b) C-UNet++. red: conv3x3 + ReLU, yellow: depthwise separable conv3x3 + ReLU, green: 2x2 max pool, orange: conv1x1 sigmoid, blue: 2x2 transpose convolution, lila: output

It is important to mention that these neural networks are specifically developed for semantic segmentation of satellite images with smaller sizes and less complexity in the training process, as it is a classification task rather than a regression one. For example, the authors mention that skip connections can be eliminated from the models as they keep high frequency information and this concrete application does not have much information in that domain. Later during the implementation chapter, it will become evident that skip connections are quite important for flare attenuation purposes, evidencing that every application requires a different architecture. Nevertheless, these studies present valuable insights on how to approach a reduction in model complexity for the U-Net architecture and will serve as a base to train several compact U-Net based models for flare attenuation.

In summary, U-Net architectures achieve their success through the combination of convolutional layers in the encoder and decoder sections, as well as the use of skip connections to transfer high-resolution features. The upcoming sections will delve into the key convolutional layers required to build a compact U-Net architecture.

2.4.3 2D Convolutional Layer

The convolutional layer is a fundamental block in computing vision algorithms, used for tasks such as feature extraction and image filtering. It involves applying a filter to an input image, to produce a feature map [47]. The filter is a four-dimensional tensor with dimensions that depend on the kernel size, the input depth (referred as m), and output depth (referred as n) as seen in Figure 21.

The convolution process involves sliding each of these n filters across the input image. Each section of the image that interleaves with the filter is called a window, which is of the

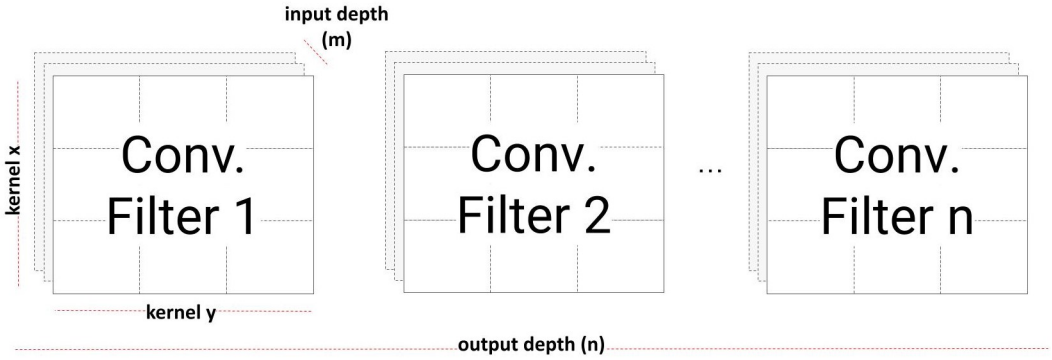

Figure 21: Dimensions of 2D convolutional filter.

same dimension as the filter kernel size in the x, y and depth axis. The entire dimension of both the filter and window goes through an element wise multiply and accumulate operation (MAC). The resulting values from each of the input depth channels (denoted by m) are then summed to produce a single output value for each position in the output feature map. This process is shown in Figure 22 and is repeated for each of the n filters before the window slides into the next set of image pixels, resulting in n output values per pixel, where n is the number of output channels for the layer. The window will continue to slide across the entire input tensor while performing the previous operation with all the filters, thus creating the entire output feature map one pixel (with depth of n channels) at a time. The weights of the kernel are learned during training, allowing the convolution operation to adapt to the specific task at hand. Common 2D kernel sizes include 2x2, 3x3, 5x5, and 7x7.

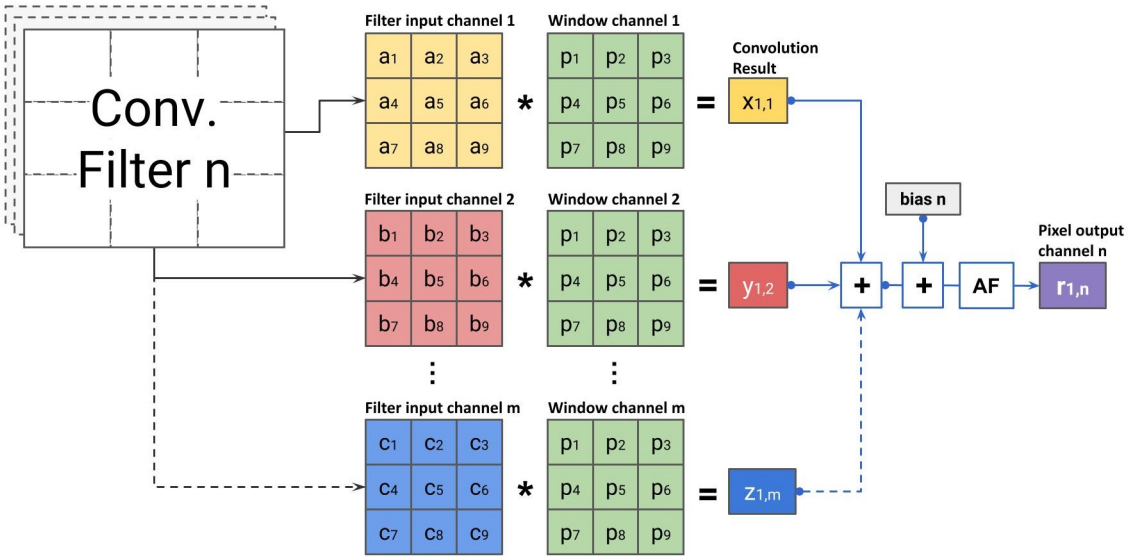

Figure 22: Example of 2D convolution operation for one filter.

2.4.4 Depth-wise Separable 2D Convolutional Layer

Depth-wise separable convolution is a technique used to reduce the number of parameters required in a typical 2D convolution, resulting in a memory-efficient layer that avoids overfitting during training. This is achieved by separating the filter into their spatial and depth dimensions, namely depth-wise and point-wise filters [34] as seen in Figure 23. Similar to the typical 2D convolution layer, the filters will slide across the image operating over the respective convolutional window, outputting one pixel with its respective output depth at a time as it is done in the traditional convolution.

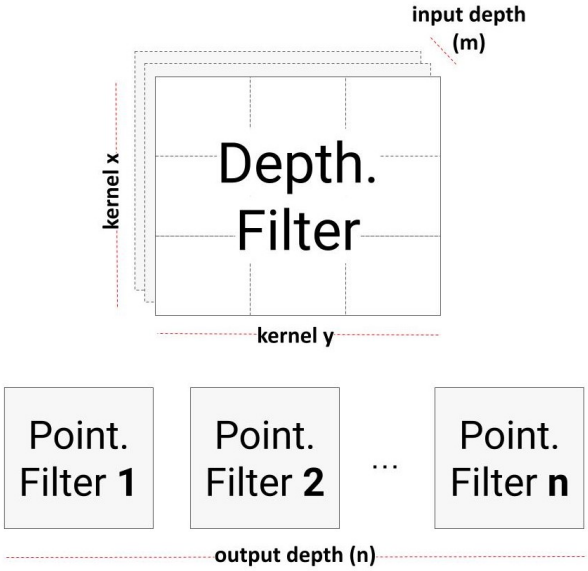

Figure 23: Dimensions of depth-wise and point-wise filters.

This process is executed as follows: in the depth-wise convolution, depicted in Figure 24, each input channel of the filter and its corresponding window undergoes an individual MAC operation. The results are then stacked together to form a final vector with the same size as the input channel (represented by m). Next, in the point-wise convolution, as illustrated in Figure 25, the depth-wise vector undergoes another MAC operation with the point-wise filter weights. This operation is repeated for each output channel (represented by n), generating the final convolution output for each pixel with a depth of n.

Similar to traditional convolutions, in this algorithm, the window continues to slide across the entire input tensor, performing the same operation with all the filters. This process continues until the entire output feature map is generated, pixel by pixel.

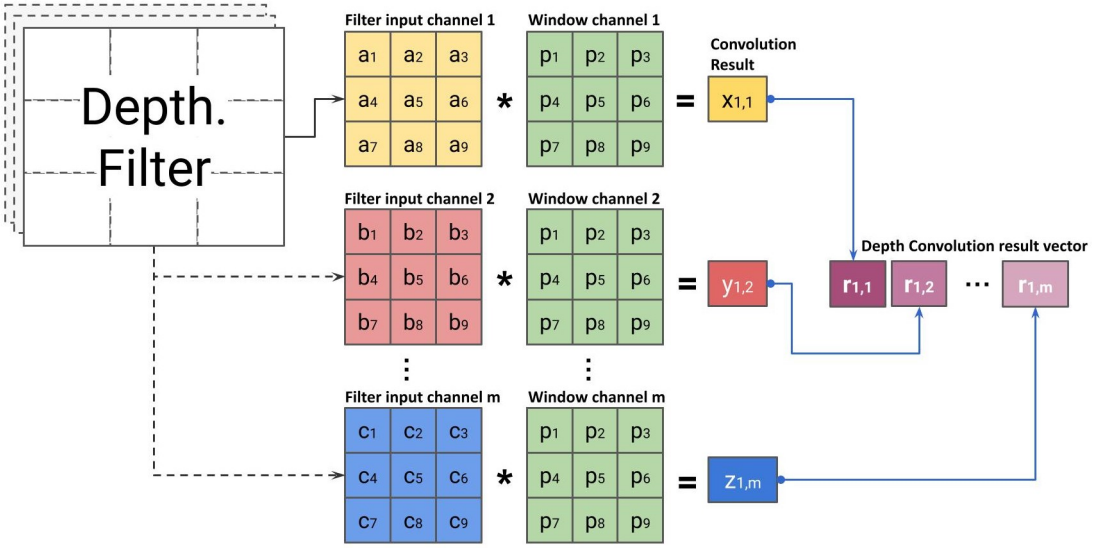

Figure 24: Example of depth-wise convolution step for one filter.

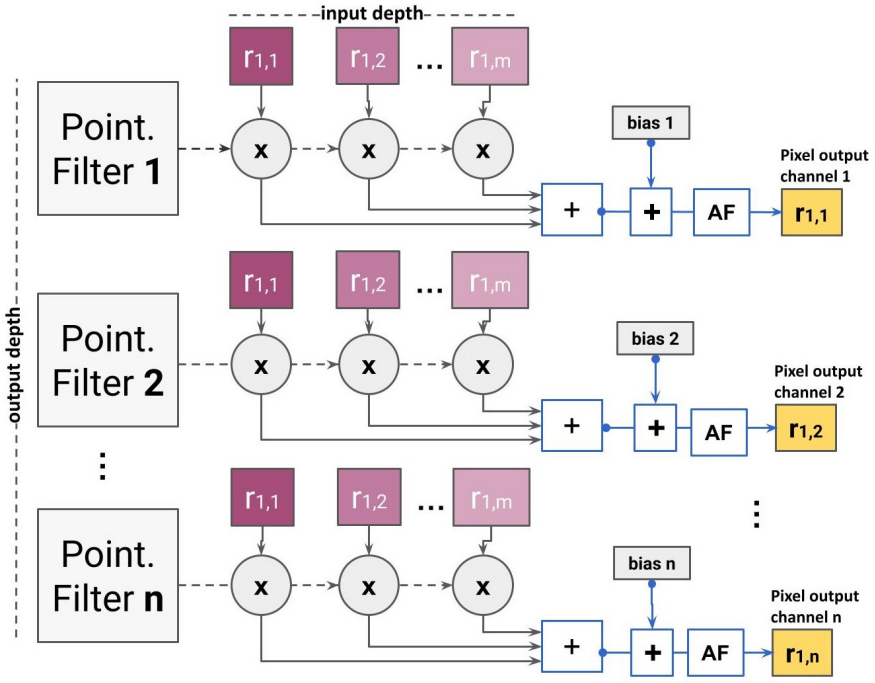

Figure 25: Example of point-wise convolution step for n filters.

It is worth noting that depth-wise separable convolution can significantly reduce the number of parameters required, compared to a normal 2D convolution. For example, a 3x3 kernel with input and output dimensions of 3 would require 81 parameters in a normal 2D convolution (3x3x3x3), while the depth-wise 2D convolution only needs 27 parameters (3x3x3) for the spatial depth-wise step and an additional 9 parameters (3x1x3) for the depth-wise step.

2.4.5 Transposed 2D Convolutional Layer

The transpose convolutional layer applies a set of filters to an input to produce an output of a larger spatial dimension [48]. The filter weights are learned during the training process, and can be used to perform tasks such as image segmentation or image reconstruction. However, it is important to note that transpose convolutional layers can lead to artifacts in the output image, such as checkerboard patterns or blurring, if not used carefully. To address this issue, several techniques have been proposed, such as stride-2 convolutions. The transpose convolution exhibits a similar iterative behavior to the preceding convolutional layers. However, instead of operating on a convolutional window, this algorithm processes a single pixel along with all its channels, as illustrated in Figure 27. Each transpose filter is element-wise multiplied with the corresponding pixel channel values, resulting in m matrices. These matrices are then summed, along with the corresponding filter bias, to generate an output matrix that has increased dimensions compared to the input pixel. This process is repeated for each transpose filter, and the resulting matrices are stacked together, resulting in a matrix with output depth of n.

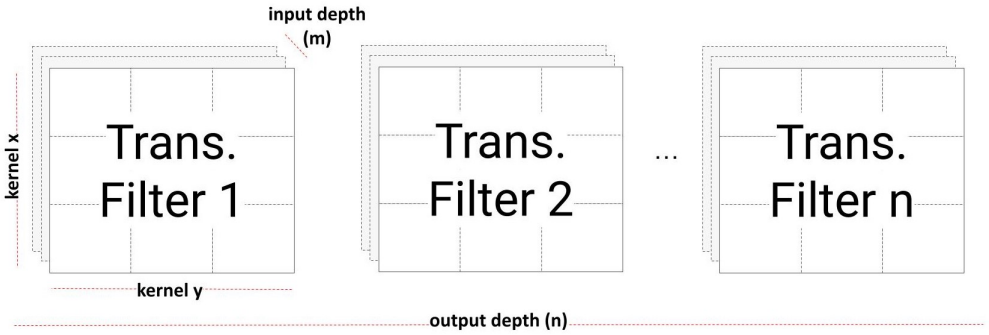

Figure 26: Dimension of transpose filter.

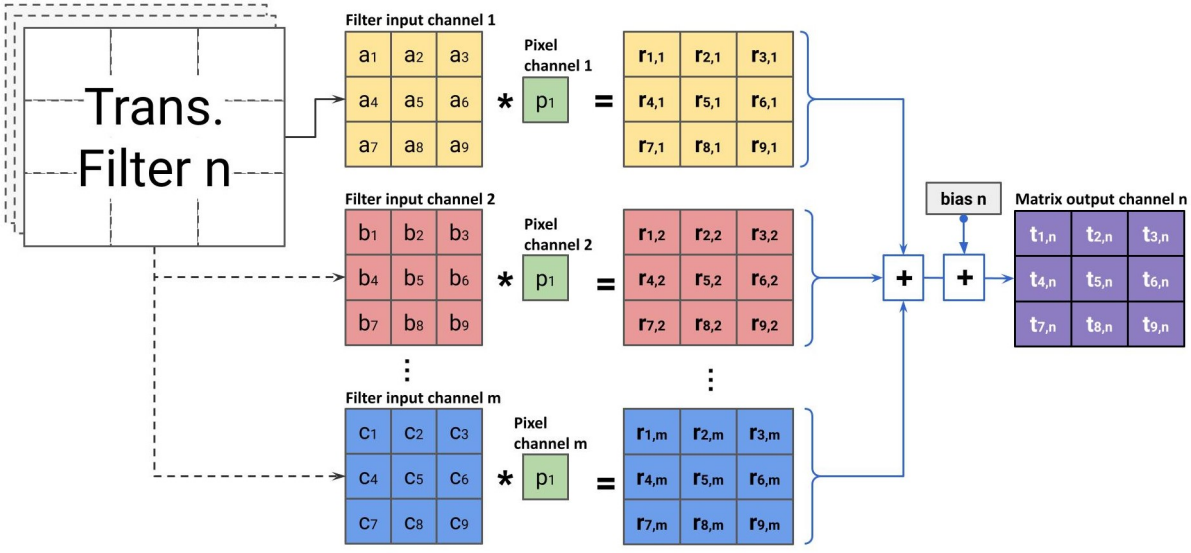

Figure 27: Example of 2D transpose convolution for one filter.

However, certain considerations need to be taken into account depending on the number of strides and the size of the filter kernel. If the kernel size is larger than the number of strides, the convolutional results from each filter n become temporary results as they will overlap with the next matrix convolution results. This can be seen in Figure 28, where a stride of 2 and a kernel size of 3x3 result in temporary matrix sections that intersect with each other. These intersecting sections, indicated by the gray-colored area in the output matrix, will have their values added together as it is shown. On the other hand, if the kernel size is equal to the stride, as shown in Figure 29, there is no intersection between the convolution results, and there is no need to consider temporary results since they are already final.

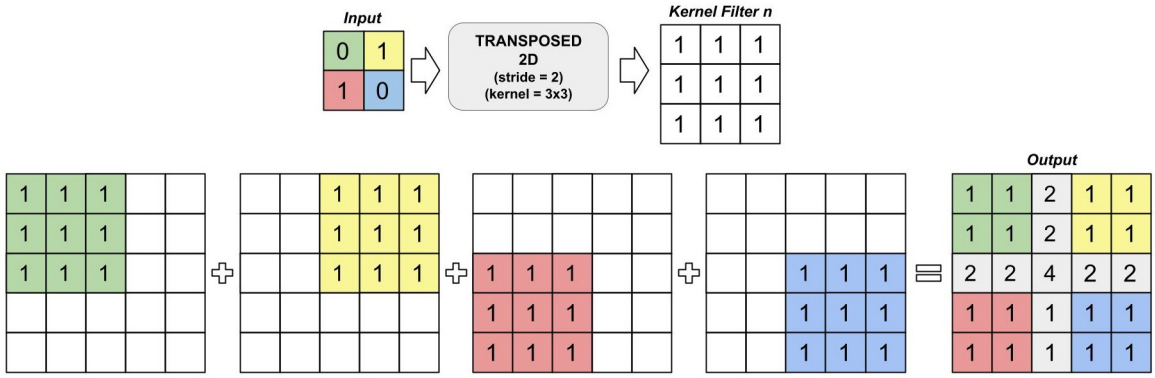

Figure 28: 2D transpose convolution using a kernel of size 3x3 and a stride of 2.

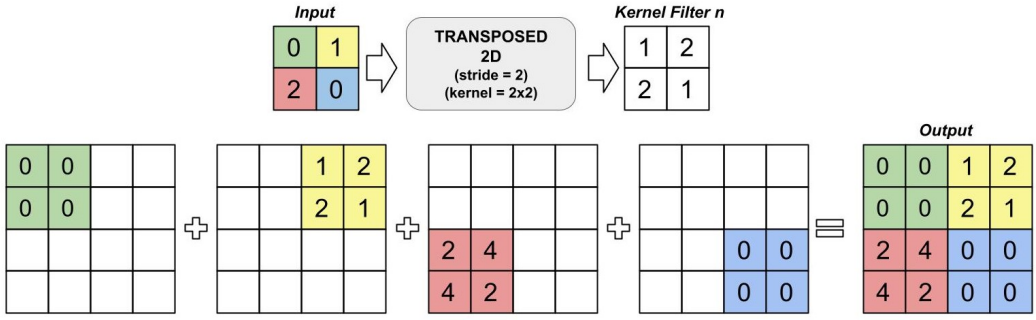

Figure 29: 2D transpose convolution using a kernel of size 2x2 and a stride of 2.

2.4.6 2D Max-pooling Layer

Max-pooling is a technique commonly used in deep learning for down-sampling data between layers. In the context of image processing and computer vision, down-sampling is a common technique used to reduce the spatial resolution of an image or feature map. Max-pooling works by sliding a window over the n-dimensional vector input data and selecting the maximum value within that window as the output of the pooling operation as seen in Figure 30. This technique can help reduce the size of the data and provide a way to extract the most important features from the input. Additionally, max-pooling can also help prevent overfitting by reducing the amount of noise in the data and making the network more robust to data input variations. However, it is important to note that there are some limitations to max-pooling, such as the possibility of losing important features during the down-sampling process. Therefore, other pooling techniques such as average pooling have also been proposed.

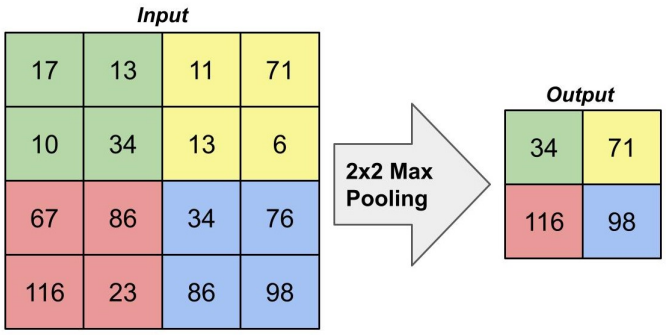

Figure 30: Example of max-pooling operation.

2.4.7 Adding Layer

The main purpose of this layer is to merge skip-connections, which consist of output values from non-sequential layers. Typically, one of these layers originates from the encoder, while the other comes from the decoder section in an auto-encoder structure. By incorporating inputs from earlier layers into later layers through skip connections, the network effectively preserves valuable information that might have been lost otherwise during backpropagation, particularly in deeper layers.

There are various approaches to merging skip connection information, depending on the specific application and network architecture. For instance, the original U-Net model adopts a concatenation layer, initially introduced in the Inception neural network [49]. However, this operation doubles the filter dimension of the resulting tensor, as depicted in Figure 19, thereby increasing the model's complexity. In contrast, the ResNet neural network [41] introduced an addition layer, which performs element-wise addition between two tensors of the same shape, resulting in a single tensor with unchanged dimensions.

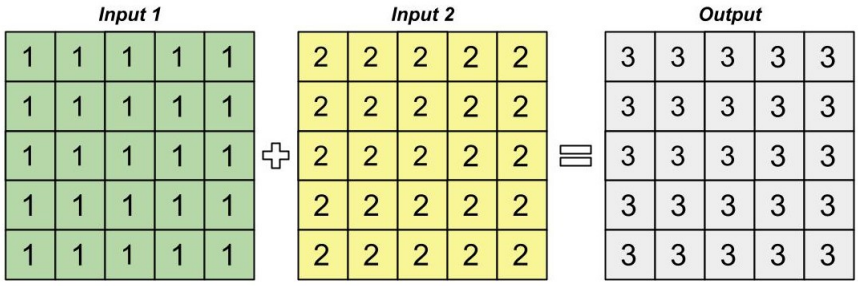

Figure 31: Example of adding operation between two feature maps.

2.4.8 Batch Normalization Layer

Batch normalization is a widely-used technique for normalizing data in neural networks. This layer does not change the dimensions of the input, but instead applies a transformation that centers the output around zero and normalizes it to have unit variance. This is crucial in some cases in order to ensure that the network is able to learn effectively by reducing the likelihood of vanishing or exploding gradients, which can hinder the training process. By normalizing the output, batch normalization helps to ensure that the network is able to learn more quickly and accurately, leading to better overall performance.

2.4.9 Dropout Layer

The dropout technique is a powerful regularization method that is commonly used in deep learning models to prevent overfitting. It works by randomly and temporarily disabling a percentage of neurons in a layer during the training process [21]. This means that these neurons are ignored during the forward propagation step and no updates are made to their weights during the backward propagation step. By randomly dropping out neurons, the model becomes less sensitive to the specific weights of individual neurons and allows the model to generalize better to new, unseen data. It is worth mentioning that dropout is only applied during training and during inference, all neurons are active and used to make predictions.

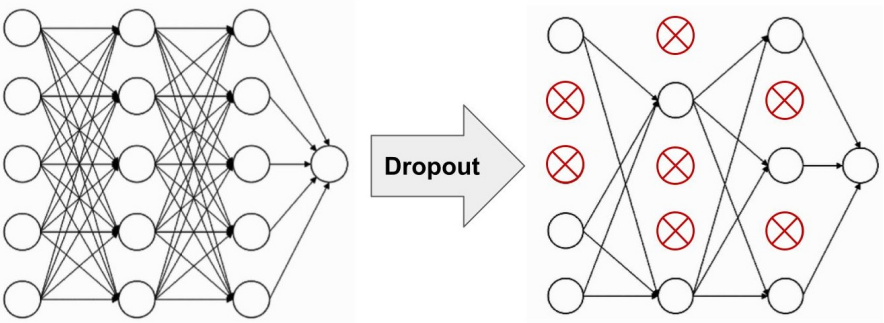

Figure 32: Example of Dropuout regularization technique.

2.5 Hardware Accelerators for Inference - GPU

The Graphics Processing Unit (GPU) has emerged as a crucial computing component due to its ability to perform massive parallel processing. Originally developed for accelerating 3D graphics rendering in video games, GPUs have since evolved into more flexible and programmable devices (known as GPGPUs [50]), finding applications in a wide range of fields, including deep learning. Although GPUs excel at parallel arithmetic operations, their purpose is more narrow compared to Central Processing Units (CPUs). Therefore, in a computing system, CPUs and GPUs usually work together, with CPUs in charge of orchestration-related tasks, such as the operating system, while the GPU handles the heavy arithmetic computations.

GPUs feature a Single Instruction Multiple Data (SIMD) architecture that enables effective data parallelism across multiple processors with a single instruction. GPUs are particularly efficient at matrix arithmetic operations in floating-point format, which is why the number of "Floating Point Operations per Seconds (FLOPS)" is a key performance indicator for GPUs.

NVIDIA is a global leader in GPU manufacturing and introduced the Compute Unified Device Architecture (CUDA) as a parallel computing platform to diversify GPU use. This platform allows developers to optimize their software for NVIDIA GPUs [51]. Developers can choose how to run their programs in terms of the number of cores and clock frequency in the CPU and GPU, considering the trade-off between throughput and power consumption. NVIDIA also produces Jetson modules, a series of GPUs specifically designed for embedded AI applications such as robots, drones, autonomous cars, sensor arrays, and computer vision systems that require AI capabilities with low to medium power consumption. The Jetson Nano [52] is one of these modules that delivers around 472 GFLOPS while using only 5 to 10 watts of power depending on its configuration. In addition to the GPU, the Nano includes Quad-core ARM Cortex-A57 and 4 GB of memory. It is important to mention the level of flexibility a developer can have to decide how to run its program in terms of number of cores and clock frequency in the CPU and GPU, which implies a design trade-off between throughput and power consumption.

Figure 33: Jetson Nano [52].

2.6 Hardware Accelerators for Inference - FPGA

Field-Programmable Gate Arrays (FPGAs) are integrated circuits that allows for the development of custom logic for rapid prototyping of a digital design solution. In today's technological landscape, FPGAs have gained popularity for developing customized accelerators, especially in complex applications like deep learning inference [53] [54].

One of the key strengths of FPGAs lies in their exceptional parallel processing capabilities, enabling the simultaneous execution of multiple operations. The inherent parallelism of FPGA designs significantly enhances performance and speeds up inference tasks. Furthermore, FPGAs offer a high degree of customization, empowering developers to implement optimizations tailored to specific applications while maximizing overall performance.

In addition to their parallelism and flexibility, FPGA accelerators are renowned for delivering low-latency inference, making them ideal candidates for real-time applications. Moreover, FPGAs are highly energy-efficient, as they can be precisely configured to execute only the necessary computations for inference, minimizing power consumption while maintaining performance levels. Another noteworthy feature of FPGAs is their re-configurability, which allows their hardware to be easily reprogrammed and updated to accommodate changes in algorithms or models.

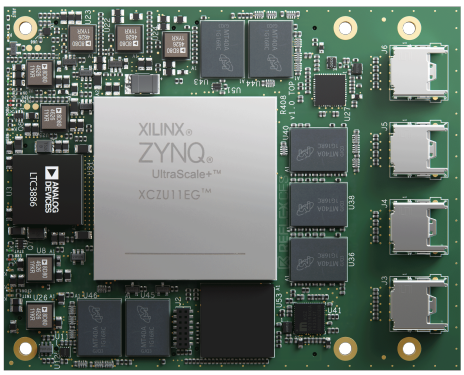

Figure 34: Zeus Zynq UltraScale [55]. Xilinx is a leading company in advanced programmable logic devices, particularly FPGAs

While FPGAs offer numerous advantages for developing tailored accelerators, it is important to acknowledge that implementing FPGA solutions requires highly specialized expertise in hardware description languages such as VHDL and Verilog. Particularly for complex algorithms like image or signal processing, the overall design efforts and time to market can significantly increase due to the inherent challenges in designing and validating RTL (Register Transfer Level) designs. However, platforms such as those provided by XILINX [56] can greatly assist developers in this journey. One notable tool is High-Level Synthesis (HLS), which introduces an additional layer of abstraction for describing the logic behavior of an algorithm.

2.6.1 Image Processing with FPGA

Digital image processing is a critical field in various technological domains, presenting increasing demands over time, including larger image sizes, higher bit resolution depths, and deeper color channels. Traditional software-based implementations of image processing algorithms are no longer optimal, particularly for real-time systems that require high throughput [57]. Image processing via hardware implementation poses a most viable solution for improving performance of image processing systems through the development of specialized hardware accelerators.

Two potential solutions for a hardware implementation are individual DSPs (Digital Signal Processors) and ASICs (Application-Specific Integrated Circuits). However, DSP units are limited in their configuration flexibility and lack the potential to reach the level of task parallelism that could be reached with a custom design in an FPGA. On the other hand, while ASICs offer the highest performance, they are inflexible, expensive, and require significant effort and time to produce. Additionally, GPUs (Graphics Processing Units) are another interesting option for image acceleration, as they provide high throughput, but they are not well-suited for low-power applications.

On the algorithmic side, particularly important for convolution tasks, is the buffer implementation. During convolution, each output pixel is computed by sliding an nxm window over the input image and performing the convolution operation between the window and the corresponding pixels. Storing the entire image in a buffer is highly inefficient in terms of time and memory space [58]. To address this, buffers such as shift registers, as illustrated in Figure 35, are commonly used. These buffers store only the width of an image, and are updated by consuming values incoming from a serial input stream to simulate the window sliding process. In cases where the output image needs to have the same dimensions as the input image, a process called padding is required. Padding involves increasing the size of the input image based on the dimensions of the kernel being applied. This is done by adding zeros or defined values to the borders of the image. By padding the input image, the convolution operation can be performed on every pixel, including those near the borders, preventing loss of information at the edges of the image.

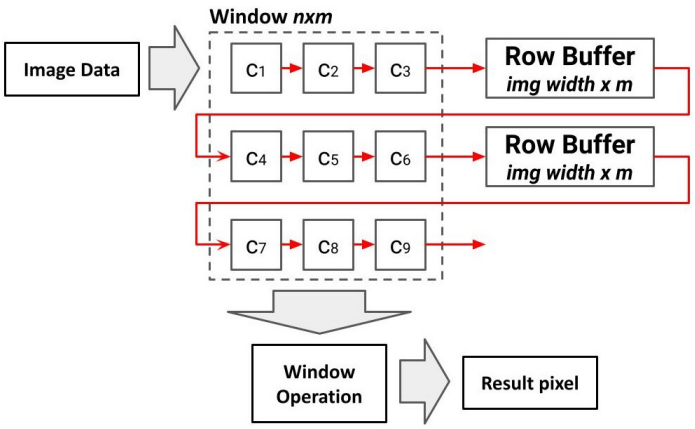

Figure 35: Diagram of hardware implementation of window filtering based on [58].

2.6.2 Vitis HLS

High-Level Synthesis (HLS) is a powerful technology that can efficiently transform behavioral logic described in high-level languages, such as C/C++ or SystemC, into digital hardware on a Register-Transfer Level (RTL). This tool offers an easier and quicker way of exploring design options, resulting in more effective hardware solutions with low time-to-market times. HLS also provides other essential benefits, including increased productivity, easier portability of solutions, and the re-use of designs and intellectual property (IP) with simple changes on optimization directives [59].

At the core of HLS tools are binding and scheduling, which are responsible for RTL generation by mapping control and data-flow operations into hardware design through an optimization process [60]. Scheduling determines the clock cycle in which an operation will occur, while binding determines the functional units used for each operation. The result of the optimization process depends on user directives and constraints such as latency, hardware resource utilization (also referred to as area), and throughput, as well as the available microelectronic technology. Typically, HLS tools prioritize performance (throughput) optimization, followed by latency and area optimization.

In Vitis HLS, developers can analyze the generated RTL design through various visualization tools and obtain insights into: task scheduling, pipeline analysis, latency, and hardware resource utilization (area). The hardware resources commonly used in synthesis reports are presented and explained below.

- Flip-Flops: the basic building blocks of digital circuits required for storing bits of data.
- DSP (Digital Signal Processing) Units: specialized hardware modules provided in some FPGAs that execute signal processing operations efficiently.
- LUT (Look-Up Table): configurable memory elements used to implement combinational logic functions.
- BRAM (Block RAM): dedicated on-chip memory resource used for storing and buffering data.
- URAM (UltraRAM): high-performance memory resource provided in some Xilinx FPGAs, offering larger storage capacity and higher bandwidth compared to traditional BRAM.

2.6.3 Optimization Directives

One significant advantage of HLS technology is the ability to use user directives and pragmas to configure the synthesis results for high-level code. HLS pragmas are embedded into the C/C++ code to ensure that specific optimizations will always occur during re-synthesis. Additionally, optimization directives can be specified as Tcl commands using an external file specific to the solution. This approach provides greater flexibility during the design exploration phase. If the solution is re-synthesized, only the embedded pragmas are applied. Some of these important HLS directives are explained below based on [61].

The pragma *#HLS inline* directive allows a function to be dissolved into a calling function, removing it as a separate entity in the RTL hierarchy. In some cases, this merging process allows operations within the function to be shared and optimized along with the calling function, resulting in improved performance. However, it is important to note that an inlined function will not be available for later reuse. As a result, consecutive calls to the inline function will result in increased RTL area, which can negatively impact the design's performance and resource usage.

The pragma *#HLS dataflow* directive enables task-level pipelining between functions, with the objective of increasing the concurrency of the RTL implementation and overall latency of the design. This optimization allows operations in a function to start before the previous function completes all its operations, as long as there are no data dependencies between functions and iterations. However, it is important to note that this optimization can increase the complexity of the design and may require careful management of the data dependencies between functions.

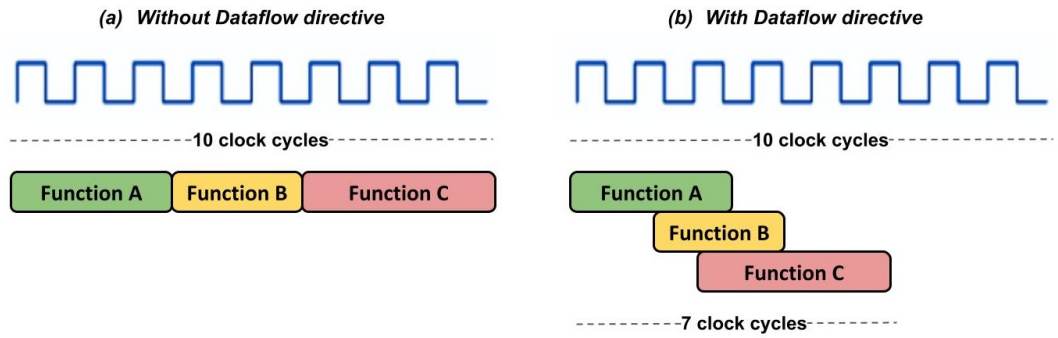

Figure 36: Example of implementation without and with dataflow optimization directive.

The pragma *#HLS pipeline* directive partitions the loop iterations into pipeline stages, allowing multiple iterations to be executed concurrently. This can significantly improve the performance of the design by increasing the throughput of the loop. However, it is important to note that pipelining also increases the resource utilization of the design, which may lead to increased area and power consumption. Designers must consider any data dependencies within the loop, as these may limit the scope to which the loop can be pipelined.

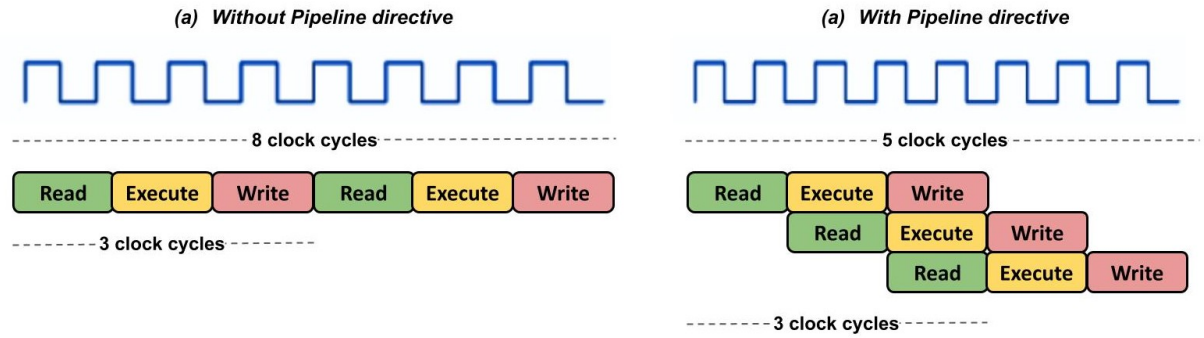

Figure 37: Example of implementation without and with pipeline optimization directive.

The pragma *#HLS array partition* directive divides an array into smaller sections, which can be helpful in achieving several parallel read/write accesses that cannot occur with only one array address. Partitioning an array results in an RTL implementation with multiple small memories and their respective addresses instead of one large memory, which can potentially increase parallelization potential in the design by allowing multiple memory accesses to occur concurrently, alleviating existing memory bottlenecks. However, it is important to note that partitioning an array requires additional memory instances or registers to be allocated, which can increase the area and power consumption of the design.

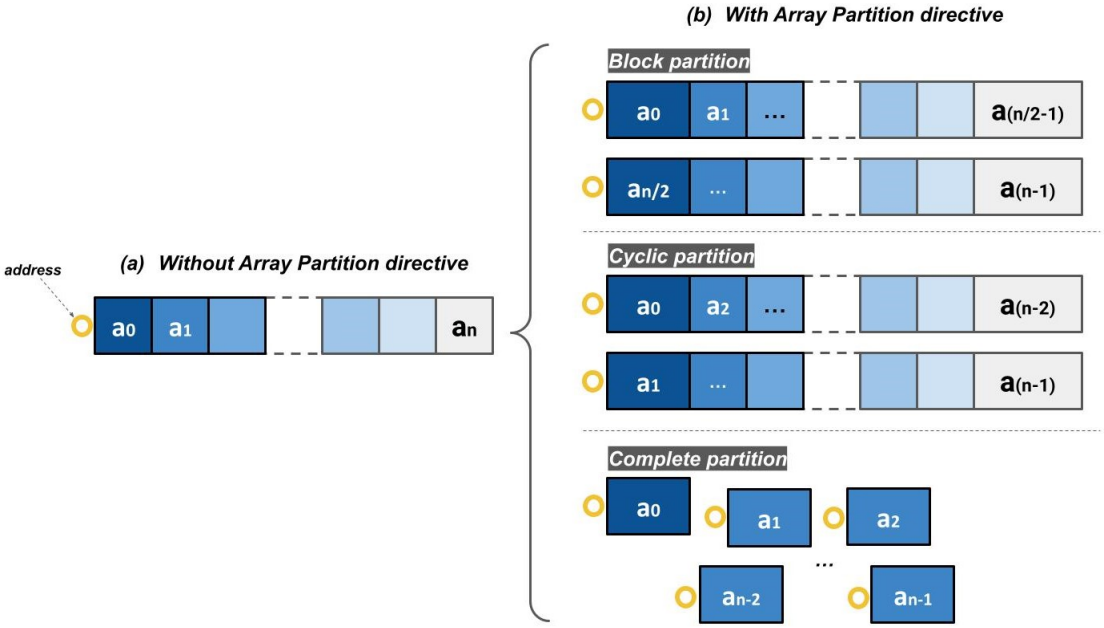

Figure 38: Example of implementation without and with array partition directive.

3 Design and Implementation

3.1 Design Considerations

The primary objective of this work is to evaluate potential image post-processing solutions for flare attenuation. However, several considerations need to be taken into account during this process to assess the prototype design and its results, with a focus on identifying areas for improvement and future optimizations. Here are some key considerations:

- Real-time performance: The solution should aim to achieve a processing speed close to 30 frames per second (FPS) to be suitable for real-time applications, such as automotive.
- Camera-agnostic solution: The solution should not only be designed for one type of camera or optic system.
- Embedded system compatibility: The solution should be designed to be deployable on an embedded system with constrained resources. It should be optimized to operate efficiently within these limitations.
- Subtle flare attenuation: The solution should be capable of attenuating flare artifacts in a subtle manner, ensuring that no additional artifacts or distortions are introduced into the image during the process.
- Flare artifact type: The solution should be designed to attenuate flare artifacts caused by the presence of a light source within or in close proximity to the camera's field of view.

By considering these factors, the evaluation of the prototype design and its results will provide insights into its effectiveness, potential improvements, and future optimizations for achieving successful flare attenuation in practical applications.

3.2 Flare Attenuation through Deconvolution

The previous chapter emphasized the significance of measuring the Point Spread Function (PSF) of the camera system for effective flare attenuation through deconvolution. Proper characterization of the PSF is of great relevance to evaluate and select the most suitable deconvolution algorithm, as well as to replicate the same results in productive or real environments. However, accurately characterizing the PSF is a complex task that involves several approximations and assumptions as discussed in Section 2.1.3. Since it is not feasible to measure the PSF of a specific camera in this project due to the lack of necessary equipment, a closed-loop evaluation procedure shown in Figure 39 has been proposed to assess the deconvolution algorithm's performance in a controlled environment. To assess the algorithm's effectiveness, an artificial image with added flare artifacts has to be generated by applying convolution between a flare-free image an a simulated PSF that could generate a realistic flare artifact. In addition, gaussian noise is added to the flared image as a final stage in simulating the processes of taking a picture with a camera. Then, the image is processed using the deconvolution algorithm to reconstruct the image while attenuating the flare. The quality of the reconstructed image will be compared with the original reference image (flare-free) using a comparison metrics such as SSIM. This comparison helps determine the extent to which the deconvolution algorithm reduces flare artifacts and helps decide between which deconvolution algorithm to choose and what will be the expected performance.

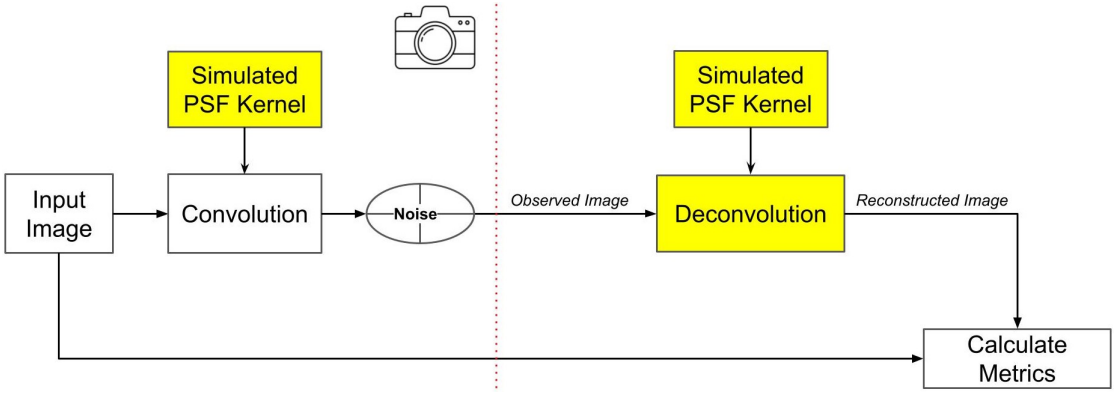

Figure 39: Closed-loop evaluation procedure to assess performance of deconvolution algorithm using a known simulated PSF.

The closed-loop evaluation procedure discussed in this context draws inspiration from works such as [62] and [63], which utilized a similar approach for generating artificial test images in a controlled environment. However, these works primarily focused on image deblurring, where the point spread function (PSF) kernel could be easily modeled and introduced to an image using image processing libraries like MATLAB. In the case of lens flare and stray light in general, there is a lack of literature and established methods for producing PSFs that could generate realistic artifacts on an image by means of convolution. Therefore, the following section will explore different approaches and methods that have been tested out during this project to generate artificial PSFs for assessing the efficiency of the deconvolution algorithm and the overall solution.

3.2.1 Genetic Algorithm

Genetic algorithms (GA), introduced in Section 2.2, are used to try to find a shift-invariant PSF kernel that could generate a lens flare-like artifact. For this purpose, two images are considered as shown in Figure 40, one with flare and the other without it. The objective is to find the kernel that transforms the original image (without flare) to the one with flare by applying 2D convolution operation. It is important to note that these tests were conducted on a grayscale image. This approach was chosen as each RGB channel possesses its own PSF, and for simplicity, it was more practical to focus on a single channel.

Figure 40: Example of original image (left) and target image with flare (right).

The genetic algorithm follows an iterative cycle consisting of several steps to optimize the PSF kernel weights. First, the GA generates an initial set of individuals or chromosomes, where each chromosome represents a potential PSF kernel as shown in Figure 41. The chromosomes are evaluated by reshaping them from their flattened form to a two-dimensional representation suitable for convolution. The reshaped kernel is convolved with the original image, and the resulting convolved image is compared to the reference flared image. The fitness value of each chromosome is calculated based on a chosen metric, such as Mean Squared Error (MSE), Mean Absolute Error (MAE), or Structural Similarity Index (SSIM). This value represents the similarity between the convolved image and the reference image, with a higher fitness indicating a better match. The fitness results along with its respective chromosomes are stored in a list called the mating pool. From this pool, individuals with the highest fitness values are selected as survivors and proceed to the mating process. The selection process ensures that individuals with better fitness have a higher chance of contributing to the next generation.

During the mating process, the selected survivors are combined using various techniques, such as crossover, to produce new offspring chromosomes. Moreover, the newly generated offspring undergo a mutation process, where random changes are introduced to some of the genes (weight values of the kernel). Mutation adds stochasticity to the algorithm and helps in escaping local minima, potentially leading to a better global optimal solution. Next, the mutated offspring replace a portion of the existing population, and the process repeats from the evaluation step. The algorithm continues iterating until a termination criterion is met, typically when the best chromosome with the highest fitness is found.

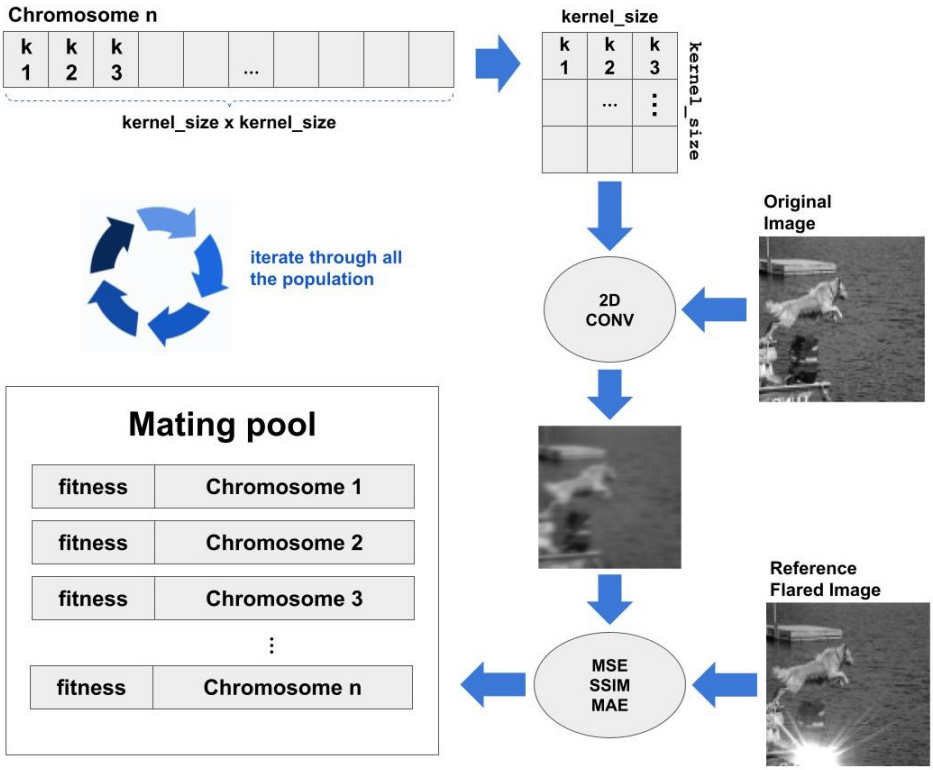

Figure 41: Explanation of important steps in the Genetic Algorithm for PSF kernel discovery.

Despite modifying several parameters (generations, mutation probability, chromosome size, and different fitness functions such as MSE, MAE, SSIM) in an attempt to achieve the highest fitness, the genetic algorithm (GA) did not succeed in finding kernel weights that generated a flare-like effect on the original image through convolution. One of the highest fitness results is shown in Figure 42, and demonstrates that the fitness gradually increases with each generation, but it plateaus around the 400th generation at a relatively low value. Despite some marginal improvements in subsequent generations, the algorithm fails to significantly enhance the fitness until the final generation. This suggests that the GA struggles to find an optimal solution for replicating flare artifacts on the original image due to the complexity of the search problem. The results show that only blurrier versions of the original image can be achieved, but no artifacts that could resemble lens flare. These results can be attributed to the required complexity of the shift-variant PSF, which is represented by a four-dimensional tensor, as explained earlier. A two-dimensional shift-invariant kernel may not be adequate to accurately model the PSF using the current search approach. Modifying the genetic algorithm to accommodate a more complex PSF representation would require a significant number of generations and a more advanced heuristic to approach an optimal solution, both of which are not part of the scope of this work.

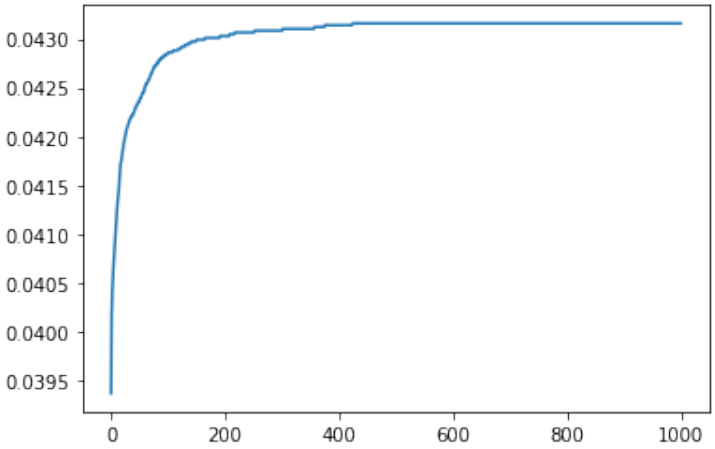

Figure 42: Fitness evolution curve through the generations.

3.2.2 Blind Deconvolution

As explained in the previous chapter, blind deconvolution is a technique used without necessarily having explicit information about the PSF. However, it is mostly used for recovering a sharp version of an input blurry image and no applications to flare attenuation have been reported. The MATLAB "deconvblind" image function was used for this purpose. The function requires an initial estimate of the PSF and the number of iterations it should try to reconstruct the image and estimate the PSF. The attempts of using this method resulted in kernels that could not approximate a flare-like effect on a clean image.

Figure 43: Blind deconvolution restoration result.

3.2.3 Using the Optical Parametric Model

In order to obtain the Point Spread Function (PSF) for every point in the camera's field of view (shift-variant kernel), a parametric model suggested in [8] is used instead of a simple interpolation technique. This parametric model, represented by Equation (7) and Equation (8), consists of two components: S_I, which models imperfections due to phenomena other than lens scatter, and S_D, which represents the scatter component. The parameters β, θ, A, B, d, f, and g are used to control how the model approximates the PSF based on the optical properties. In [64], measurements of the PSF were taken from a camera system, and a linear optimization process was used to estimate the parameters of the parametric model. The resulting parameters (β, θ, A, B, d, f, g) were then substituted into the parametric model equation, and the shift-invariant PSF kernel was obtained by evaluating the equation at the center of the image.

Table 1: Parametric model parameters calculated in [64].

Parameter	β	θ	d	b	α	f
Value	0.826	1.063	1.175	-0.000007427	-0.00001514	2.127

Following this same approach, but considering the obtained parametric model in [64] and presented in Table 1, the idea is to determine whether such a kernel can generate lens flare artifacts when convoluted with an image.

$$s_I\left(x_0, y_0, x_1, y_1; \beta, \sigma\right) = \frac{\beta}{\sigma} \exp\left(-\frac{\left(x_1 - x_0\right)^2 + \left(y_1 - y_0\right)^2}{\sigma^2}\right) \tag{7}$$

$$s_D\left(x_0, y_0, x_1, y_1; a, b, d, f, g\right) = \frac{d}{\left(\frac{\left(x_1 x_0 + y_1 y_0 - x_0^2 - y_0^2\right)^2}{\left[a\left(x_0^2 + y_0^2\right) + g\right]^2} + \frac{(-x_1 y_0 + y_1 x_0)^2}{x_0^2 + y_0^2}\right)^b \left(x_0^2 + y_0^2\right) + g\Bigg]^2}{f} + 1\Bigg)^f} \tag{8}$$

After multiple attempts using different kernel sizes, the results consistently showed similar outcomes as depicted in Figure 44. Instead of lens flare artifacts, the images exhibited a blurred version of the original image. Despite experimenting with various kernel sizes for the parametric model, the desired lens flare effect could not be achieved. This discrepancy arises from the fact that the PSF is unique to a particular camera system operating under specific conditions. Ideally, to obtain results, it would be necessary to use images captured with the exact same camera.

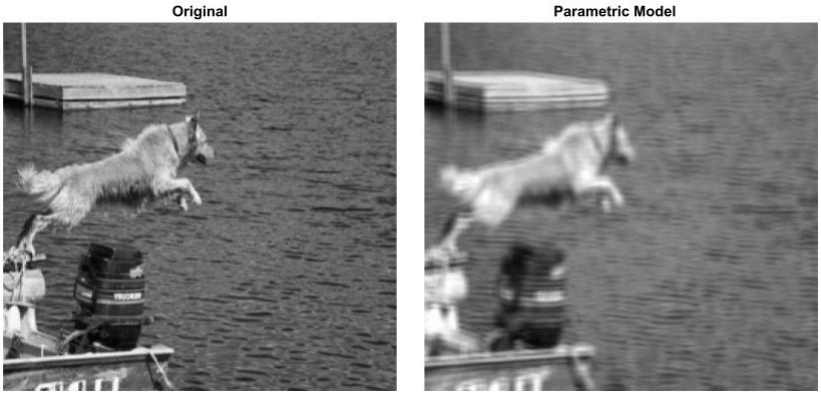

Figure 44: Results of using parametric model.

3.2.4 Fast Fourier Transform

Considering the fact that a convolution operation in spatial domain such as the one in Equation (1) can also be described as a multiplication operation in the frequency domain as seen in Equation (9), it is theoretically possible to retrieve the PSF if both of the other variables are known. To this end, two images with and without flare are used to clear out the PSF in the frequency domain.

$$PSF(f) = \frac{\text{Flared\_Image}(f)}{\text{Original\_Image}(f)} \tag{9}$$

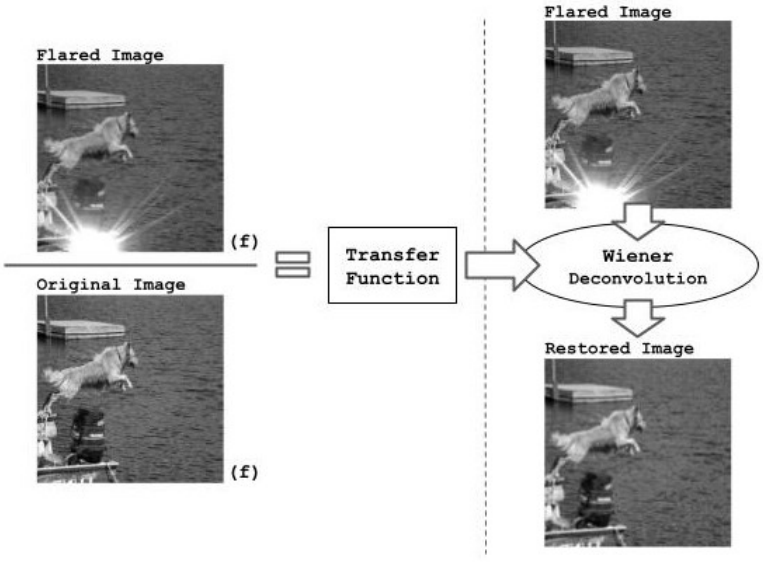

Figure 45: Process of using the Fast Fourier Transform (FFT).

Although the transfer function obtained by dividing the flared and original images in the frequency domain successfully reconstructs the image without flare through the deconvolution algorithm (as shown in Figure 45), it does not represent the Point Spread Function

(PSF) of an optical system. Instead of attenuating the flare artifact by modeling the behavior of stray light, the coefficients of the transfer function have been determined so that their weighted combination, applied through deconvolution, reproduces the original image. Treating this transfer function as a PSF to introduce lens flare into other images will result in incoherent results, as shown in Figure 46. This is because the transfer function is very specific to the pair of images used for its estimation.

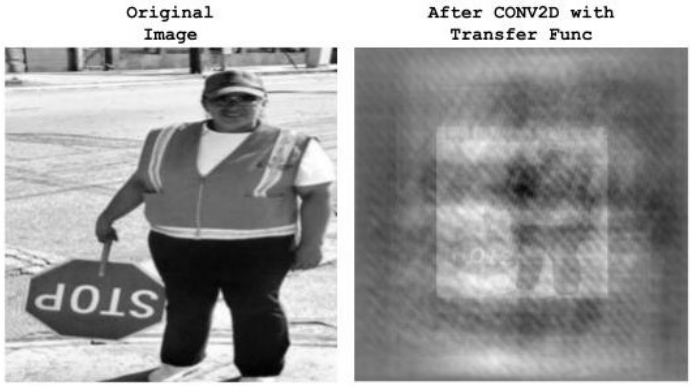

Figure 46: Example of using the transference function (PSF) on a different image to generate a flare artifact.

3.2.5 Flare Attenuation through Deconvolution - Assessment

Lens flare attenuation poses a significant challenge that has not been extensively addressed with image processing techniques. The limited amount of studies point towards the use of image reconstruction through deconvolution and stray light characterization (PSF modeling). However, after conducting a thorough literature review and a practical pre-study to determine the feasibility of utilizing the closed-loop process depicted in Figure 39, it has become evident that this approach is not viable due to the following reasons:

1. The estimation of a real PSF of an optic system requires specialized tools and setups that were not accessible for this particular study. As a result, the characterization of the PSF and its subsequent utilization in the analysis of the deconvolution algorithm performance were not feasible. Important to mention that the reconstruction deconvolution algorithm can only be applied to images taken by the same camera system used to characterize the PSF.

2. Despite conducting various tests in Section 3.2, it was not feasible to accurately replicate a simulated PSF kernel capable of generating realistic lens flare artifacts. The convolution operations between the artificially generated PSF and a clean image resulted in a blurred version of the original image rather than a flare artifact that accurately represents real-life scenarios. Without a valid PSF it is not possible to assess the effectiveness of any deconvolution algorithm, let alone compare multiple algorithms using image quality metrics.

3. Even if a known PSF is available, deconvolution algorithms for stray light attenuation have primarily been developed and tested for static camera system setups [64] [7], or in general static optical system setups such as in microscopy, astronomy, or healthcare applications [4], [8], [6], [10]. These algorithms have been tested and proposed for such controlled environments. However, deploying these solutions in dynamic and rapidly changing environments, such as in industrial applications, presents additional challenges. The PSF in these scenarios can vary significantly due to factors like camera motion, changing lighting conditions and especially lens wear and tear, which will require a frequent PSF calibration for each of the RGB channels.

4. Iterative deconvolution algorithms, such as Richardson-Lucy introduced in Section 2.1.4, have demonstrated their suitability for image reconstruction in the presence of varying PSFs. However, one important limitation is that there is no fixed number of iterations that guarantees optimal reconstruction. Determining the optimal number of iterations can be challenging and computationally expensive. This makes it difficult to estimate the processing speed of the algorithm, which is crucial for real-time applications.

3.3 Flare Attenuation with Deep Learning

As introduced in Section 2.4, deep learning approaches have shown remarkable results in image restoration for lens flare artifacts attenuation in recent years. This is achieved through the use of convolutional neural networks (CNNs), which can learn to extract relevant features from the input image and generate the corresponding output image with high accuracy. The proposed solution contemplates the use of deep learning due to the following main reasons:

1. Deep learning-based approaches do not require any PSF characterization of a specific camera system, making it more robust than traditional deconvolution techniques and enabling a camera agnostic solution for flare attenuation.
2. Deep learning-based approaches have demonstrated their superiority over traditional methods in handling various factors that are commonly present in real-life scenarios, including varying scenes, camera and lens types, light exposure, camera angles, and other environmental factors.
3. Previous studies have demonstrated promising outcomes in flare attenuation tasks using well-established convolutional neural network (CNN) architectures, such as U-Net.
4. Previous studies have successfully demonstrated the feasibility of using synthetic flare datasets to train deep learning models for accurate predictions on real-life data. By leveraging synthetic data during the training phase, the deep learning model can learn and capture the essential features and characteristics of flare artifacts.
5. The existing deep learning flare attenuation approaches have not been designed for deployment in constrained embedded systems due to their high complexity. By considering the previous studies as baselines, it is possible to leverage their insights to guide through exploration and analysis while carefully assessing the trade-off between model complexity and performance.
6. While deep learning offers several advantages, it can also have its drawbacks. One significant challenge is the difficulty in interpreting and explaining the reasoning behind inference predictions, especially when compared to more deterministic approaches used in traditional image reconstruction algorithms. Consequently, a considerable number of synthetic images are necessary to thoroughly test the solution and statistically verify its ability to reduce flare without introducing additional artifacts to the image.
7. On the other hand, implementing deep learning architectures like convolutional networks in hardware poses additional challenges. These networks often involve a large number of filters, with each convolutional layer requiring computations on different-sized dimensions. These dimensions are typically significant, which necessitates multiple iterations to compute. As a result, the hardware implementation of such complex networks becomes more intricate and must be evaluated carefully.

3.3.1 Dataset

As already mentioned, the most feasible approach to training a deep neural network for flare attenuation is by generating a synthetic dataset that is complex and diverse enough to enable the network to learn and generalize to real-life scenarios. In this work, the flare dataset is selected from [28] because of its high quality, diversity of lens flare patterns and the fact that it has already been used for training a deep learning model. The other public flare dataset [39] is especially created for night applications and does not include real flares. Therefore, the selected dataset comprises 2000 synthetic flares and 3000 real flares captured by a camera rotating around a light source in a dark room. Moreover, the scene dataset is based on the Flick 30k [65]. Figure 47 shows samples of both datasets.

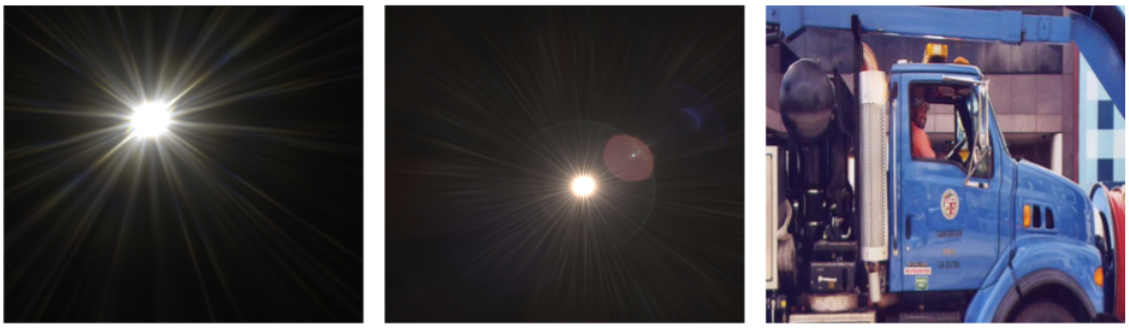

Figure 47: Sample images from the flare (synthetic and real) and the scene datasets.

The procedure to generate the synthetic dataset involves several steps based on a method proposed in [28] and illustrated in Figure 49. First, a scene and a flare image are randomly selected from their respective datasets. Then, the flare image undergoes random scaling, rotation, and color augmentation procedures introduced in Section 2.3.6. Finally, the modified flare image is overlaid onto the scene to create a composite image, which is used as the input for the deep learning model. On the other hand, the corresponding original image of the scene is used to create the ground-truth image. This process is repeated multiple times with different random combinations of flare and scene images to create a diverse and complex synthetic dataset of 32,000 pairs of images (with and without flare) as seen in Figure 48. The script used to merge the flares and scenes datasets is based on [28] and it is available in the GitHub repository for this project in Appendix 7.1.

Figure 48: Example of scene with flare and original scene.

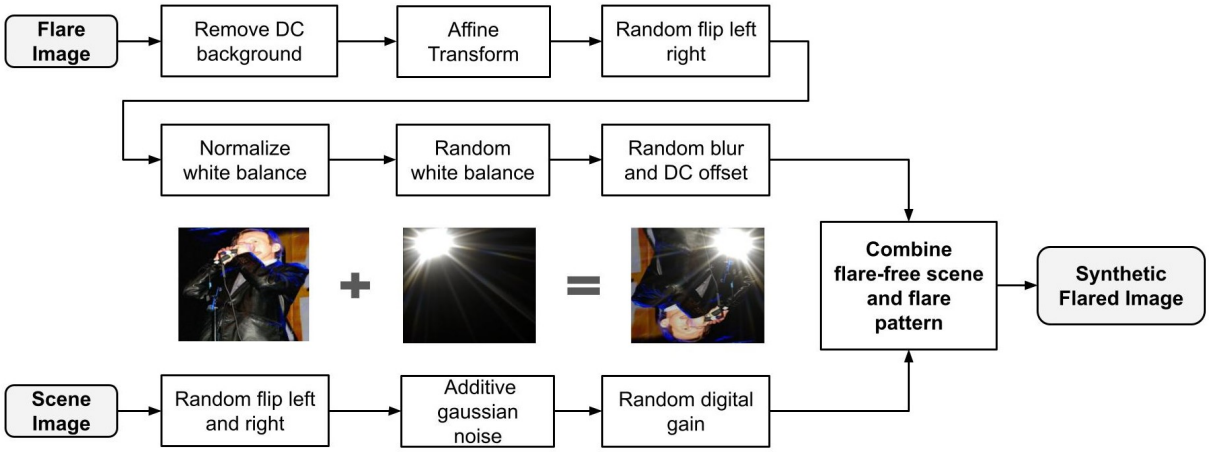

Figure 49: Procedure to merge scenes and flares to generate the synthetic dataset based on [28].

3.3.2 Masking Saturated Pixels on Target Image

To ensure that the neural network focuses on attenuating the flare and not the light source, it is important to address the issue of saturated pixels that occur when the light source generating the flare is within the camera's field of view. These saturated pixels have RGB values that are far beyond the point of recovery. Therefore, the neural network should not focus on removing the light source, otherwise it will be a waste of model capacity and the resulting image would be susceptible to artifacts appearing where the light source (saturated pixels) is supposed to be. It is important to recall that the solution is meant to attenuate the flare and not eliminate the light source. Therefore, the ground-truth image is modified by removing saturated pixels from consideration. The new ground-truth is computed by using the flare and its corresponding scene, where the pixels with an RGB value above a defined saturation threshold (0.97) are passed into the ground-truth scene. The result of this process is illustrated in Figure 50. This modification ensures that the neural network learns to attenuate the flare without focusing on the saturated section.

Figure 50: Example of scene merged with flare (left) and new ground-truth with saturated pixels (right).

3.3.3 Neural Network Model - Overview

Section 2.4 of this work, discussed how a U-Net based architecture was chosen for the task at hand. However, it is important to note that the original U-Net with 30 million parameters was not evaluated in this study due to its large size. Instead, the evaluation started with a smaller model inspired by the structure of U-Net and implemented using transfer learning. In this case, MobileNetv2 [34], specifically designed for applications with limited computing resources, was selected as the backbone due to its compact size compared to other well-known models like Resnet18 [41] or VGG16 [42].

To further reduce the complexity of the transfer learning network, the scaling parameter (alpha) of MobileNetv2 was adjusted to strike a balance between accuracy and complexity. The alpha parameter controls the number of channels in the network, with a value of 1.0 indicating maximum width, using the full set of channels specified in the original architecture. In this work, an alpha value of 0.35 was used, significantly reducing the original design network from around 2 million parameters to 141,646 parameters.

Figure 51 illustrates the architecture of the transfer learning version, referred to as FlareNet-TL. However, as mentioned in Section 2.4.1, it is important to acknowledge that the model still presents a complex and deep architecture, indicating the intricacy of the chosen approach. A diagram of the model can be seen in the GitHub repository of this project in Appendix 7.1, showing its complexity and the amount of layers it contains.

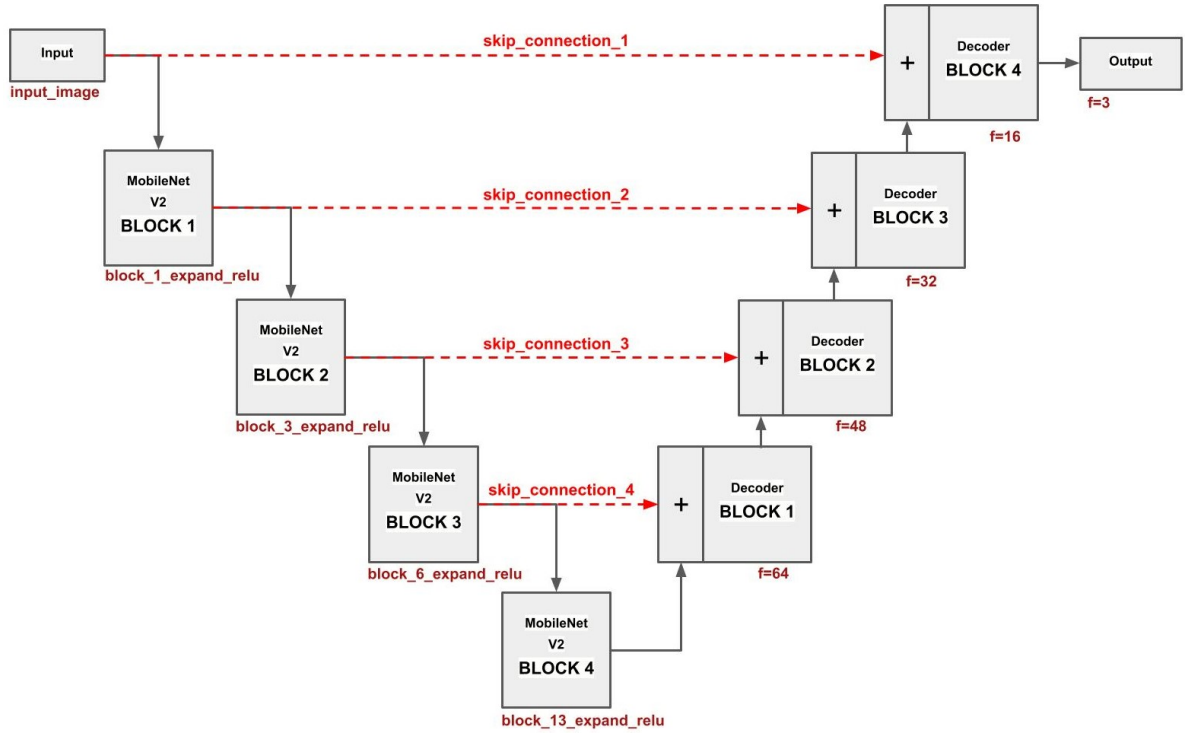

Figure 51: Diagram of FlareNet-TL model

Although the transfer learning-based network demonstrates promising results, part of its architecture inherited from MobileNetv2 poses an extra complexity when it comes to hardware implementation due to its higher number of layers and parameters. To address this limitation and seek a simpler yet effective neural network, a new model without transfer learning is proposed inspired on more compact models such as the C-UNet [44] and HC-UNet [46], introduced in Section 2.4.2.

Through several iterations, different hyper-parameters and CNN layers were evaluated to identify the optimal architecture with good performance while keeping the parameter count limited following a similar approach to [44]. The best architecture, depicted in Figure 52, emerged from this iterative process. During the exploratory phase, significant modifications were made to the network's depth, filter sizes, and inclusion of skip connections. These adjustments, which will be further explained in Section 4.3, were crucial in achieving the desired performance while maintaining a compact parameter count.

The resulting model, referred to as FlareNet-simple, has a total of 92,051 parameters. Further insights gained during the exploratory phase will be elaborated on in the subsequent chapter. However, it is important to mention that a smaller model was also tested, resembling the C-UNet++ architecture [44]. However, during the training process, minimal to no improvement was observed, indicating that the model was too small for the task at hand. It was concluded that a good trade-off between accuracy and model complexity can be achieved within a vicinity of 92,000 parameters.

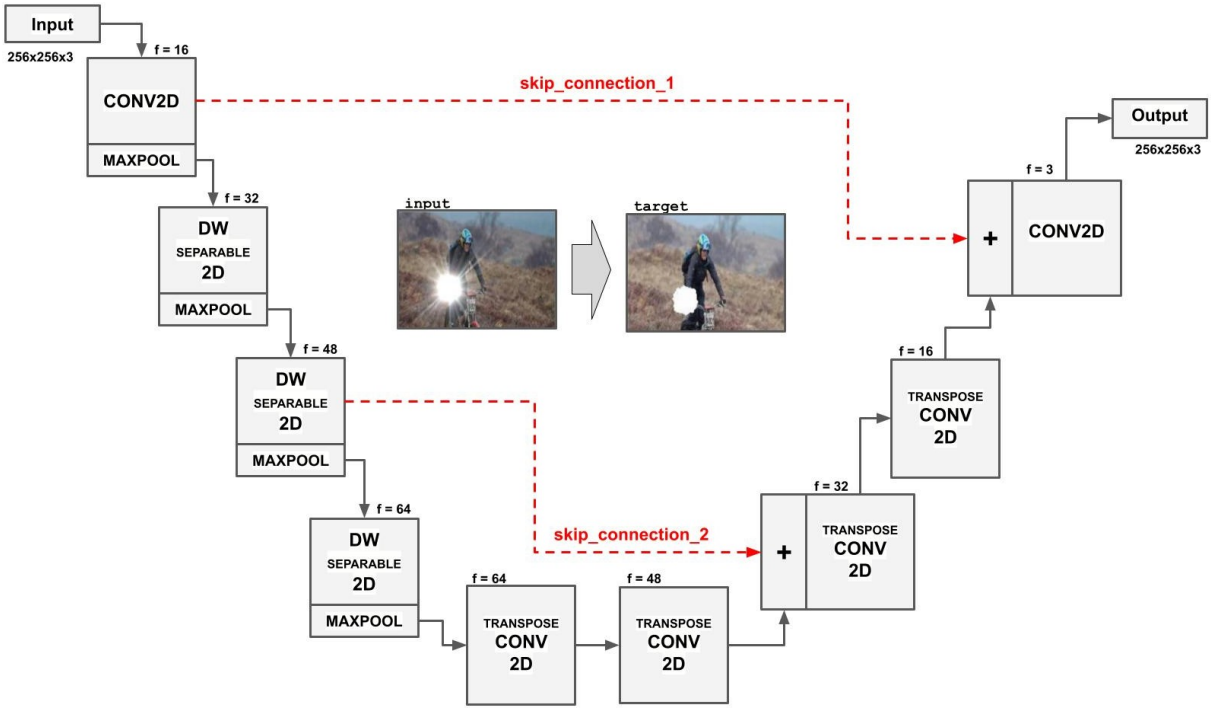

Figure 52: Diagram of FlareNet-simple model.

3.3.4 Training

This section provides technical details of the deep learning implementation used in the project. The complete script used for training and validation, including detailed comments and instructions on how to run it, can be accessed on the project's GitHub repository in Appendix 7.1. To build and train the proposed versions of the FlareNet neural network, we use Jupyter Notebook, which allows for faster training using a GPU. For instructions on how to configure the setup for GPU training, please refer to the Appendix. Additionally, TensorFlow is used as the deep learning framework, based on the author's prior experience with it. An overview of the training pipeline is depicted in Figure 53.

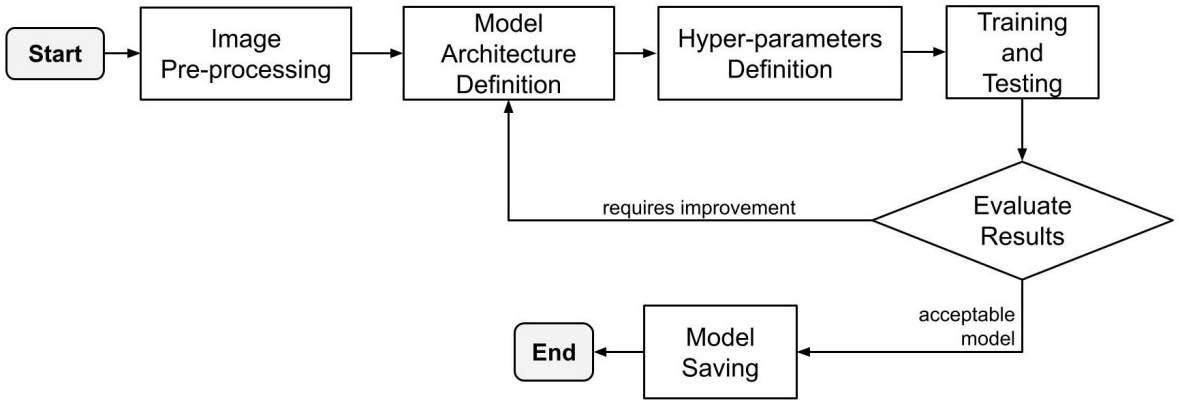

Figure 53: Overview of the process to train the deep learning model.

Important to mention that the following libraries are used along with TensorFlow in this implementation: i) Scikit-learn: Open source library based on python for machine learning applications, ii) OpenCV: Open source Computer Vision library, iii) Numpy: Open source library based on python, highly optimized to work with large multidimensional vectors and matrices, along with math functions to operate on them.

Image Data Pre-processing
Furthermore, the most important steps to condition the input and output (ground-truth) images are the following: i) resizing image to a standard size of 255x255 pixels, ii) normalization of RGB values, resulting in 255x255x3 matrix of floating point values between 0 and 1, and iii) train-validate-test split of the 32,000 image dataset in a random way following a 80%, 10% and 10% ratio respectively. Overall, 25600 instances are used for training, 3200 for validation, and 3200 for testing.

U-Net Architecture Definition
After several iterations to find the most suitable architecture and hyper-parameters, the proposed model is implemented in this section using the Keras API with a TensorFlow backend as presented in Listing 1. In this section, only the implementation of the FlareNet-simple model will be explained, the implementation of its counterpart (FlareNet-TL) can be found in the GitHub repository. The first layer is a convolutional layer with 16 filters of size (3, 3) and applies a ReLU activation function. The resulting

feature map is then max-pooled with a kernel of size (2, 2). The next layers follow a similar pattern of a separable convolution layer with a ReLU activation function and a max-pooling layer. The number of filters is doubled for each layer, resulting in 32, 48, and 64 filters, respectively. The network then applies a series of transpose convolution layers with strides of 2, effectively upsampling the feature map back to the original input size. The transpose convolution layers are followed by skip connections that merge the output of the corresponding convolutional layer in the encoder section and the input of the transpose convolution layer in the decoder section by an adding operation. Finally, the output of this layer is fed to another convolutional layer with 3 filters and a sigmoid activation function, which produces the final output image with the same dimensions as the input image.

Training
After defining the hyper-parameters, such as loss function, optimizer, kernel initializers, and the number of epochs (which will be discussed in Sections 4.1.1 and 4.1.2 for the best models), two callback functions are instantiated: i) Early Stopping: form of regularization used to avoid over-fitting during training. The learning process stops if the accuracy does not improve after a number of epochs called the "patience", and ii) Reduce Learning Rate: reduces the learning rate during training by a given "factor" after no improvement is seen for a "patience" number of epochs.

Model Evaluation
Lastly, the model is evaluated using the test dataset to assess its performance on unseen data, providing an indication of its generalization capabilities. After evaluation, the model is saved to preserve its architecture, weights, optimizer, and training configurations for future reference and reproducibility.

Listing 1: Model Definition

```
def FlareNet_simple():

 inputs = Input((256, 256, 3))

 x1_skip = Conv2D(16, (3, 3), padding='same')(inputs)
 x1_skip = Activation("relu")(x1_skip)
 x2 = MaxPooling2D((2, 2))(x1_skip)
 x3 = SeparableConv2D(32, (3, 3), padding="same")(x2)
 x3 = Activation("relu")(x3)
 x4 = MaxPooling2D((2, 2))(x3)
 x5_skip = SeparableConv2D(48, (3, 3), padding="same")(x4)
 x5_skip = Activation("relu")(x5_skip)
 x6 = MaxPooling2D((2, 2))(x5_skip)
 x7 = SeparableConv2D(64, (3, 3), padding="same")(x6)
 x7 = Activation("relu")(x7)
 x8 = MaxPooling2D((2, 2))(x7)

 x9 = Conv2DTranspose(64, (3, 3), padding="same", strides=2)(x8)
 x9 = Activation("relu")(x9)
 x10 = Conv2DTranspose(48, (3, 3), padding="same", strides=2)(x9)
 x10 = Activation("relu")(x10)

 skip1 = Add()([x5_skip, x10])
 skip1 = Activation("relu")(skip1)
 x14 = Conv2DTranspose(32, (3, 3), padding="same", strides=2)(skip1)
 x14 = Activation("relu")(x14)
 x15 = Conv2DTranspose(16, (3, 3), padding="same", strides=2)(x14)
 x15 = Activation("relu")(x15)

 skip2 = Add()([x1_skip, x15])
 skip2 = Activation("relu")(skip2)
 outputs = Conv2D(3, (1, 1))(skip2)
 outputs = Activation("sigmoid")(outputs)

 FlareNet_simple = Model(inputs, outputs)

 return FlareNet_simple
```

3.4 Model Quantization

The trained FlareNet-simple is quantized from a 32-bit floating point representation to a 8-bit integer representation with the goal of considerably reducing the amount of memory space required to store the weights and speed up computation by requiring only integer arithmetic units instead of floating point ones. As previously explained in Section 2.3.8, quantization aware training usually results in a model with better performance.

Therefore, this section dives into the technical side of the deep learning quantization aware training. The complete script which includes training and performance evaluation can be found in the GitHub repository of this project in Appendix 7.1. Jupyter Notebook is again used with the same training setup along with the GPU. To quantize the model, the TensorFlow Lite [36] model optimization library is used, which allows to load an already trained model and further train it by considering the weights as 8-bit integer values. The model is trained for a short amount of epochs with the same dataset and hyper-parameters which are detailed later in Section 4.1.3. This process is done following the code in Listing 2.

Listing 2: Quantization Aware Training

```
import tensorflow_model_optimization as tfmot

quantize_model = tfmot.quantization.keras.quantize_model
q_aware_model = quantize_model(trained_model)
q_aware_model.compile(loss=SSIMLoss,
                      optimizer = Nadam(0.001),
                      metrics=[SSIMLoss])

history = q_aware_model.fit(train_dataset,
                    validation_data=valid_dataset,epochs=EPOCHS,
                    steps_per_epoch=train_steps,
                    validation_steps=valid_steps)
```

Once the model has been trained, the model is exported into a Tensor Flow Lite format and saved for further inspection as shown in Listing 3. The quantized weights of each layer can be analyzed using the Netron application [66], which allows for easy inspection and extraction of the model parameters.

Listing 3: Tensor Flow Lite Model Conversion

```
converter = tf.lite.TFLiteConverter.from_keras_model(q_aware_model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
quantized_tflite_model = converter.convert()
```

However, extracting the bias weights of the 2D Transpose layer from the TensorFlow Lite model was not possible due to incompatibility issues that require further investigation. As a result, we will utilize a fixed-point representation that is defined in Section 3.5.3.

3.5 Hardware Implementation with HLS

As an important part of the-proof-of-concept, the aim of this section is to estimate the hardware resources required to implement the FlareNet-simple architecture as a digital circuit. To achieve this, each layer of the architecture is modeled and implemented using HLS, introduced in Section 2.6.2. The relevant parts of the implementation will be presented and explained, but the theoretical justification for the algorithmic implementation of each layer can be found in the previous chapter. For the complete implementation, please refer to the project repository in GitHub found in Appendix 7.1.

3.5.1 Design Overview

The overall data flow of the neural network model is shown in Figure 54. The architecture consists of five different types of layers: i) 2D Convolution, ii) 2D Max-Pooling, iii) 2D Depth-wise separable convolution, iv) 2D Transpose Convolution, and v) Adding, each of which will be implemented and explained in detail. Additionally, each layer considers buffers to cache input and intermediary values during the inference process. The data is transmitted from layer to layer using stream data types and each layer is implemented as templated functions to be called with corresponding values for parameters such as input/output size, input/output depth, kernel weights, among others. The entire implementation model is built in the FlareNet function, including reading the input image data and writing the output inference results.

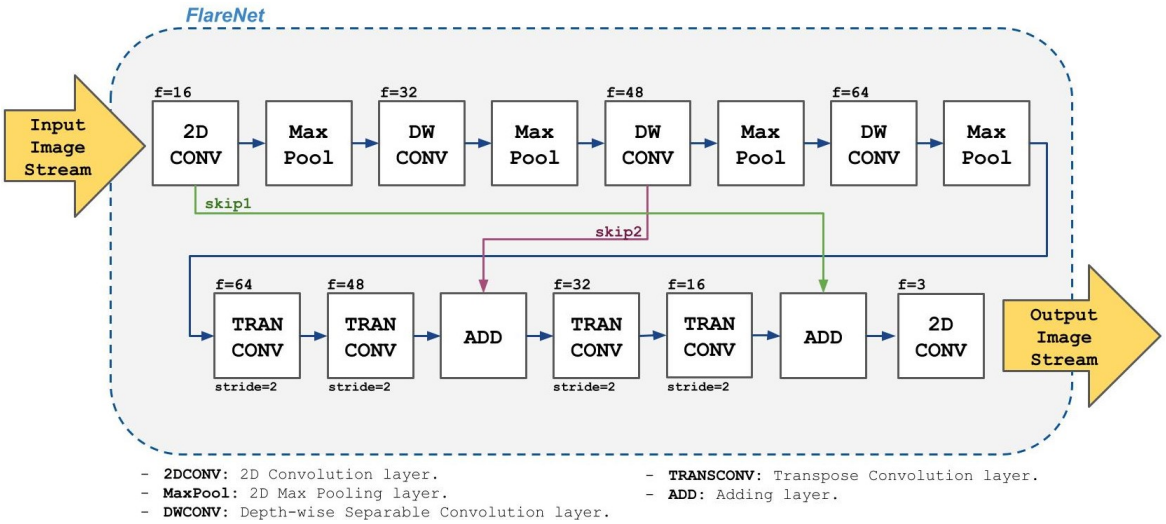

Figure 54: Diagram of the structure of the FlareNet model to be implemented in HLS.

3.5.2 Vitis HLS - File Architecture

The file architecture of an HLS implementation for the neural network follows the same good practices as a C++ design. The file structure, as depicted in Figure 55, is utilized to implement the FlareNet-simple neural network in Vitis HLS.

- The "weights.h" file contains twenty-one multidimensional arrays that correspond to the filter weights for each respective layer. These weights are extracted from the trained FlareNet neural network. The "FlareNet.cpp" file invokes this "weights" file to utilize the filter weights.
- The "FlareNet.cpp" and "FlareNet.h" file defines various constant parameters, including input size, output size, input depth, and output depth of the neural network. Additionally, it includes functions related to each of the five types of layers present in the network: 2D Convolution, 2D Depth-Wise Separable Convolution, 2D Max-Pooling, 2D Transpose Convolution, and Adding. Moreover, it contains functions for the buffers along with their initialization and update procedures. Finally, it also constructs the structure of the FlareNet neural network as seen in Figure 54 by connecting different layers and passing the corresponding parameters to each layer.
- The "FlareNet_TestBench.cpp" file in the HLS implementation is responsible for reading input stimuli, which are images with flare artifacts. It then calls the FlareNet function to perform inference on these input stimuli. The resulting output from the FlareNet model is compared with the "golden" reference result obtained from a software execution. If the output and the reference result are equivalent, the test is considered to have passed. The "Test Bench" file serves as the main function in the HLS implementation, overseeing the testing and verification process.

Overall, this file structure enables the implementation of the FlareNet neural network in Vitis HLS, allowing for hardware synthesis and verification through the test bench.

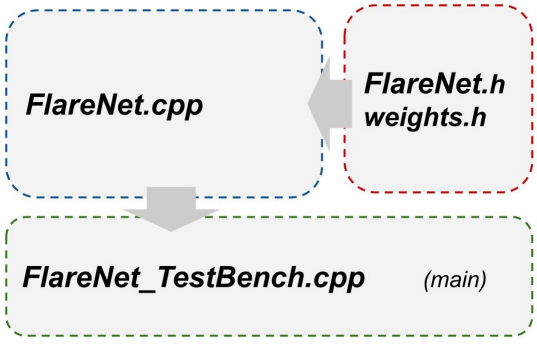

Figure 55: File architecture of the HLS implementation in VITIS.

3.5.3 Functions and data types

As mentioned earlier, the implementation of the model introduced in Figure 54 involves five different types of layers, each implemented as an individual function. However, the model requires the usage of these layers in multiple instances with varying parameters, such as input tensor size, depth, filter kernel size, and filter depth. To achieve flexibility and modularity in the design, VITIS HLS supports native C++ templates. When different C++ template values are passed to a function, unique instances of the function are created for each template value. During synthesis with VITIS HLS, these copies are synthesized independently [67]. The template parameters can be used to define the characteristics of the input tensor, such as size and depth, as well as other configurable options specific to each layer as shown in the Listing 4. Additionally, it enables easier exploration of different design configurations and evaluates their performance.

Listing 4: Functions and Templates

```
MaxPooling2D<in_size, pool_size, out_depth>
                (stream_in,stream_out);

Conv2D_transposed<in_size, in_depth, pool, out_size, out_depth>
                  (stream_in, stream_out, trans_weights, trans_bias);
```

An important element considered during the implementation in HSL are the streams. Streams are usually applied to efficiently handle data transfer between different modules in a hardware design, due to the fact that it provides a standardized communication protocol that facilitates parallel processing of data [67]. There are several benefits to using streams in the design, such as enabling dataflow-oriented designs and other optimizations such as data buffering, data parallelism, and pipeline scheduling that can improve the performance, latency, and resource utilization of the design. Listing 5 shows how the streams are defined in HLS.

Listing 5: Streams

```
hls::stream<datatype_input> input_stream;
hls::stream<datatype_output> output_stream;
hls::stream<datatype_internal> stream_0;
```

Selecting the appropriate data type representation for the input and output image values as well as the model weights is very important as it will impact the use of resources and the latency of the design. Arbitrary Precision (AP) fixed data type is a commonly used number representation that provides flexibility and allows for precise control over the range and resolution of fixed-point numbers [67]. Listing 6 shows how this data type is defined in HLS. The <A, B> notation indicates that the fixed type has a total width of A bits, with B of those bits reserved for the fractional part (decimal places), and the remaining $A - B$ bits allocated for the integer part.

Listing 6: Datatypes

```
typedef ap_fixed<18, 8> model_type_in;
typedef ap_fixed<18, 8> model_type_out;
typedef ap_fixed<18, 8> model_type_weights;
```

In the context of this application, we have allocated 10 bits for the integer part and 8 bits for the fractional part (decimal places). The selection of the number of bits is done based on comparing the image inference results from the implementation in C++ with the inference results directly from the deep learning framework with full bit representation (float32) to select the smallest fixed-point representation that generates minimum differences between both of them. This is essential to ensure that values in internal layers during inference calculation have a sufficient bit range representation to prevent truncation or overflow issues as it is possible to see in Figure 56 if less bits are used for the integer section.

Figure 56: Examples of different FlareNet inference results depending on the fixed-point resolution.

3.5.4 Buffers

Buffers play a crucial role in the implementation of CNNs as the convolutional layers require the input information to be cached in order to have the multi dimensional values needed for the convolution operation. In addition, they provide temporary storage for intermediate results during forward propagation in the neural network. For this application, two types of buffers were designed: with and without zero-padding. The zero-padding buffer is utilized in the 2D convolutional and 2D depth-wise separable convolutional layers. It works by adding zeros to the edges of the input tensors, which expands the tensor size and makes it compatible with the size of the filter. On the other hand, the simple version of the buffer (without zero-padding) is used only in the Max-Pooling layer.

The dimensions of the input buffer depends on the kernel size, tensor input length, and tensor input depth. The convolutional window is a section of the input buffer that is the same size as the kernel size. The two input buffer structures are similar, but the buffer with zero-padding is larger due to the logic to add zeros to both sides of the image (left and right), as shown in Figure 57 and Figure 58. In addition, the simple version of the

buffer for max-pooling utilizes only "kernel-1" amount of line buffers instead of "kernel" line buffers as its simpler logic allows it.

Each type of buffer has its own function for initializing its values. The "initialize buffer function" is used to fill the buffer with values from the input stream. This function fills the buffer from left to right and top to bottom, with the lower right corner being the last value to be inputted. The difference between the initialization of the two buffers can be seen in Figure 57 and Figure 58. The zero-padding buffer introduces zeros in the first and last columns and the first row of the buffer, while the values in between are filled based on the same logic as the non-zero-padding buffer.

It is important to note that VITIS HLS provides a dedicated C++ class called "line buffers" [68] for handling this type of convolutional buffers and includes buffering functions such as shift, insert, get, etc. However, our attempts to synthesize a solution using them were unsuccessful. This could potentially be attributed to an issue with the VITIS software version running on Windows, which requires further investigation. Similarly, the problem arises when attempting to use vector classes. While using the native "line buffers" could potentially aid in synthesizing a more efficient design, implementing the buffers as normal arrays simplifies the debugging process, as the flow of information in the arrays is easier to follow.

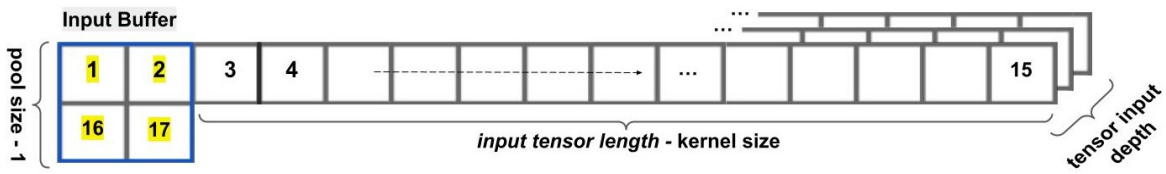

Figure 57: Example of initializing max-pooling input buffer with kernel pool size of 2x2.

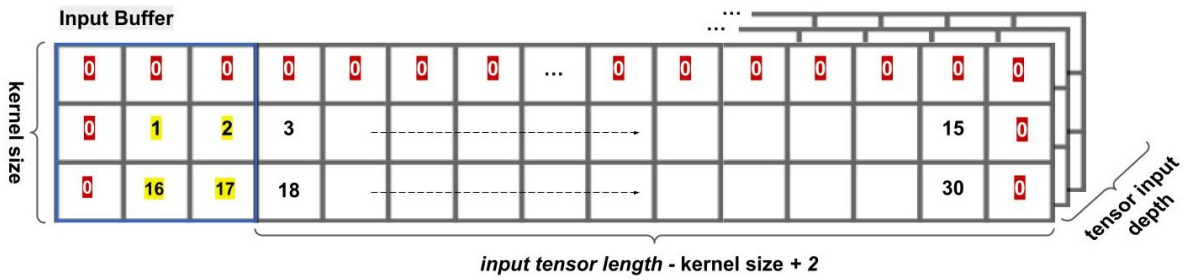

Figure 58: Example of initializing input buffer with zero-padding with kernel size of 3x3.

Each type of buffer has its own function for updating its values. The "update buffer function" shifts all the existing values in the input buffer one step to the left. In addition, the last two values from the first row of the convolution window are copied back to the first two rows of the last column of the input buffer, as represented by the two blue squares in Figure 59. This is because these values will still be required in future iterations, replicating the behavior of the window sliding one stride down with a kernel of size 3x3. Finally, a new value from the input stream is added in the last column and row of the input buffer, as shown by the red square in Figure 59. The difference in the behavior of the update

function for the two types of buffer is based on how the zero-padding buffer handles the borders to input zero values whenever it is necessary. Therefore, it must evaluate whether the window is positioned in the first column, last column or last row of the input tensor.

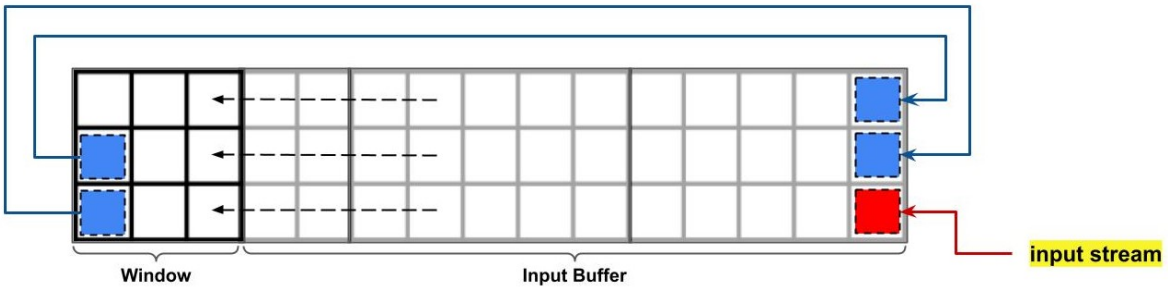

Figure 59: Example of how values are updated in the zero-padding input buffer.

The following will describe only the two functions related to the zero-padding buffer, the rest can be found in the GitHub repository. The "initialize input buffer" function is described by the following pseudo-code in Listing 7. It consists of three nested loops that iterate through the input buffer. The function implements zero-padding by adding zeros to the first row, first column, and last column of the input buffer. For all other positions, the function reads values from the input stream and fills the buffer accordingly as seen in Figure 58. The optimization directive *#HSL PIPELINE* is used in the depth channel inner loop so that the synthesized tool will aim to insert one value into the buffer for every clock cycle.

Listing 7: Initialize Buffer

```
for (x = 0; x < kernel_size; x++):
 for (y = 0; y < input_size+1; y++):
  for (chn = 0; chn < depth; chn++):
   #pragma HLS PIPELINE
   if (x < 1):
        pixel_val = 0;
   else if (y < 1):
        pixel_val = 0;
   else:
        input_stream >> pixel_val;
    if (y > kernel_size-1):
        if (y == input_size):
            input_buffer[x][y-kernel_size+1][chn] = 0;
        input_buffer[x][y-kernel_size][chn] = pixel_val;
    if (y < kernel_size):
        window[x][y][chn] = pixel_val;
```

The following pseudo-code describes the algorithm of the "update input buffer" function in Listing 8 and Listing 9. It consists of three nested loops that iterate through the convolutional window, shifting all the values to the left. Additionally, the two lower values of the first column are saved in a temporal array.

Listing 8: Update Window

```
temporal[kernel_size-1][depth];
pixel_val = 0;
for (x = 0; x < kernel_size; x++)
 for (y = 0; y < kernel_size; y++)
  for (chn = 0; chn < depth; chn++)
   #pragma HLS PIPELINE
   if ((y == 0) and (x > 0)):
        temporal[x-1][chn] = window[x][y][chn];
   if (y < kernel_size-1)
        window[x][y][chn] = window[x][y+1][chn];
   else
        window[x][y][chn] = input_buffer[x][0][0];
```

Another set of nested loops repeat the same shift-left task for the rest of the input buffer. On the last column, the temporal values that were saved earlier are copied back into the first two rows of the last column in the input buffer, following the pattern shown in Figure 59. If the window is positioned in the first column, last column, or last row of the input buffer, the function accounts for zero-padding values. Otherwise, the function retrieves the next value from the input stream. It is worth noting that all of the control flags, including "last column," "first column," "last row," and "max pooling," are evaluated and updated in the layer function that calls the update function.

Listing 9: Update Buffer

```
for (x = 0; x < kernel_size; x++):
 for (y = 0; y < input_size-kernel_size+2; y++):
  for (chn = 0; chn < depth; chn++):
  #pragma HLS PIPELINE
  if (y < input_size-kernel_size+1):
        input_buffer[x][y][chn] = input_buffer[x][y+1][chn];
  else:
        if (x < kernel_size-1):
            input_buffer[x][y][chn] = temporal[x][chn];
        else:
            if (last_row_flag != 1):
                if (last_col_flag == 1 or first_col_flag == 1):
                     pixel_val = 0;
                else:
                    input_stream >> pixel_val;
                input_buffer[x][y][chn] = pixel_val;
            else:
                input_buffer[x][y][chn] = 0;
```

3.5.5 2D Convolutional Layer

The 2D convolutional layer follows the logic explained in detail in Section 2.4.3, and is implemented in hardware as shown in Figure 60. The work done in [47], [69] and [70] helps as reference to further understand the logic behind the implementation of the convolutional layers in C++. The design starts by initializing the zero-padding input buffer using the first values from the input stream. Then, the convolution process begins between the convolution window and the kernel filters. The convolution operation is carried out as a multiply-accumulate operation (MAC), as explained in the corresponding chapter. To this result, the biases for each filter are added. The resulting value is then passed through a rectified linear unit (ReLU) activation function, which sets all negative values to zero. This process is repeated for each filter based on the output depth size, while the result per filter is sent to the output stream. The input buffer and kernel window values are updated to include new data from the input stream after every kernel filter has been convoluted. This process is repeated until the last value from the input stream has been introduced to the kernel window and processed.

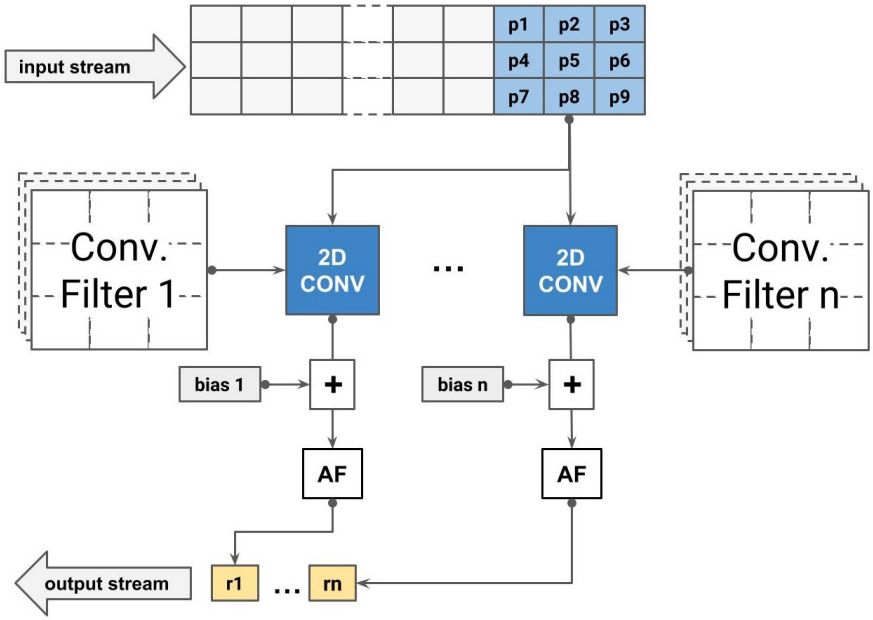

Figure 60: Overview of hardware implementation of 2D Convolution Layer.

The following pseudo-code in Listing 10 describes how the convolution operation is performed using an input buffer and a set of predefined filters. The first two loops are used to slide the input tensor across the filters and will be found in every other convolutional layer. After initializing the first values of the convolutional window and input buffer, the three last nested loops perform the convolution operation for each filter, using the optimization directive *#HSL PIPELINE* to specify that the convolution operations for each n filter should be executed concurrently in hardware. After the convolution operation is complete, the respective bias value is added and the result is passed through the activation function (ReLU), the final result is written to two output streams. Finally, the function

in-charge or updating the buffer and window values is called, which simulates the concept of the window sliding one stride, ready to repeat the process with new values in the next iteration. It is essential to note that certain layers require two output streams: one to proceed to the subsequent sequential layer and the other to function as the skip connection. Since streams can only be consumed once, it is imperative to have two streams in such cases. The GitHub repository contains the actual C++ implementation, which also includes a different 2D convolutional layer which is tailored for the last layer, where a 1x1 kernel is utilized. This implies that the convolution operation only requires the values of a single pixel. Despite this variation, the overall procedure remains largely unchanged.

Listing 10: 2D Convolution - Kernel 3x3

```
input_buffer[kernel_size][input_size-kernel_size+2][input_depth];
window[kernel_size][kernel_size][input_depth];

init_buffer_and_window();

for (x = 0; x < input_size; x++):
 for (y = 0; y < input_size; y++):
  for (filter = 0; filter < output_depth; filter++):
    #pragma HLS PIPELINE
    window_conv_result = 0;
    for (win_chn = 0; win_chn < input_depth; win_chn++):
     for (win_x = 0; win_x < kernel_size; win_x++):
      for (win_y = 0; win_y < kernel_size; win_y++):
        #pragma HLS PIPELINE
        window_conv_result += window[win_x][win_y][win_chn] *
                              weight_filt[win_x][win_y][win_chn][filter];

    window_conv_result += bias[filter];
    window_conv_result = relu(window_conv_result)

    output_stream_1 << window_conv_result;
    output_stream_2 << window_conv_result;

  update_buffer_and_window();
```

3.5.6 Depth-wise Separable 2D Convolutional Layer

The Depth-wise separable 2D convolution layer follows the logic explained in detail in Section 2.4.4, and is implemented in hardware as illustrated in Figure 61. Like the original convolution, it requires buffering the input data to fill a convolutional window, using the same buffer structure and functions. However, this convolution layer works with two types of filters: the depth-wise filter and the point-wise filter.

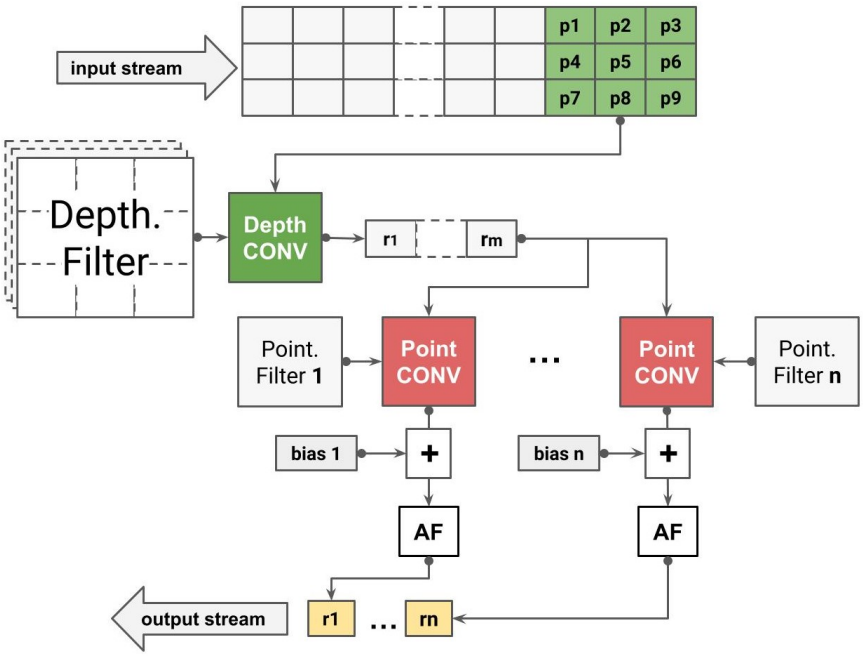

Figure 61: Overview of hardware implementation of Depth-wise Separable 2D Convolution Layer.

Similar to the original convolution, the depth-wise convolution window and depth-wise filter are of the same dimensions, but the MAC operation outputs one value per input channel, which is stored in a depth-wise vector following the pseudo-code seen in Listing 11. Once the depth-wise convolution is completed and the depth-wise vector is filled with the same number of values as the number of input channels, the point-wise convolution begins.

Listing 11: Depth-wise convolution

```
for (win_chn = 0; win_chn < input_depth; win_chn++):
  depthwise_res = 0;
  #pragma HLS PIPELINE
  for (win_x = 0; win_x < kernel_size; win_x++):
    for (win_y = 0; win_y < kernel_size; win_y++):
    depthwise_res += weight_depth_filt[win_x][win_y][win_chn] *
                     window[win_x][win_y][win_chn];

  depthwise_vector[win_chn] = depthwise_res;
```

The next convolutional step is a MAC operation between each point-wise filter and the respective depth-wise vector as described in the pseudo-code seen in Listing 12. The output of the point-wise convolution is added with the respective bias and then passed through a rectified linear unit (ReLU) activation function. The final result is then sent to the output stream. Similar to the normal convolution, the input buffer and kernel window values are updated to include new data from the input stream after every kernel filter has passed through the depth-wise and point-wise operations. This process is repeated until the last value from the input stream has been introduced and processed using the convolutional window.

Listing 12: Point-wise convolution

```
for (filter = 0; filter < output_depth; filter++):
    pointwise_res = 0;
    for (win_chn = 0; win_chn < input_depth; win_chn++):
        #pragma HLS PIPELINE
        pointwise_res += depthwise_vector[win_chn] *
                         weight_point_filt[win_chn][filter];

     pointwise_res += bias[filter];
     pointwise_res = relu(pointwise_res)
     output_stream << pointwise_res;
```

3.5.7 Transposed 2D Convolutional Layer

The Transpose 2D convolutional layer follows the logic explained in detail in Section 2.4.5, and its hardware implementation resembles that of Figure 62. The work done in [71] and [72] works as an initial reference to understand the logic behind the implementation of this layer in C++. This particular type of convolution differs from others in that it does not require buffering of the input data, as it operates on a single pixel value (including all of its input channels) to produce an output array with expanded dimensions. However, buffering the output of each transpose convolution remains necessary, as the intermediate results are required for the subsequent iterations. To address this, a buffer is defined and initialized with the bias values of the kernel filters. In the implementation, this buffer is referred to as the "transpose buffer". Similar to other convolution processes, a MAC operation is performed between the pixel depth values and the kernel filters for all filters that contribute to the output definition.

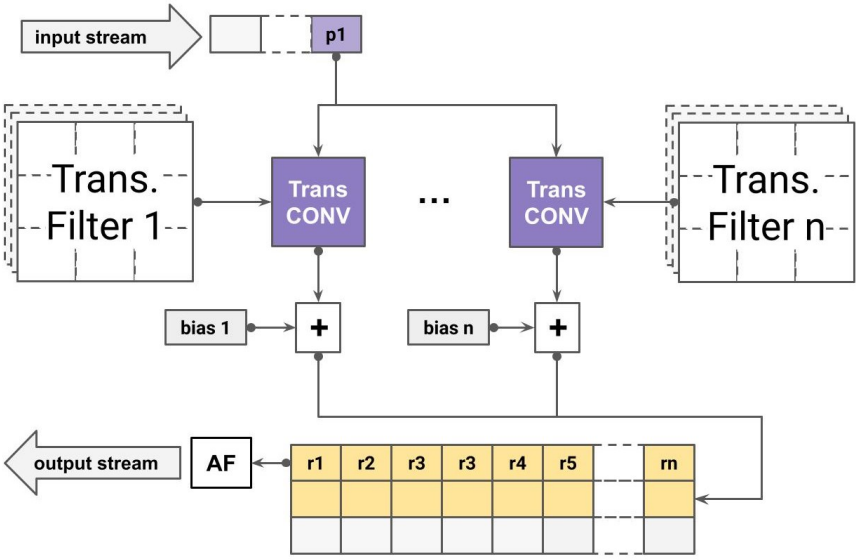

Figure 62: Overview of hardware implementation of Transposed 2D Convolution Layer.

However, the result of the MAC operation between the pixel depth values and the input depth values of the filter is stored temporarily in the "transpose buffer". In Figure 63, you can observe how the output values from previous convolutions of pixel channels are accumulated in the buffer by sliding right with a stride of two, effectively doubling the dimension of the input tensor along the "y" axis (columns). The described logic is evident in the pseudo-code provided in Listing 13.

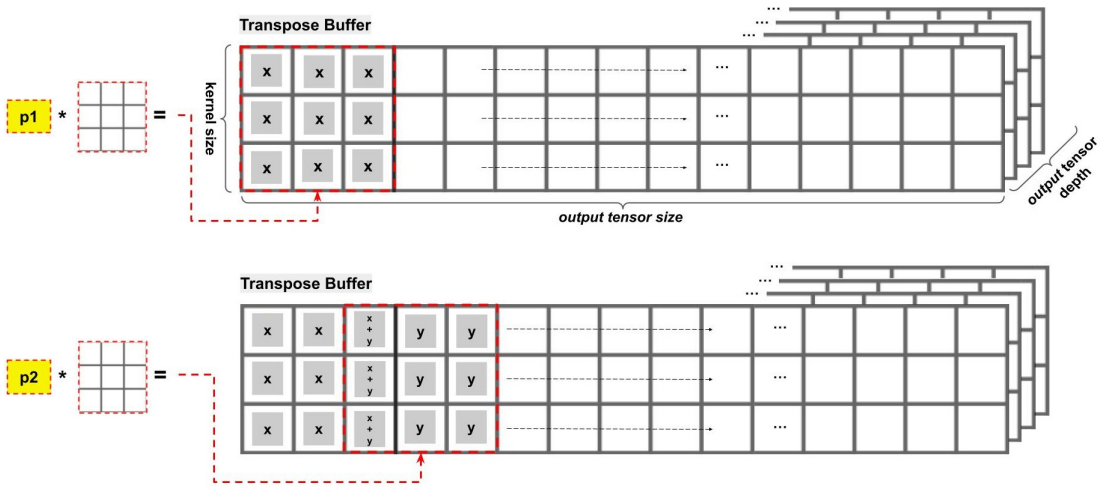

Figure 63: Example of how the transposed buffer is used to accumulate intermediate results.

Listing 13: Transposed 2D Convolution - Filling Transpose Buffer

```
fill_pixel_chn_values(pixel_vec);

for (filter = 0; filter < output_depth; filter++):
  for (win_x = 0; win_x < kernel_size; win_x++):
    for (win_y = 0; win_y < kernel_size; win_y++):
      conv_res = 0;
      #pragma HLS PIPELINE
      for (int win_chn = 0; win_chn < input_depth; win_chn++):
        #pragma HLS PIPELINE
        conv_res += pixel_vec[win_chn] *
                    weight_filt[win_x][win_y][filter][win_chn];
      tran_buff[win_x][y+win_y][filter] += conv_res;
```

Once the transposed buffer is fully filled, it is ready to output its values through the output stream. However, only the first two rows can be outputted at this stage, as the last row in the buffer is still needed to be accumulated with the results from the next iteration. It is important to note that before the values are sent to the output, they undergo a ReLU activation function, which eliminates any negative values. Additionally, the transpose buffer needs to be updated for the next iteration, as depicted in Figure 64. In this process, the last row is shifted up by two strides to become the first row, effectively doubling the dimension of the input tensor along the "x" axis (rows). Meanwhile, the middle and last rows are set back to the bias value to prepare for the next iteration. The described logic is evident in the pseudo-code provided in Listing 14. Same as before, the optimization directive *#HSL PIPELINE* is used in both iterative sections at the depth and filter inner loop respectively.

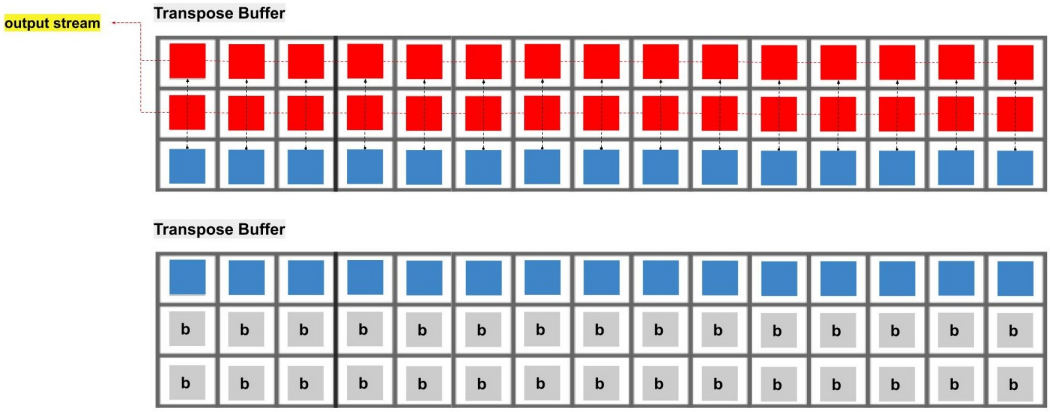

Figure 64: Example of how the transposed buffer outputs and updates its values.

Listing 14: Transposed 2D Convolution - Writing Output

```
for (int tran_x = 0; tran_x < kernel_size; tran_x++) {
  for (int tran_y = 0; tran_y < output_size; tran_y++) {
    for (int tran_f = 0; tran_f < output_depth; tran_f++) {
      #pragma HLS PIPELINE
      if (tran_x == 0):
        if (tran_buff[tran_x][tran_y][tran_f] < 0):
          output_stream << 0;
        else:
          output_stream << tran_buff[tran_x][tran_y][tran_f];
          tran_buff[tran_x][tran_y][tran_f] =
          tran_buff[tran_x+stride][tran_y][tran_f];
        else if (tran_x == 1):
          if (tran_buff[tran_x][tran_y][tran_f] < 0):
            output_stream << 0;
          else:
            output_stream << tran_buff[tran_x][tran_y][tran_f];
            tran_buff[tran_x][tran_y][tran_f] = bias[tran_f];
        else:
          tran_buff[tran_x][tran_y][tran_f] = bias[tran_f];
```

3.5.8 2D Max-Pooling Layer

The 2D Max-pooling layer follows the logic explained in detail in Section 2.4.6, and its hardware implementation is shown in Figure 65. Similar to the convolution layers, this implementation also requires buffering the input data before proceeding with the max-pooling operation. However, it computes a maximum value search operation instead of a multiply and accumulate operation. In addition, the output depth will remain the same as the input depth, resulting in writing one maximum value to the output stream per input channel.

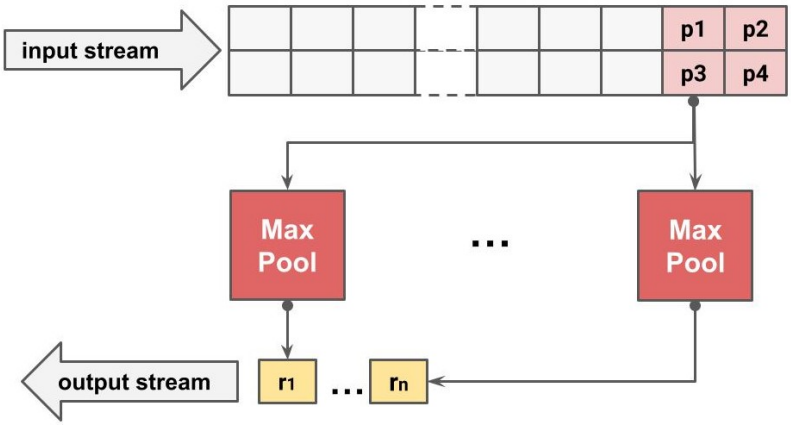

Figure 65: Overview of hardware implementation of 2D Max-Pooling Layer.

The input buffer used for max pooling operations is illustrated in Figure 66. The buffer is initialized with a window size of 2x2. When updating the input buffer, values are shifted twice (stride=2) every time a new window is required, which reduces the dimension by two on the "y" axis. Important to mention that for this type of buffer, a new input stream value is added to the lower corner of the window. Once the entire input tensor length has been processed, the buffer is filled again with the next new values. In contrast to 2D Convolution and 2D Depthwise Separable convolution, there is no need to reuse values from the buffer, because the window is of size 2x2 and its values are supposed to be shifted with a stride of 2 through the "x" dimension, effectively reducing its dimension by half.

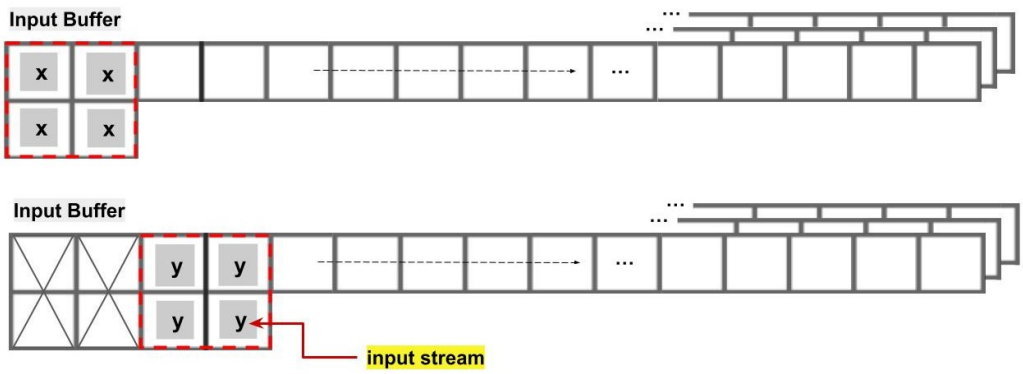

Figure 66: Example of how the input buffer updates its values.

The described logic is implemented in the pseudo-code provided in Listing 15. The structure closely resembles that of a regular convolutional layer, with the addition of a conditional statement. This conditional statement ensures that the maximum value search is executed only after the buffer has shifted by two units (due to a stride of 2) along the y width dimension. In this implementation, an output is written for every channel m during each clock cycle, as indicated by the optimization directive *#HSL PIPELINE*. Then, the buffer is updated by shifting the values to the left. Once all the maximum operations have been performed on the width dimension, the buffer is initialized again with the subsequent input values from the stream.

Listing 15: 2D Max-Pooling

```
input_buffer[pool_size][input_size-pool_size][depth];
window[pool_size][pool_size][depth];

init_buffer_and_window_maxpool();

for (x = 0; x < (input_size/2); x++):
  for (y = 0; y < input_size; y++):
    if (y%pool_size == 0):
      for (win_chn = 0; win_chn < depth; win_chn++):
        #pragma HLS PIPELINE
        maxpool_val = 0;
        for (win_x = 0; win_x < pool_size; win_x++):
          for (win_y = 0; win_y < pool_size; win_y++):
            if (maxpool_val < window[win_x][win_y][win_chn]):
                maxpool_val = window[win_x][win_y][win_chn];

        output_stream << maxpool_val;

    update_buffer_and_window_maxpool();
    if (x != ((input_size/2)-1)):
        init_buffer_and_window_maxpool();
```

3.5.9 Adding Layer

The Adding layer follows the logic described in Section 2.4.7 and its hardware implementation is shown in Figure 67. The logic is simple and requires two input streams. The input channel values from both input streams are added together, element-wise, and then passed through the activation function (ReLU). The resulting values are transferred to the output stream, resulting in the same output depth as the input depth.

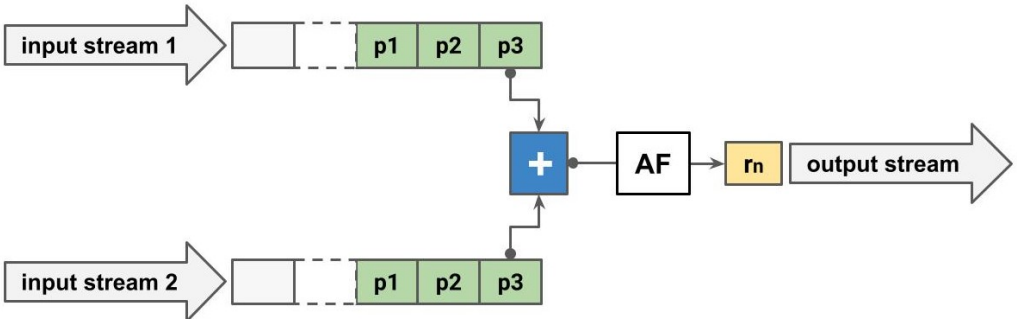

Figure 67: Overview of hardware implementation of Adding Layer.

The provided logic is implemented in the pseudo-code provided in Listing 16. The first two loops are necessary for iterating over the input tensor values, while the last nested loop fetches all the depth values of the corresponding pixels and performs the addition operation. As a result of the optimization directive *#HSL PIPELINE*, the tool will try to produce one result at every clock cycle.

Listing 16: Adding Layer

```
for (x = 0; x < input_size; x++):
  for (y = 0; y < input_size; y++):
    for (chn = 0; chn < depth; chn++):
      #pragma HLS PIPELINE
      in_stream_1 >> pixel_val_1;
      in_stream_2 >> pixel_val_2;
      adding_result = pixel_val_1 + pixel_val_2;
      adding_result = relu(adding_result)
      output_stream << adding_result;
```

3.6 Synthesis and Validation with HLS

Through Vitis HLS, it is possible to facilitate the debugging and validation process of the neural network implemented in C++. A test bench is utilized to ensure the correct functional behavior of the network. The test bench follows a straightforward logic as depicted in Figure 68. Initially, a series of images are read from text files, where each file contains the pixel channel values in a flattened format. These values are then normalized by dividing them by 255, which corresponds to the maximum RGB intensity value, as required by the neural network. Subsequently, the normalized values are reorganized into a three-dimensional array with dimensions of 255x255x3. The FlareNet function is then called to perform inference on this input. The resulting output is written into a text file and immediately compared with the corresponding golden reference. If the values from the output result match the golden reference exactly, the evaluation proceeds to the next test image. This process continues until either all the test images are successfully evaluated or a discrepancy is found between the output and golden reference results.

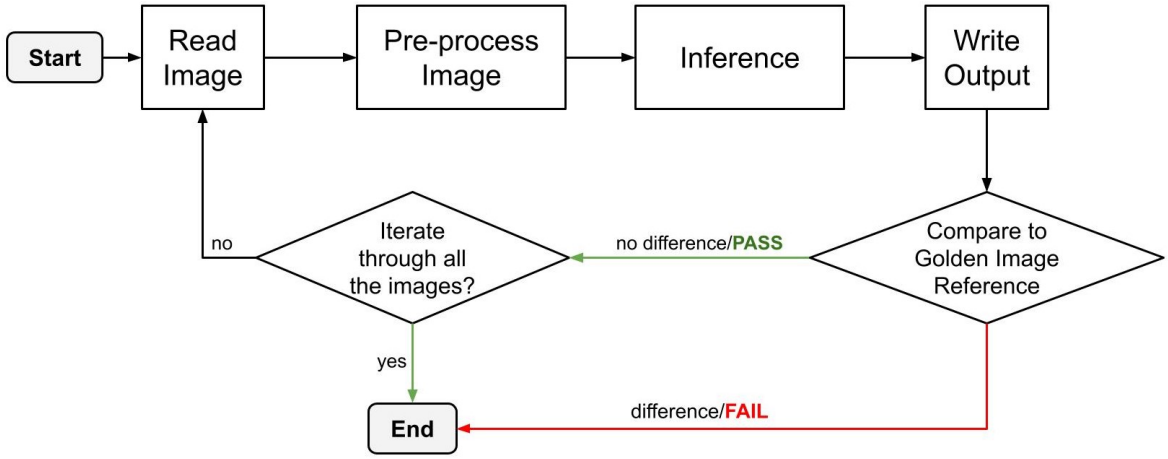

Figure 68: Logic behind test bench verification process.

After ensuring the functionality of the C++ implementation, the synthesis process is performed using VITIS. During synthesis, certain parameters and constraints need to be specified. First, a clock frequency constraint is set, which in this case is configured to 300 MHz. This constraint helps optimize the design for the target frequency. Additionally, the target FPGA board is selected for synthesis. In this scenario, the chosen FPGA board is the Zeus Zynq UltraScale [55] (target device: XQZU11EG-FFRC1760-2-i) as it is a industrial graded device used in automotive and artificial intelligence applications. These specifications guide the synthesis tool to generate the optimized hardware implementation for the specified FPGA target.

After the synthesis process, the VITIS tool generates an RTL (Register Transfer Level) design along with a comprehensive report that provides important performance metrics which will be presented and discussed in Section 4.5. These metrics include latency, estimated clock frequency, and utilization of hardware resources such as Flip-Flops (FF), Digital Signal Processing units (DSP), Look-up Tables (LUT), among others. This in-

formation helps developers assess whether the design fits within the available resources of the FPGA and identify potential areas for performance improvement. To validate the functional behavior of the generated RTL design, Co-Simulation is performed using HLS. This approach utilizes the same test bench that was used for software validation (C++ implementation). However, instead of executing inference on the software model, inference is carried out using the RTL design. Input values are passed to the RTL design, and the output is compared against the golden reference as it is done during the software validation. This process ensures that the RTL implementation produces the same results as the C++ implementation, providing confidence in the correctness of the hardware design.

3.7 Inference in GPU

To evaluate the performance of the trained model on an AI accelerator, such as a GPU, two devices are used: a laptop (Sword 15 A11UE [73], equipped with a Core i7 processor and a NVIDIA RTX3060 GPU) and the Jetson Nano GPU from NVIDIA [52]. To run inference on both devices, an application is built using C++, OpenCV, and ONNX-runtime. The integration of the deep learning network in a C++ application is possible through the use of a model converter such as "Open Neural Network Exchange" (ONNX [74]). ONNX is an open-source ecosystem that allows representing machine learning models with a standardized set of operators across multiple frameworks, providing access to hardware optimizations while maximizing performance. In the case of running the model on NVIDIA GPUs, ONNX utilizes CUDA for optimizations. The behavior of this application and how the different components and tasks interact with each other are presented in a general view on Figure 69. Visual Studio 2022 is used to develop the proposed application.

The following sections will provide further details on important aspects to consider for the implementation, along with code snippets of crucial parts. For a complete implementation, including detailed comments and instructions on how to build and run the application in the Jetson Nano using CMAKE, refer to the GitHub repository of the project in Appendix 7.1.

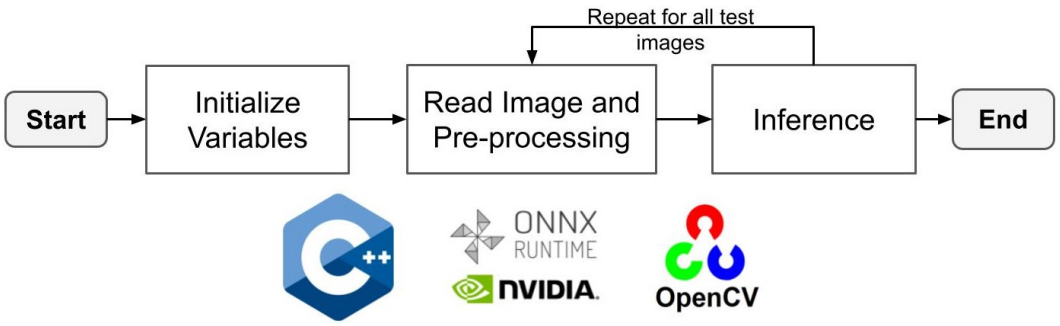

Figure 69: C++ inference application overview.

Model Conversion

To use the deep learning model trained using TensorFlow, first it has to be converted to an ONNX extension. This is done in Listing 17.

Listing 17: Model Conversion

```
model = tf.keras.models.load_model("/PATH/FlareNet.h5",
                                   custom_objects={'SSIMLoss':SSIM})
format = (tf.TensorSpec((None,256,256,3),tf.float32,
                        name="input_image"),)
onnx_model,_=tf2onnx.convert.from_keras(model,input_signature=format,
                                        opset=12)
onnx.save(onnx_model, "/PATH/FlareNet.onnx")
```

Variable Initialization

To run inference, an environment must be first instantiated. This environment is the main entry point of the ONNX Runtime to create a so-called session for inference. In addition, the memory space for the input and output tensors following the ONNX format are generated based on the input and output image dimensions used to train the network. The model will run inference using the address space of the input and output tensors as reference, expecting to find the input image on the input tensor so it could leave the inference result on the output tensor. Finally, the inference session is instantiated in the environment and the ONNX extension model is loaded. The most important parts of this process can be seen in Listing 18.

Listing 18: Variable Initialization - ONNX

```
Ort::Env env;
Ort::RunOptions runOptions;
Ort::Session session();

//Define and initialize the Input and Output Tensors.
const array<int64_t,4> input_shape = {1,256,256,3};
const array<int64_t,4> output_shape = {1,256,256,3};
//Define input and output vector-flatten size.
const int output_dim = 3*255*255;
const int input_dim = 3*255*255;
float* input_img = new float[input_dim];
float* inference_img = new float[output_dim];

auto input_tensor =
    Ort::Value::CreateTensor<float>(input_img, input,
    input_dim, input_shape.data(), input_shape.size());
auto output_tensor =
    Ort::Value::CreateTensor<float>(memory_info, inference_img,
    output_dim, output_shape.data(), output_shape.size());

//Create ONNX Session with FlareNet model.
session = Ort::Session(env, model_directory, Ort::SessionOptions{});
```

Image Pre-processing

Test images are conditioned using OpenCV to fulfill the input data shape and pixel value required by the FlareNet model. This process is shown in Listing 19 and follows the next steps: i) resizing image to 256x256, ii) flattening the image to a one dimensional array as required by ONNX input tensor, and iii) normalizing the pixel values between 0 and 1 by dividing the RGB channel values by 255.

Listing 19: Image Pre-processing

```
cv::Mat image = imread("/PATH/test_flared_img_x.jpg");
//Resize the image to the input dimension of the FlareNet.
resize(image, image, Size(256, 256));
//Reshape the image to 1D vector.
image = image.reshape(1, 1);
//Normailze number to between 0 and 1 and convert to vector.
vector<float> image_vec;
image.convertTo(image_vec, CV_32FC3, 1./255);
```

Inference

During inference, the model expects the following relevant inputs as seen in Listing 20: i) The input layer name of the model, which identifies where the input tensor is located, ii) The address pointing to the memory space of the input tensor, where the input image is expected to be, iii) The output layer name of the model, which identifies where the output tensor should store the inference result, iv) The address pointing to the memory space of the output tensor, where the resulting image will be stored. As illustrated in Figure 69, the application iterates through 1000 images and calculates the average execution time for the inference process.

Listing 20: Run Inference

```
//Copy image data to tensor input array.
copy(image_vec.begin(), image_vec.end(), input);
//Run Inference.
session.Run(runOptions, input_names.data(), &input_tensor,
            1, output_names.data(), &output_tensor, 1);
```

4 Results and Discussion

This section provides detailed insights into the implementation results, including the training outcomes of the deep learning model, the impact of quantization effects, and the results achieved during inference on both synthetic and real-life flared images. Additionally, the chapter explores the outcome of HLS synthesis and investigates how different optimization directives influence the overall performance of the design. Furthermore, it evaluates the deployment performance of the neural network solution on both mid-end and low-end devices with GPU accelerators.

4.1 Neural Network Training

This section presents the details of the training conditions found to be the best for both of the deep learning architectures.

4.1.1 FlareNet with Transfer Learning

The FlareNet-TL model, introduced in Section 3.3.3, was trained with a total of 23,644 instances, each consisting of a pair of images (input and target). In addition, 3178 images are used for validation during the training process. The hyper-parameters found to be the most suitable ones during training are the following:

- **Epochs:** 100 epochs with a regularization technique called Early Stopping setup with a patience of 15 epochs. During training, the neural network stopped improving after 46 epochs.
- **Loss function:** A functioned based on the structural similarity (SSMI) metric was selected as the loss function (1 - SSMI) as it shows better performance at focusing on reducing the flare and restoring the color contrast of the image while avoiding artifacts in the process.
- **Batch:** A batch size of 8 instances is selected as it reduces the amount of memory required to compute each training step of an epoch.
- **Optimizer:** Nesterov Adam (Nadam).
- **Kernel Initializer:** He Normal.
- **Learning rate:** An initial learning rate of 1e-4 was selected. This small learning rate is better suited to fine tuning in transfer learning. In addition, the learning rate is reduced automatically by a factor of 0.1 every 8 epochs if there is no evident loss reduction when evaluating the model with the validation dataset.

The test dataset comprises of 3178 instances and is used to assess how well the model is able to generalize to unseen data. The Structure Similarity Index (SSIM) is used to evaluate how similar are the predicted images when compared to the ground-truth (images without flare). In addition, the SSIM is also calculated between the input image (with flare) and the ground-truth, to build the baseline from where improvements are expected. Due to the fact that the only difference is the flare in between the three images (input, ground-truth and prediction), if the SSIM value between the prediction and ground-truth is higher than the SSIM value between the input and ground-truth, it is possible to conclude that the flare artifacts are being attenuated. Overall, the results in Table 2 shows that the predicted images are more similar to the ground-truth than the input images (with flare), evidencing an improvement in the image quality.

Table 2: Inference Accuracy

	Input vs Ground-truth	**Pred. vs Ground-truth**
SSIM	0.772	0.888

Figure 70 shows how the validation loss is closely decreasing at a similar ratio as the training loss, which is a good indicator of how well the model is learning, while not over-fitting the training dataset.

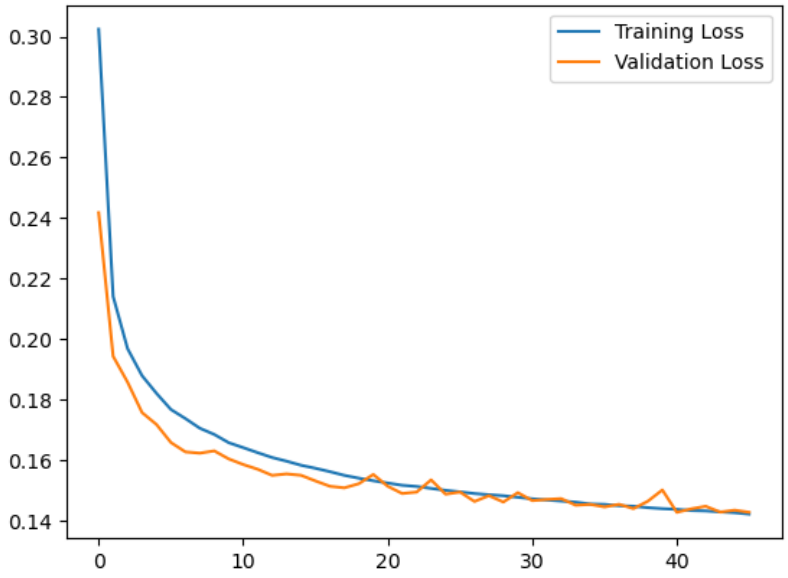

Figure 70: Learning curve of best FlareNet-TL model.

In order to enhance the model's validation, a new dataset comprising 25,425 synthetic test images is generated. In addition, two more metrics are used to evaluate how close the output predicted images are compared to the ground-truth. MAE and MSE values are considerably lower on the predicted image when compared to the input image, using the ground-truth as reference. The results presented in Table 3 exhibit similar results as those observed in the previous evaluation using the smaller test dataset. This consistency

demonstrates the model's ability to generalize well to unseen data. However, it is important to note that the current dataset consists solely of synthetic images. To ensure a comprehensive evaluation, real-world images containing actual flare artifacts should also be considered in the assessment process.

Table 3: Inference Accuracy

	Input vs Ground-truth	**Pred. vs Ground-truth**
SSIM	0.803	0.902
MAE	0.105	0.057
MSE	0.028	0.008

4.1.2 FlareNet

A simpler version of the previous model without transfer learning, named FlareNet-simple, also introduced in Section 3.3.3, was trained with the same dataset of 23,644 instances, along with 3178 images used for validation. The hyper-parameters found to be the most suitable ones during training are the following:

- **Epochs:** 200 epochs with a regularization technique called Early Stopping setup with a patience of 15 epochs.
- **Loss function:** The same functioned based on the SSMI metric was selected as the loss function (1 - SSMI).
- **Batch:** A batch size of 12 was selected.
- **Optimizer:** Nesterov Adam (Nadam).
- **Kernel Initializer:** He Normal.
- **Learning rate:** An initial learning rate of 1e-3 was selected. This small learning rate is better suited to start training a neural network without trained weights. In addition, the learning rate is reduced automatically by a factor of 0.1 every 10 epochs if there is no evident loss reduction when evaluating the model with the validation dataset.

The test dataset consisted of the same 3178 instances and the SSIM metric is also used to evaluate how similar are the predicted images when compared to the ground-truth (images without flare). Similar results, shown in Table 4, confirm also that the predicted images are more similar to the ground-truth than the input images, evidencing an improvement in the image quality. However, as expected, the previous model performs slightly better due to the transfer learning component and the overall higher complexity of the model structure.

Table 4: Inference Accuracy

	Input vs Ground-truth	Pred. vs Ground-truth
SSIM	0.772	0.866

Figure 71 shows how the validation loss is closely decreasing at a similar ratio as the training loss, which is a good indicator of how well the model is learning, while not over-fitting the training dataset.

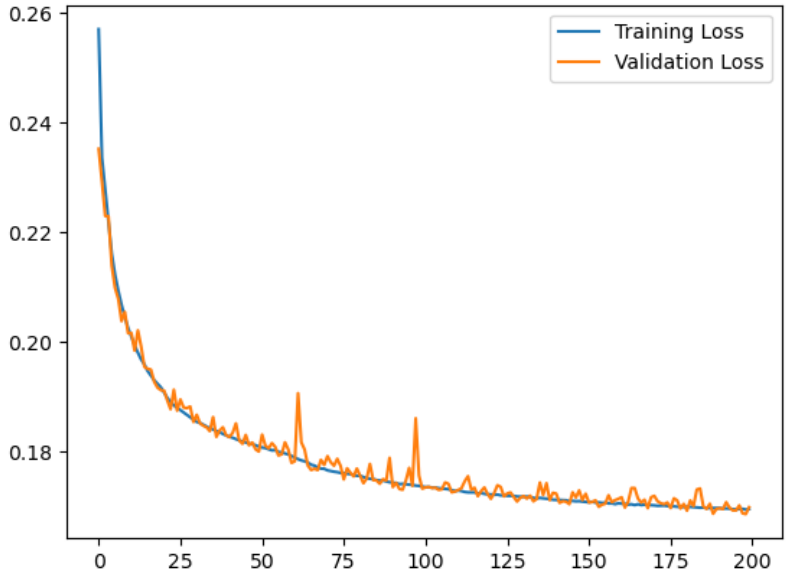

Figure 71: Learning curve of best FlareNet-simple model.

The same large test dataset, consisting of 25,425 synthetic test images, is utilized to validate the performance accuracy observed in the FlareNet-simple model. The results, as shown in Table 5, demonstrate a consistent performance compared to the smaller dataset, however inferior to the transfer learning-based model.

Table 5: Inference Accuracy

	Input vs Ground-truth	Pred. vs Ground-truth
SSIM	0.803	0.881
MAE	0.105	0.073
MSE	0.028	0.011

4.1.3 Quantization-aware

The previous model (FlareNet-simple) was quantized to assess the impact of such optimization techniques on the accuracy and size of the model. The model was trained with the same training dataset as before, with the following hyper-parameters:

- **Epochs:** only five epochs as the model is already trained and it requires a few epochs to be trained with the entire dataset during aware quantization.
- **Loss function:** The same functioned based on the structural similarity (SSMI) metric was selected as the loss function (1 - SSMI).
- **Batch:** The same batch size of 12 was used to follow a similar training process during quantization.
- **Learning rate:** A small learning rate of 1e-4 was used because the neural network is already trained.

After applying quantization-aware training, the model is assessed using Tensorflow on the same small test dataset consisting of 3178 instances, which was previously used to evaluate the two preceding models. The outcomes displayed in Table 6 reveal a slight reduction in accuracy performance compared to the previous result. This outcome is anticipated since it represents a trade-off due to the model's weight being reduced to approximately 30% of its initial size, as illustrated in Table 7.

Table 6: Inference Accuracy

	Input vs Ground-truth	**Pred. vs Ground-truth**
SSIM	0.772	0.850

Table 7: Model Size (Mb)

	Model Size (Mb)
Original (float32)	0.363
Quantized (int8)	0.1134

Despite demonstrating good performance with int8 quantized weights, it was not possible to obtain all the necessary weights from the TensorFlowLite model using the Netron App [66]. Specifically, the bias weights for the Transpose 2D Convolution were not retrievable, making it impossible to utilize them in the subsequent designs. Nevertheless, these results highlight the potential benefits of employing a quantized model for this application, and further efforts should be undertaken to ensure proper exportation of the weights.

4.2 Model Inference

The FlareNet models are performing two tasks: i) attenuating the flare, and ii) restoring the RGB values of the image. To accomplish the first objective, the network's encoder section is responsible for extracting high-level features that are related to the location and intensity of the flare. On the other hand, the decoder section then uses this information to reconstruct a flare-free image that has the same resolution as the input image. As for the second objective, the network must learn how to restore the RGB values of the entire image, as the process of attenuating the flare can lead to color distortions in the non-flare regions of the image. To address this issue, the network is trained to minimize the difference between the restored image and the original image in terms of structural similarity, as measured by the SSIM function. This helps ensure that the network produces visually pleasing and natural-looking images after flare attenuation. The above predictions using synthetic images done by the FlareNet-TL show both of these objectives being fulfilled. Inference examples for the same images processed using the FlareNet-simple model demonstrate a similar effect in attenuating the flare artifact. These examples can be found in the last Section 7.4 of the Appendix .

4.2.1 Synthetic Test Images

Figure 72: left: input image, middle: ground-truth, right: FlareNet-TL prediction.

Figure 73: left: input image, middle: ground-truth, right: FlareNet-TL prediction.

Figure 74: left: input image, middle: ground-truth, right: FlareNet-TL prediction.

Figure 75: left: input image, middle: ground-truth, right: FlareNet-TL prediction.

Figure 76: left: input image, middle: ground-truth, right: FlareNet-TL prediction.

Figure 77: left: input image, middle: ground-truth, right: FlareNet-TL prediction.

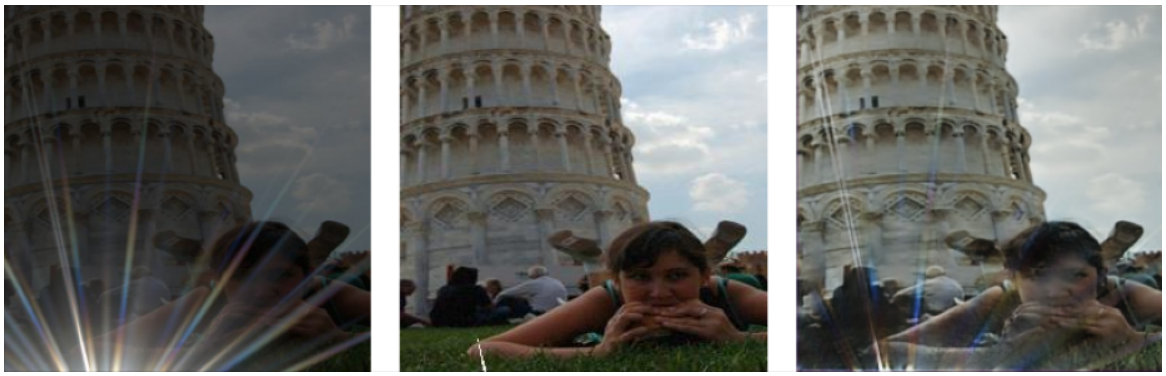

Figure 78: left: input image, middle: ground-truth, right: FlareNet-TL prediction.

Figure 79: left: input image, middle: ground-truth, right: FlareNet-TL prediction.

Figure 80: left: input image, middle: ground-truth, right: FlareNet-TL prediction.

Figure 81: left: input image, middle: ground-truth, right: FlareNet-TL prediction.

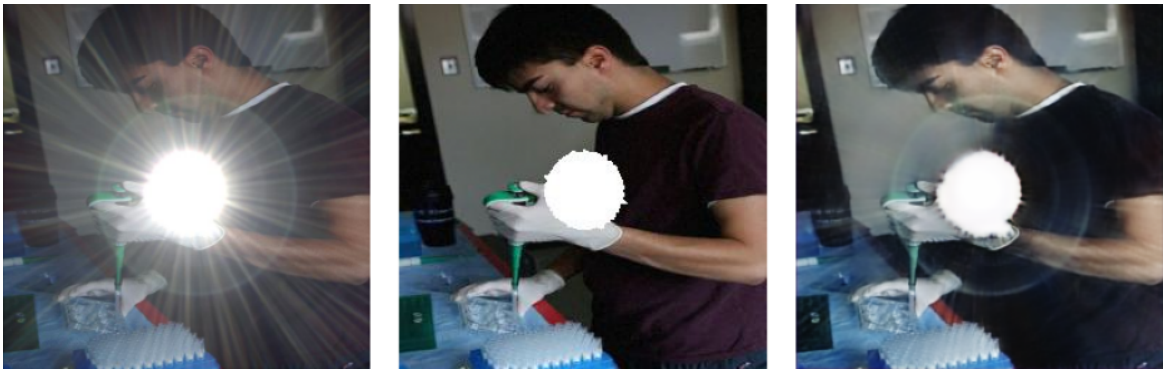

Figure 82: left: input image, middle: ground-truth, right: FlareNet-TL prediction.

Figure 83: left: input image, middle: ground-truth, right: FlareNet-TL prediction.

Figure 84: left: input image, middle: ground-truth, right: FlareNet-TL prediction.

Figure 85: left: input image, middle: ground-truth, right: FlareNet-TL prediction.

4.2.2 Inference on Real images

To assess the generalization capability of the FlareNet models to real-life cases of lens flare artifacts, a collection of real images containing these effects was assembled. It is important to note that since there is no ground-truth available for comparison, there is no objective metric to quantitatively measure the extent of attenuation achieved by the model as it was provided in Section 4.1. However, a visual inspection of the following images reveals a noticeable reduction in the intensity of the flares, indicating that the model successfully attenuates them to some degree. To ensure a diverse and comprehensive test, the images were captured using different cameras, demonstrating that the model can effectively attenuate flare artifacts regardless of the camera system used. It is important to note that although the flares in the real images are originating from the sun as the primary light source, this serves as a proof-of-concept. It is crucial to further evaluate the model's performance on different types of flares originating from various light sources. Figures 86 to 89 (taken by Xiaomi, Samsung, and Iphone cameras) demonstrate the superior performance of FlareNet-TL in attenuating flare while effectively avoiding the generation of additional artifacts in the image. In comparison, FlareNet-Simple occasionally produces black spots around the light source. These findings reinforce the results obtained from the synthetic dataset in Section 4.1, indicating that FlareNet-TL exhibits better generalization capabilities when encountering new data.

Figure 86: left: input, middle: FlareNet-TL prediction, right: FlareNet-simple prediction.

Figure 87: left: input, middle: FlareNet-TL prediction, right: FlareNet-simple prediction.

Figure 88: left: input, middle: FlareNet-TL prediction, right: FlareNet-simple prediction.

Figure 89: left: input, middle: FlareNet-TL prediction, right: FlareNet-simple prediction.

During the evaluation process, images from [28] were also utilized. Notably, in Figures 91, 92, and 93, it is evident that the FlareNet model effectively improves the contrast of RGB colors of the input images, however displaying minimal reduction in the flare artifacts. Instead, in cases where increasing RGB color contrast is not necessary, as observed in Figure 90, the model seems to focus more on attenuating the flare artifacts.

However, it is important to note that the model encounters additional challenges when dealing with reflections (shown as the colorful arch-shaped artifacts), resulting in some undesired distortion, as depicted in Figures 94 and 95. Addressing these specific challenges would require further improvements. One potential solution is to incorporate a more diverse dataset that includes this type of flare artifacts. Additionally, exploring a more complex U-Net-based architecture at the cost of increased computational resources could also be worth considering.

Figure 90: Examples of FlareNet-TL inference using real-life images from [28].

Figure 91: Examples of FlareNet-TL inference using real-life images from [28].

Figure 92: Examples of FlareNet-TL inference using real-life images from [28].

Figure 93: Examples of FlareNet-TL inference using real-life images from [28].

Figure 94: Examples of FlareNet-TL inference using real-life images from [28].

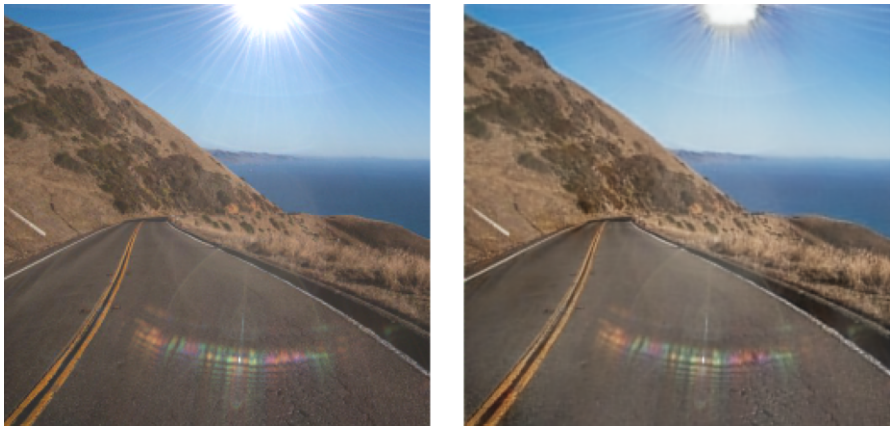

Figure 95: Examples of FlareNet-TL inference using real-life images from [28].

It is worth mentioning that the current state-of-the-art model for flare attenuation [28] utilizes the original U-Net architecture, as defined in [40], with 23 convolutional layers and nearly 30 million learnable parameters. In addition, [28] also introduces a different model with transfer learning with several layers from VGG19 as backbone, adding up to around the same size of the U-Net architecture. Finally, [38] works with a modified U-Net architecture adding a deeper layer, increasing its original size and complexity. It is evident that these models have not been designed for deployment on memory constraint devices such as embedded systems. In comparison, the FlareNet-TL model employs roughly less than 0.5% of the parameters found in the state-of-the-art flare attenuation model.

Despite its smaller size, FlareNet-TL demonstrates promising results, as evident from previous examples and the observed increase in image quality during evaluation with the synthetic test dataset using the SSIM metric. These results indicate the potential of the FlareNet architecture for efficient and effective image restoration with reduced computational requirements.

Furthermore, the images (Figure 96 to Figure 102) presented below provide visual comparisons between the predictions made by FlareNet-TL (middle image) and the state-of-the-art flare attenuation model (right image) from [28]. As expected, due to its higher complexity, the predictions made by the state-of-the-art model are highly superior, nearly eliminating any signs of flare. However, FlareNet still strives to blend the flare artifacts into the rest of the image and improve color contrast without introducing artifacts, especially for pixels that are further away from the light source.

It is worth noting that even the state-of-the-art model encounters difficulties in attenuating reflection flares, as illustrated in Figure 102. However, it does manage to correctly attenuate the remaining flare. In contrast, the FlareNet model, as mentioned earlier, exhibits poor performance when attempting to attenuate this type of artifact. This disparity in performance can be attributed to the larger capacity of the state-of-the-art model, which allows it to learn more complex features and thus better handle such challenges than its counterpart, which is less than 0.5% of its size.

Figure 96: left: input, middle: FlareNet-TL prediction, right: state-of-the-art model [28].

Figure 97: left: input, middle: FlareNet-TL prediction, right: state-of-the-art model [28].

Figure 98: left: input, middle: FlareNet-TL prediction, right: state-of-the-art model [28].

Figure 99: left: input, middle: FlareNet-TL prediction, right: state-of-the-art model [28].

Figure 100: left: input, middle: FlareNet-TL prediction, right: state-of-the-art model[28].

Figure 101: left: input, middle: FlareNet-TL prediction, right: state-of-the-art model[28].

Figure 102: left: input, middle: FlareNet-TL prediction, right: state-of-the-art model[28].

4.3 Neural Network Training - Highlights

In this section, the most relevant considerations found during training to have a positive impact on the overall result, are explained.

4.3.1 Advantage of using Transfer Learning

Results have demonstrated that the transfer learning-based FlareNet surpasses the simple FlareNet model (both introduced in Section 3.3.3) in attenuating flares as seen in Table 8. This outcome is expected since transfer learning can extract more meaningful features in the encoder section, leveraging the pre-trained weights on a larger image dataset. In addition, the transfer learning based model requires approximately one quarter of the amount of epochs required by its counterpart to reach an acceptable SSIM result, as its decoder side has already been trained and its weights are already adjusted to extract important features from input images. However, it is important to recall that the transfer learning-based FlareNet has a deeper architecture as mentioned in Section 2.4.1, which can be more challenging for implementing in hardware. One sample result can be observed in Figure 103, illustrating the advantage of the transfer learning-based model in reducing flares while effectively restoring the original contrast and RGB color values of the image. Importantly, the transfer learning approach avoids introducing any noticeable artifacts.

Table 8: Transfer Learning vs Simple version of FlareNet model.

	Ground-truth	**FlareNet-TL**	**FlareNet-simple**
SSIM	0.803	0.902	0.881
MAE	0.105	0.057	0.073
MSE	0.028	0.008	0.011

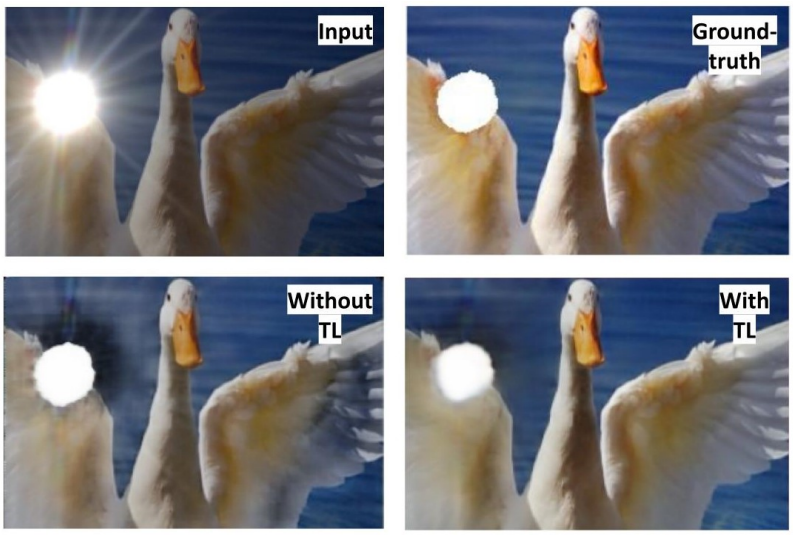

Figure 103: Inference using FlareNet model versions with and without transfer learning.

4.3.2 Impact of using Skip-connections

During the process of reducing the network size and the number of trainable parameters, various configurations inspired by the U-Net architecture were tested. The objective was not only to find an architecture suitable for attenuating the flare but also to minimize the resource requirements, particularly if they did not significantly impact the results. One of the experiments involved removing the skip connections, introduced in Section 2.4.7, to evaluate their impact on the overall outcome. As depicted in Figure 104, the absence of skip connections had a noticeable detrimental effect on the restored image quality. Skip connections facilitate the propagation of both high-level and low-level features throughout the network. In the absence of skip connections, information can become distorted or diluted as it passes through multiple deeper layers, resulting in a loss of fine-grained details.

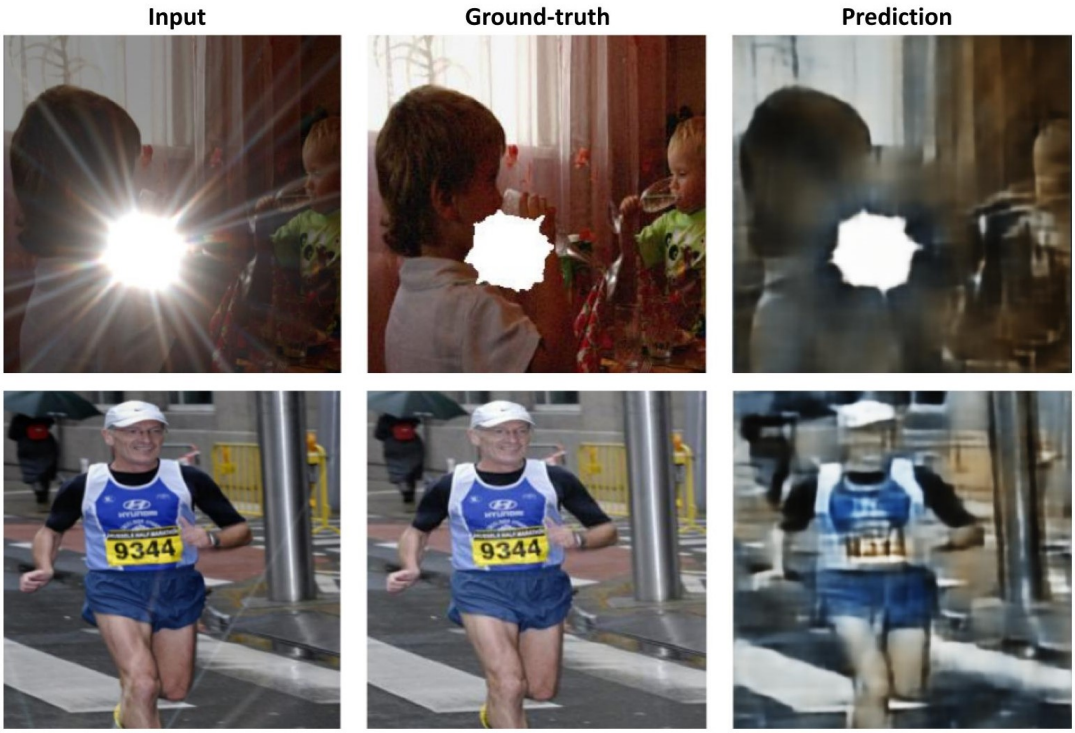

Figure 104: Examples of inference using model without skip-connections.

4.3.3 Reduce number of Learning Parameters

Reducing the number of model parameters is crucial for achieving a lightweight model that can be efficiently accelerated in hardware. However, this reduction must be carefully balanced with the trade-off between accuracy and model complexity. Here are several approaches that were employed to achieve parameter reduction:

- Depth-wise separable CNN: Instead of using traditional 2D convolutional layers, separable convolutional layers introduced in Section 2.4.4 were employed. These convolutions split the standard convolution into two separate operations: depth-wise convolution and pointwise convolution, reducing the number of parameters by factorizing the convolutional filters.
- Building own U-Net based model: To achieve a lighter and more efficient model for image restoration, a U-Net inspired architecture introduced in Section 3.3.3 was employed, with a focus on reducing the number of layers, filters, and complex layers like batch normalization. The goal was to strike a balance between model complexity and the ability to achieve high-quality image restoration.
- Using MobilNetv2 in transfer learning model: MobilNetv2 [34] was chosen as the backbone architecture due to its smaller size and efficiency compared to larger models like VGG16 [42]. By using MobilNetv2, the number of parameters is significantly reduced while still maintaining a good level of accuracy.
- Alpha Reduction 0.35: as already explained in Section 3.3.3, the alpha reduction technique involves scaling down the number of filters in the network layers of the MobilNetv2 architecture, which makes the model lighter and requires fewer parameters to train.

The impact of these parameter reduction techniques can be observed in several aspects. First, the model size is significantly reduced, making it more suitable for hardware acceleration and deployment on resource-constrained devices. Second, the training process becomes more efficient as fewer parameters need to be learned and optimized. Third, the inference speed is improved due to the reduced computational requirements of the model. Finally, there is a trade-off in terms of accuracy, where the model may achieve lower performance compared to larger and more complex models.

The exploratory phase of reducing the model complexity was done gradually. This involved training a model on the same dataset and carefully examining its performance in attenuating flare. The process was then repeated after reducing the model's complexity, allowing for an evaluation of the trade-off between flare attenuation capacity and model size. This approach is similar to the comparison conducted between the FlareNet-simple model and its transfer learning version.

4.3.4 Loss Selection

During the training process, the choice of a suitable loss function plays a crucial role in achieving desirable results in image regression tasks. In regression algorithms, commonly used metrics such as Mean Squared Error (MSE) and Mean Absolute Error (MAE) have been employed for tasks like image reconstruction and flare removal, as seen in the state-of-the-art model for flare attenuation in [38] and [28]. However, when utilizing these metrics as the loss function for training any of the FlareNet model versions, it resulted in the generation of artifacts in the predicted images, particularly in the areas surrounding the light source or near the flare.

Considering that the FlareNet models are considerably smaller compared to the original U-Net architecture used in the state-of-the-art model in [38] and [28], comprising less than 0.5% of the size, adjustments in the training process were necessary. To address this issue, the structural similarity (SSIM) function was considered as a loss metric. Compared to MAE or MSE, the incorporation of SSIM yielded significantly improved results for this smaller neural network.

Considering one of the first neural architectures trained under the same conditions for comparison purposes (dataset and hyperparameters), Table 9 shows the impact of choosing between the three image metrics, showcasing again the advantage of using SSIM. Even though the models trained with MSE and MAE still present improvements, the main disadvantage is the generation of artifacts on the inference image. In addition, the models trained with MAE and MSE are slower to train and usually get stuck in local minima, limiting the learning capacity of the network.

The SSIM metric assesses the structural similarity between two images by measuring factors like luminance, contrast, and other structural differences. By integrating SSIM into the loss function, the model is encouraged to generate images that not only minimize pixel-level differences but also preserve the structural and perceptual characteristics of the original image as a primary objective. As a result, the artifacts near the light source and around the flare are considerably reduced, leading to visually improved predictions that are free from artifacts. Figure 105 and Figure 106 provide examples that showcase the enhanced performance achieved on models trained with the SSIM loss function compared to the ones trained with MSE and MAE under same training conditions (dataset and hyperparameters). These findings confirm that for a smaller model like the versions of FlareNet, a loss function incorporating SSIM as part of its evaluation criteria is highly advantageous.

Table 9: Performance comparison between models trained with different loss functions.

	Ground-truth	**Loss SSIM**	**Loss MAE**	**Loss MSE**
SSIM	0.802	0.888	0.851	0.840
MAE	0.105	0.062	0.071	0.075
MSE	0.028	0.009	0.010	0.012

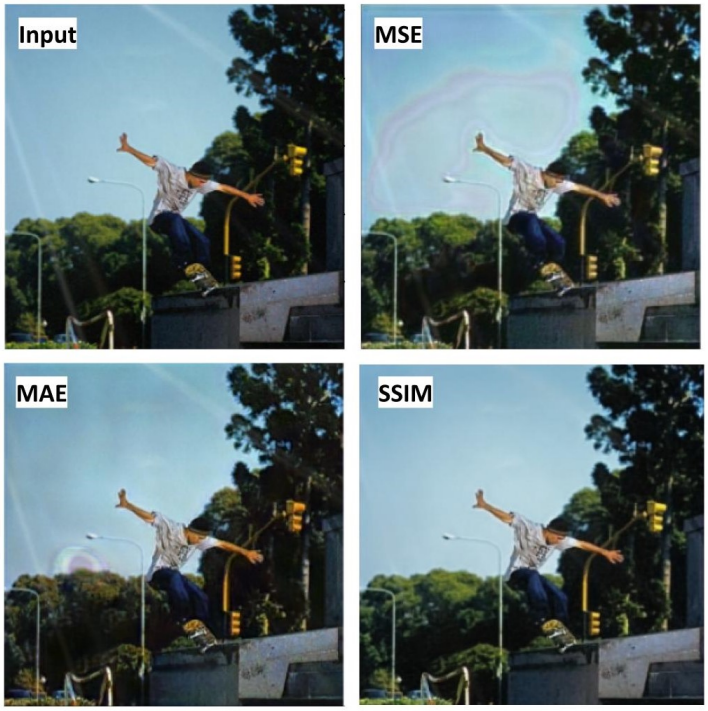

Figure 105: Examples of inference using model trained with different loss metrics. It is possible to see in the MAE and MSE images, that some "purple" circular aches are being generated on the image, which is not the case for the model using SSIM.

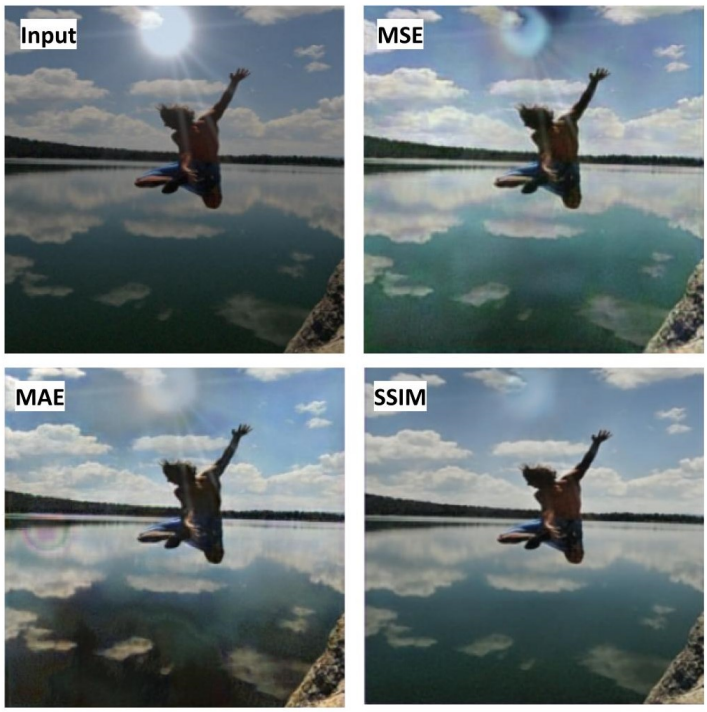

Figure 106: Examples of inference using model trained with different loss metrics.

4.3.5 Importance of Dataset Variety

It is important to emphasize the significance of selecting an appropriate dataset tailored to the specific type of flare attenuation, its color for instance. Given the model's relatively small size (less than 150,000 parameters), its ability to effectively extract features from the dataset is limited. As a result, during the initial testing phase, the focus was placed on the most common type of flare (white tone) for training the model, as depicted in the previous section showcasing the results of the test and real images.

However, to validate the model's capability to handle a wider range of flare colors, a new dataset was created. This dataset incorporated flares with different colors by randomly modifying the random white balance during the synthetic dataset generation process. The modification required to accomplish this color change is included in the respective python script as a detailed comment. The results demonstrated that even with the FlareNet-simple architecture, the model could successfully handle various flare colors, as evident from the inference results on the test images displayed in Figure 107, Figure 108 and Figure 109.

Furthermore, real-life images with actual flares were also subjected to testing, as showcased in Figure 110 and Figure 111. These tests confirmed that the model can generalize well to real-world scenarios, effectively attenuating flares present in the images.

Figure 107: Examples of FlareNet-simple inference on synthetic dataset.

Figure 108: Examples of FlareNet-simple inference on synthetic dataset.

Figure 109: Examples of FlareNet-simple inference on synthetic dataset.

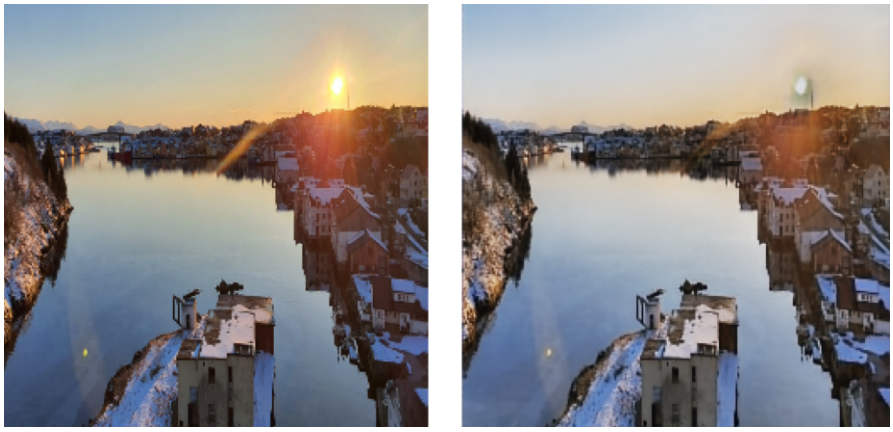

Figure 110: Examples of FlareNet-simple inference on real-life images.

Figure 111: Examples of FlareNet-simple inference on real-life images.

4.4 Impact of flare attenuation in other algorithms

As mentioned earlier, lens flare can introduce unwanted artifacts into images, thereby reducing their quality and adversely affecting algorithms that rely on artifact-free images. This section will show two examples of the potential implications of flare attenuation achieved by the deep learning model on other algorithms that utilize the image as input.

One such algorithm is an edge detector, as depicted in Figure 112. The processed image from the FlareNet model allows for better feature extraction, particularly in the lower left corner, while eliminating edge detections caused by flare artifacts in the upper part of the image. Although further studies are required for a comprehensive analysis of the impact, it is evident that an edge detector benefits from using the predicted image generated by models like FlareNet.

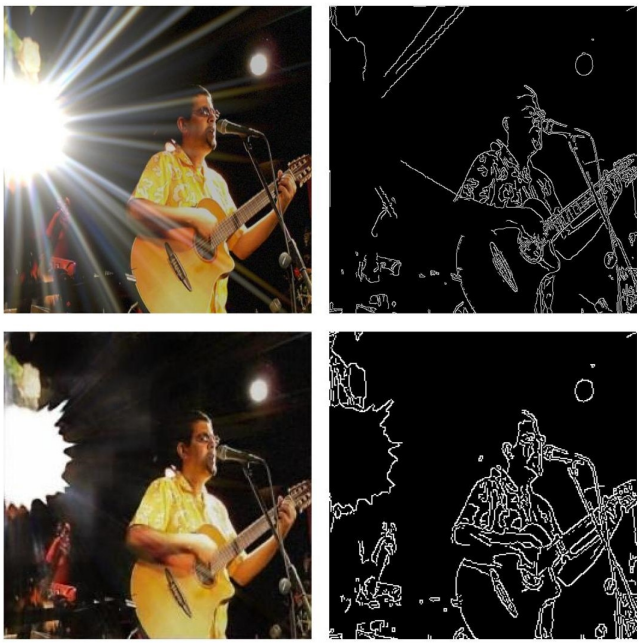

Figure 112: Example of using Edge Detector after flare attenuation.

Let's consider a more complex algorithm, such as a deep learning object detection model: YOLOv3 [75]. In the predicted image with attenuated flare (Figure 114), the confidence of the object detector in detecting a human is higher (98% compared to 85%) than in the image without flare attenuation (Figure 113). Furthermore, the unprocessed image exhibits two incorrect detections (car and bicycle), whereas such errors are not present in the predicted image.

Figure 113: Example of using YOLOv3 Object Detection model before flare attenuation.

Figure 114: Example of using YOLOv3 Object Detection model after flare attenuation.

These observations suggest that the flare attenuation performed by the FlareNet model can have positive effects on the performance of other algorithms, such as edge detection and object detection. However, further research is necessary to objectively evaluate and quantify the overall impact of these improvements in different scenarios and with a considerable amount of images.

4.5 Hardware Synthesis

This section presents the findings of the hardware synthesis of the FlareNet-simple model using High-Level Synthesis (HLS) and VITIS. The main objectives are to analyze the performance of the synthesized solution and assess the impact of different optimization directives on key metrics such as latency and resource utilization (area). This analysis provides an estimated idea of the required hardware resources as well as the estimated performance, based on VITIS HLS simulation and synthesis tool, which can be improved for a future implementation in an FPGA. While the analysis provides valuable insights, these results are considered as a starting point for refining the design and making necessary adjustments for a practical FPGA deployment. Throughout the following sections, the target FPGA is the Zeus Zynq UltraScale (target device: XQZU11EG-FFRC1760-2-i) [55] and the clock frequency for synthesis is set to 300 MHz.

For this proof-of-concept, the FlareNet-simple model is chosen due to its more compact architecture compared to the transfer learning variant. It contains fewer skip connections and very few convolutional layers as already explained in Section 3.3.3. To evaluate the runtime performance of the FlareNet implementation in C++, a series of 1000 executions were conducted on the development device [73]. The average time taken to complete one inference was found to be 60.44 sec, with little differences between observed execution times. It is important to note that this type of software execution might not provide the most accurate reference, as various factors can influence the results, such as concurrent tasks running on the operating system, the clock frequency of the device, among other potential variables. Despite these limitations, this initial software-based latency estimation serves as a starting point for understanding the performance characteristics of the FlareNet model implemented in C++, and how much this can be enhanced by implementing the model as a digital circuit.

4.5.1 Sequential Implementation

An initial implementation was synthesized disabling all optimization directives. Consequently, all optimization techniques such as dataflow, pipeline, loop unrolling, and array partitioning were disabled. This resulted in the HLS tool generating a sequential design, where tasks within layers and functions are executed sequentially, one after the other. Although this approach allows for resource re-utilization, leading to lower resource consumption/area (as indicated in Table 11), it also increases the latency (as observed in Table 10). It is particularly noticeable that the sequential implementation requires a minimal number of DSPs. However, the estimated latency for this implementation is approximately as high as 18,428 ms. It is worth clarifying that these are estimated values provided by the VITIS HSL synthesis tool after generating the RTL design.

Table 10: Latency Analysis - No optimizations - Floating Point

	HW Latency (cycles)	HW Latency (ms)
Min	4889194250	16134
Max	5584202570	18428

Table 11: Utilization Analysis - No optimizations - Floating Point

	BRAM 18K	DSP	FF	LUT	URAM
Total	31	5	10962	26507	0
Available	1200	2928	597120	298560	80
Utilization (%)	31	0.002	1	8	0

In the previous implementation, the input, output, and weights were represented using floating-point values. However, utilizing a fixed-point representation as discussed in Section 3.5.3, offers significant advantages in terms of efficiency in the RTL design. Table 13 demonstrates this by showing that the fixed-point implementation requires half the number of BRAMs (Block RAMs) compared to the floating-point implementation. Furthermore, the use of a more constrained data type leads to a significant reduction in the number of clock cycles. This can be observed in Table 12, where the latency is significantly reduced compared to the floating-point implementation.

Table 12: Latency Analysis - No optimizations - Fixed Point

	HW Latency (cycles)	HW Latency (ms)
Min	1928993322	6366
Max	2604813930	8596

Table 13: Utilization Analysis - No optimizations - Fixed Point

	BRAM 18K	DSP	FF	LUT	URAM
Total	181	12	7582	18454	0
Available	1200	2928	597120	298560	80
Utilization (%)	16	0.004	1	6	0

4.5.2 Pipeline Optimization

The current implementation still follows a sequential design approach. However, by applying the pipeline optimization directive (pragmas) within the loops of each layer in the depth dimension as seen in the different Listings from 7 to 16 in Section 3.5.2, we can introduce parallel execution within iterations. This allows tasks within each loop to overlap, leading to improved performance. The pipeline directive inserts pipeline registers in the design, breaking data dependencies and enabling the execution of subsequent iterations before the previous iterations complete. This concurrent execution of independent operations helps reduce the overall execution time, as shown in Table 14. In fact, the latency is reduced by almost four times compared to the previous sequential implementation.

The pipeline directive enhances performance by increasing parallelism within the loops. However, it also requires additional resources to work concurrently, as evident in Table 15. The utilization of Flip-Flops (FF), Digital Signal Processing (DSP), and Look-Up Tables (LUT) shows a significant increase. The pipeline design requires approximately 2 times the number of FFs, around 5 times the number of DSPs, and about 1.5 times the number of LUTs compared to the original sequential design. This increase is attributed to the replication of functional units to enable concurrent usage, unlike the sequential implementation where units could be reused. Additionally, pipelining requires instantiating extra registers, known as pipeline registers, as well as new pipeline control signals. It is important to note that this optimization directive does not enable true concurrent execution of tasks within layers and functions, and focuses on inner-loop concurrency. Therefore, there is still potential for improvement in the overall design by considering the possibility of concurrency between neural layers as it is known for a fact that it is not necessary for a layer to finish processing the totality of its input information before the next layer can start executing with the output values from the previous one.

Table 14: Latency Analysis - Pipeline

	HW Latency (cycles)	**HW Latency (ms)**
Min	550849346	1818
Max	730657378	2411

Table 15: Utilization Analysis - Pipeline

	BRAM 18K	**DSP**	**FF**	**LUT**	**URAM**
Total	199	55	14337	30169	0
Available	1200	2928	597120	298560	80
Utilization (%)	16	1	2	10	0

4.5.3 Dataflow and Pipeline Optimization

When both the pipeline and dataflow optimization directives are applied together, it allows for a more optimized implementation by taking advantage of both parallel execution within loops and concurrent execution of tasks within layers and functions of the neural network architecture. The optimization directive *#HSL DATAFLOW* is applied in the main function where all the layers used in the FlareNet model are instantiated. By using both directives together, the HLS tool can generate a design that benefits from both intra-loop parallelism (achieved through the pipeline directive) and inter-task parallelism (achieved through the dataflow directive). This results in reduced latency as seen in Table 16, where the latency is reduced to around one fourth, while the resources utilization remains almost the same compared to the pipeline-only implementation as seen in Table 17.

It is crucial to note that the application of this new optimization allows for concurrent execution of the neural layers. Consequently, the latency between layers is superimposed as one can start before the previous one finishes working with all the input data as already explained in Section 2.6.3 about the dataflow directive. However, if only the pipeline optimization is applied without concurrent execution between layers, the inference time will be the cumulative sum of the latency for each layer in the model as every layer has to wait to finish its computation before the next one can start.

Table 16: Latency Analysis - Dataflow and Pipeline

	HW Latency (cycles)	**HW Latency (ms)**	**Interval (cycles)**
Min	155277746	512	154750481
Max	155277746	512	154750481

Table 17: Utilization Analysis - Dataflow and Pipeline

	BRAM 18K	**DSP**	**FF**	**LUT**	**URAM**
Total	199	55	14797	29972	0
Available	1200	2928	597120	298560	80
Utilization (%)	16	1	2	10	0

4.5.4 Utilization Analysis - Layers

The implemented solution, which incorporates pipelining and dataflow optimization directives, has shown the best performance results in terms of resources and latency. The following analysis provides a detailed overview of each layer in the solution, as shown in Table 18.

Table 18: FlareNet Layers Analysis

	Lat.(cycles)	Lat.(ms)	BRAM	DSP	FF	LUT
Conv2D	154750480	511	4	28	2883	2812
2DMaxPool	136414122	450	4	0	560	2151
2DSeparable	121145271	400	8	2	1095	2746
2DMaxPool	34506186	114	6	0	747	2235
2DSeparable	37130407	123	13	2	1110	2751
2DMaxPool	6628650	21.8	6	0	751	2354
2DSeparable	9904823	32.7	15	3	1236	2907
2DMaxPool	1158122	3.8	4	0	739	2222
2DTranspose	11329093	37.4	45	1	555	1651
2DTranspose	33995157	112	37	1	564	1833
Add	196611	0.65	0	0	41	183
2DTranspose	71795301	237	32	1	572	1818
2DTranspose	106001461	350	25	1	576	1729
Add	1048579	3.5	0	0	44	188
Conv2D	3538965	11.7	0	16	1420	731

By optimizing the entire implementation using the dataflow directive, the execution of layer functions in the neural network can be parallelized, resulting in improved performance. Therefore, the absolute estimated inference time is largely affected by the layer that takes the longest to execute. In this scenario, the first 2D Convolutional layer requires 154750480 clock cycles to complete, which leaves 527266 cycles for the remaining layers to process the last pixels of the input data that the convolutional layer is processing at the end. This relationship is depicted in Figure 115.

To further understand this behavior, we can examine the Interval parameter, which represents the number of clock cycles required before a new image can be input into the first layer of the neural network. The absolute interval of the network, as shown in Table 16, is 154750481 cycles. Interestingly, this value is just one clock cycle more than the latency of the first convolutional layer. Consequently, right after the first convolutional layer finishes processing, a new image can start being processed. However, it will still take an additional 527266 cycles for the first image to complete processing, resulting in a total latency of 155277746 cycles or 512 ms, as reported.

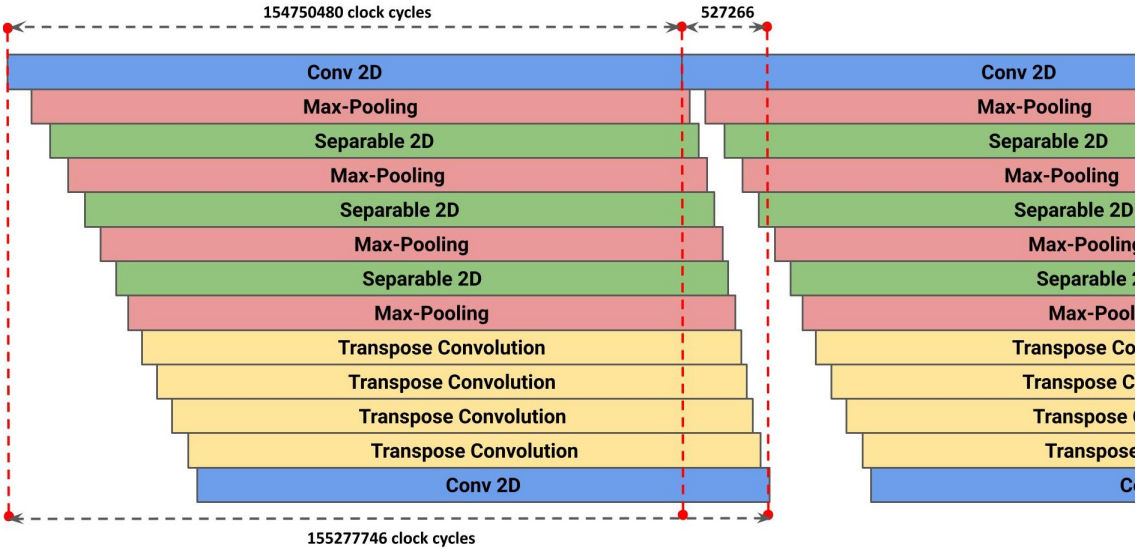

Figure 115: Example of the impact of applying dataflow directive.

On the other hand, it can be noticed that while the dimension of the input tensor decreases throughout the encoder, the latency of the layer significantly reduces. For example, the first 2D Max-pooling layer with an input dimension of 255x255 takes around 110 times more than the last 2D Max-pooling layer with an input dimension of 32x32. Additionally, on the decoder side, as the dimension increases, so does the latency. It is important to note that despite the filter dimension increasing throughout the encoder section, reaching a maximum value of 64 in the bottom layer (the last max-pooling layer), the latency still depends more on the length dimension (columns) than on the filter dimension, as this last one is pipelined.

The synthesized RTL design exhibits low resource utilization, particularly with respect to the minimal number of DSP units required. However, this suggests that certain operations are still being executed sequentially, leading to increased execution time. Unfortunately, due to time constraints, it was not possible to identify and resolve code dependencies within the buffering functions. This limitation hinders further optimization of the design, including the application of optimization directives such as *#HLS array partitioning*, as discussed in Section 2.6.3. Employing such directives could help alleviate the memory bottleneck by enabling parallel access to buffer arrays, which is currently not feasible. Additional optimization directives, such as *#HLS loop unroll*, were also evaluated in this study. However, the results showed that implementing *#HLS loop unroll* did not provide latency improvements. On the contrary, it led to increased resource utilization.

Another observation is that the latency of the 2D Transpose Layer also scales with its output dimension, despite not utilizing an input buffer like the other convolutional layers. This is because it still requires an output buffer, with size defined by the output dimension, to store temporary results while computing the actual output, thereby making the layer dependent on the increasing dimension in the decoder.

Furthermore, it is worth noting the advantages of using 2D Separable Convolutional layers instead of traditional ones in terms of DSP utilization. By separating the kernel filter into depthwise and pointwise components, the DSP requirement is significantly reduced.

4.5.5 Image Dimension Analysis

In order to assess the scalability of the design to different input dimensions, designs were synthesized using the same constraints as before, but this time considering input images of varying sizes: 64x64x3, 128x128x3, 256x256x3, and 512x512x3. As shown in Table 19, the findings align with our previous observations, indicating that the design's latency scales directly with the image size. For a small image size of 64x64x3, the design achieves an inference time of only 21.8 ms. However, for a larger image size of 512x512x3, the inference time increases significantly, reaching as high as 4 seconds. Once again, it becomes apparent that the layers that require a buffer contribute the most to the overall execution time, reaffirming the criticality of optimizing this aspect of the design. As expected, the dependency between the input image size and the number of BRAMs is evident. Since the size of the width buffer is determined by the image size, it follows that the amount of BRAMs required for it will vary accordingly.

Table 19: Image Dimension Analysis

	Latency (ms)	**BRAM**	**DSP**	**FF**	**LUT**
64x64x3	21.8	131	55	14296	29308
128x128x3	113	160	55	14554	29595
256x256x3	512	199	55	14797	29972
512x512x3	4000	271	55	14999	30295

4.6 GPU Inference

This section presents the findings of executing inference using the FlareNet-simple model on different computing devices with AI accelerators as described in Section 3.7. Two devices were considered: a medium-end and a low-end device. The purpose of this test was to evaluate the inference time on both devices. It is crucial to emphasize that the application built in Section 3.7 was utilized for this test. This application leverages the ONNX-runtime for executing inferences on the model. It is important to note that this approach differs significantly from the C++ implementation for HLS, where each layer and the overall architecture were individually implemented in C++.

The medium-end device [73] used for inference is equipped with an NVIDIA RTX3060 GPU that operates at a frequency of 900 MHz, which can be boosted up to 1425 MHz depending on the workload, and a maximum power consumption of 80 Watts. In contrast, the Jetson Nano [52] was selected as the low-end device, and its Maxwell GPU operates at 640 MHz which can also be busted but up to 921 MHz, with a maximum power consumption of 12 Watts.

By evaluating the inference execution on both devices, it is possible to gain insights into the model's performance across different hardware configurations and assess its feasibility in real-world deployment scenarios. As anticipated, the mid-end device equipped with a more advanced GPU exhibited a significantly higher throughput, surpassing the capabilities of the Jetson Nano by over 5 times, as shown in Tables 20 and 21.

Table 20: Software Latency - RTX3060 GPU

	SW Latency (ms)
Latency (mean)	31

Table 21: Software Latency - Jetson Nano GPU

	SW Latency (ms)
Latency (mean)	165

Several factors can contribute to the superior performance of the GPUs compared to a FPGA-based neural network implementation. For example, GPUs feature specialized hardware components like tensor cores that are optimized for deep learning tasks such as matrix multiplications and convolutions. Furthermore, GPUs offer a mature ecosystem with libraries like CUDA and cuDNN, which provide optimized functions for deep learning tasks that can be accessed through the use of model converters such as ONNX. These libraries leverage the parallel architecture of GPUs, offering highly efficient implementations of common deep learning operations.

It is worth noting that GPUs are designed to operate at higher clock frequencies compared to FPGAs, typically ranging from 600 MHz to 900 MHz during execution. This higher operating frequency allows GPUs to execute instructions at a faster rate, contributing to

their overall performance advantage. However, it is important to acknowledge that GPUs generally consume more power than FPGA implementations.

On the other hand, FPGA implementations require careful custom hardware design to fully exploit their parallelism potential. While high-level synthesis (HLS) tools like Vitis HLS provide a level of abstraction by allowing the use of C++ to describe functional behaviour, they still require high expertise to optimize the architecture and exploit parallelism effectively. Failure to carefully program and leverage the available optimization directives may result in under-utilization of the FPGA's potential parallelism. Therefore, optimizing the entire architecture and exploring other optimization techniques are necessary, demanding additional time.

There are however, new platforms available such as VITIS AI [76] help to deploy a neural network directly from a deep learning framework such as TensorFlow into an FPGA without having to functionally describe the algorithm using HLS, optimizing the solution to maximize inference speed. The library uses optimized deep-learning processor units (DPU) core along with a specialized software stack to accelerate several DNN models as it can be seen in [77].

5 Conclusions

The primary objective of this study was to evaluate an approach for reducing lens flare artifacts through image post-processing techniques. A comprehensive review of traditional methods was conducted, mainly focusing on image reconstruction algorithms based on deconvolution. These approaches emphasized the importance of modeling stray light using the Point Spread Function (PSF) specific to a camera system. However, necessary equipment to measure the PSF was unavailable for this work, and attempts to simulate a PSF for evaluating the performance of a deconvolution algorithm resulted in unrealistic lens flare artifacts. Additionally, even with a known PSF, literature studies focused on static camera setups, such as the ones found in microscopy or healthcare applications, indicating limited potential for PSF-based lens flare reduction in industrial settings. This limitation arises from factors such as the dynamic nature of the PSF in real-world scenarios which requires frequent characterization and update, and the uncertainty in execution time due to the iterative nature of some deconvolution algorithms. As a result, this particular image post-processing technique was not further investigated.

On the other hand, artificial intelligence, particularly deep learning neural networks, have shown promising results in flare attenuation despite limited studies. A synthetic flare dataset was generated, and an iterative training process was employed to develop the first compact and lightweight U-Net based model for lens flare reduction, named FlareNet with and without a transfer learning component. Through a systematic approach, the aim was to identify the optimal set of hyper-parameters and architecture that could effectively minimize flare artifacts while maintaining computational efficiency.

Noteworthy insights were gained during the process, including the importance of skip connections, the benefits of transfer learning, and the selection of an appropriate loss metric. It was found that, for a small model, using the Structural Similarity Index (SSIM) as a loss metric yielded effective results in reducing flare without introducing additional artifacts in the restored image. Both versions of the FlareNet model (with and without transfer learning) demonstrated improvement in image quality on the testing dataset, with a modest parameter count of less than 150,000 and a simple neural network architecture comprising five types of layers. However, as expected, the FlareNet version with transfer learning shows better performance at mitigating flare artifacts, but at the cost of using a more complex and deeper architecture. Moreover, a quantization-aware approach was applied to assess the impact of reducing the weight representation from float32 to int8, resulting in a lighter model while considering the trade-off in accuracy. The quantized network exhibited a marginal decrease in accuracy but reduced the model's memory weight to 30% of its initial size. Additionally, the FlareNet models demonstrated its ability to reduce flare in real-life images, indicating that it can achieve satisfactory visual results despite having less than 0.5% of the weights of state-of-the-art neural architectures.

Furthermore, as part of the proof-of-concept, the simple FlareNet model version was implemented in C++ using Vitis HLS to profile the required resources and performance when deployed as a digital circuit on an FPGA. Each of the model's five layers, along with

other necessary elements like zero-padding logic buffers, were implemented and tested. Synthesis and validation were performed using the VITIS tool, and reports were analyzed while experimenting with HLS optimization directives. Results demonstrated that for the selected FPGA and a clock frequency of 300 MHz, the inference time is approximately 512 ms (equivalent to approximately 2 frames-per-second) with minimal resource utilization well within the device's limits. Due to time constraints, limited attempts were made to further optimize the solution, but it became evident that a different approach was required for the buffer structure and related functions as they seemed to create a memory bottleneck that limits further parallelism done by the HSL tool.

Although this study serves as the first proof-of-concept to understand the resource requirements for a digital design implementation of a lens flare attenuation deep learning-based model, further work and expertise in HLS are necessary to optimize the convolutional layers, so it can be suitable for real-time applications. In addition, there is still potential for further task parallelization considering that at the moment the utilization is considerably low. Therefore, recommendations for future work to potentially enhance the performance of the entire solution are provided in the final section of this study.

Finally, the model's performance was evaluated on two GPU devices, comparing the execution time on different hardware accelerators. On a laptop with a medium-end GPU, the model achieved an inference speed of 32 frames-per-second, while on a lower-end GPU like the Jetson Nano, it achieved 6 frames-per-second. Considering that these GPUs operate at frequencies between 650 MHz and 950 MHz, lower inference times would be expected at the cost of higher power consumption.

6 Recommendations and Future Work

While a proof-of-concept for the solution has been successfully implemented within the four-month timeframe of this project, it is important to acknowledge that there are still several points that need to be addressed. These points have become apparent during the course of this work and require further attention and resolution.

(i) Considering the learning behavior of both versions of the FlareNet model (with and without transfer learning) during the training process, the validation loss steadily decreases in a similar proportion to the training loss, suggesting that the model still has the potential to learn without necessarily overfitting to the dataset. To explore this further, it is recommended to increase the number of epochs and also expand the dataset by adding more instances. Additionally, the current dataset generation process involves running a Python script to generate and store the 32,000 image instances in a folder. However, an alternative approach can be adopted by leveraging TensorFlow Data Generator [36] to automate the dataset creation during the network training. With this approach, there would be no need to store the entire dataset separately, and the network can be trained directly on the fly using the merged flared and deflared images generated in real-time. This could potentially help to handle larger amounts of images during training.

(ii) By applying quantization-aware training to the FlareNet model without transfer learning and using int8 weights instead of float32, a significant reduction in memory utilization of approximately 30% was achieved, while maintaining a relatively small drop in image quality reconstruction. However, during the process of extracting the int8 weights from the TensorFlow Lite model using the Netron app [66], it was discovered that not all int8 weights, specifically the bias weights of the transpose convolution filter, could be successfully extracted. Considering the potential benefits of using int8 weights, such as decreased resource utilization in the FPGA, it is recommended to explore alternative methods for extracting the weights from the TensorFlow Lite model. In addition, the same procedure can be applied to the transfer learning version to assess its performance.

(iii) To enhance the performance and applicability of the FlareNet architecture, it is important to enrich the dataset with various types of lens flare artifacts. Specifically, additional reflection types should be included. These types of artifacts present unique challenges that the current model has not adequately addressed.

(iv) It is important to note that the current solution has been trained using images with flare in a day-time condition. However, no specific evaluation has been conducted to assess its performance for applications that operate in low-light or nighttime conditions. To address this gap and ensure the effectiveness of the FlareNet architecture in nighttime scenarios, it would be valuable to train it using a dataset specifically designed for nighttime applications. The Flare7K dataset [39], which is specifically tailored for nighttime scenes, can serve as an excellent resource for this purpose.

(v) Considering the superior performance exhibited by the transfer-learning based FlareNet model, it is advisable to investigate its efficient implementation in hardware, leveraging the fact that most of the necessary layers have already been implemented in HSL. However, it is important to note that additional time, beyond the scope of this project, was required to construct and validate the bigger architectural model, as well as export and condition all of its weights.

(vi) During this study, various image comparison metrics have been employed to assess the effectiveness of the solution in reducing flare. However, it is crucial to acknowledge that certain industrial applications may necessitate an alternative evaluation criteria that aligns with industry standards. Therefore, further investigation is required to determine the appropriate evaluation criteria for such applications.

(vii) Although the proof-of-concept was conducted using images of size 255x255, it is crucial to assess the scalability of the solution to accommodate larger images as needed in industrial applications. Two potential options can be considered for handling larger image sizes: i) Down-sampling and up-sampling: One approach is to down-sample the larger images to a fixed size that the deep learning model can process, such as 255x255. After applying the FlareNet model to the down-sampled image, the inference result can be up-sampled to the original size required by the application. This approach allows for leveraging the existing model and framework while accommodating larger image dimensions, however the effect on image quality after down-sampling and up-sampling procedures has to be assessed. ii) Entire solution for fixed-sized images: Another option is to develop and implement the entire solution specifically for a fixed image size that aligns with the requirements of the industrial application. This approach involves retraining the neural network, potentially modifying the architecture to include deeper layers in the auto-encoder design to further reduce the dimension of the image. The C++ layers for HLS can be reused with different parameters to adapt to the new image size.

(viii) An initial evaluation has been conducted to assess how the deep learning model impacts other algorithms that utilize the inference image as input, showing positive benefits. However, it is important to note that this evaluation has been performed with a limited amount and variety of images containing flare. Expanding the evaluation dataset to include a wide range of images with varying levels of flare will provide a more comprehensive understanding of how the deep learning model interacts with different algorithms.

(ix) In relation to the HSL implementation, while synthesis reports offer a valuable estimation of performance, it is essential to deploy the solution on an FPGA for obtaining more precise results concerning utilization and, specifically, inference time. Nonetheless, before the actual deployment, additional optimization of the design is necessary to enhance its performance as already seen in synthesis.

(x) Furthermore, deploying the solution on an FPGA will enable a direct and objective comparison between the inference image produced by the hardware implementation, utilizing fixed-point representation (10 bits for integers, 8 bits for decimals), and the

image generated by the TensorFlow software framework, which employs a float32 representation. This comparison will allow for an objective assessment of the trade-off between using a fixed-point representation versus a floating point one in terms of any potential compromise in image quality after reconstruction.

(xi) During the HLS implementation phase, it became apparent that the logic used for implementing the zero-padding buffers needed modification to reduce latency and improve algorithm performance. One notable difference between the zero-padding and normal buffers (used in the max-pooling layers) is their width: the normal buffers have a width of [kernel_size-1], while the zero-padding buffer has a width of [kernel_size]. Although this design choice helps manage the insertion of zeros into the buffer, it introduces an additional line buffer that must be shifted left image_input_size * image_input_size times. Consequently, this considerably increases latency. It is worth mentioning that the normal buffer was also implemented in a similar fashion, and the latency of the layer was halved by reducing the dimension of the buffer in a later optimization step. However, due to time constraints within the project timeframe, it was not possible to finish the same modification to the zero-padding buffer. Therefore, additional work is required to optimize the zero-padding buffer.

(xii) To address the limitations related to buffer update function and improve parallelism utilization, a thorough examination of the synthesis results is crucial. This analysis will help identify additional potential parallelism opportunities and allow for the exploration of alternative optimization directives. For instance, the application of the *#HLS ARRAY PARTITIONING* optimization directive holds promise as a means to tackle the memory bottleneck hindering further parallelism implementation. Moreover, leveraging native HLS Line Buffering functions, as outlined in the VITIS User Guide [67], could offer a compelling solution. By pursuing further optimization of the design, it may be possible to handle images with larger dimensions without incurring prohibitive inference times. For instance, the current 4-second inference time for a 512x512 image could potentially be reduced to acceptable levels.

(xiii) Another approach to implementing the entire network in C++ using HLS is to leverage dataflow compilers specifically designed for deep learning inference on FPGAs. These compilers, such as VITIS AI [76], FlexCNN [77], or FINN [78], offer a streamlined process for loading models directly from deep learning frameworks like TensorFlow or model versions in ONNX format into VITIS. Although these platforms are relatively recent, they have shown promising results in efficiently and user-friendly deploying highly complex neural networks. However, while dataflow compilers may provide convenient ways to deploy complex neural networks on FPGAs, they abstract away implementation details, making it challenging for developers to have fine-grained control over understanding the design outcomes.

(xiv) Performing a comparison of the inference time between the state-of-the-art U-Net based network [28] and FlareNet would be a valuable endeavor. However, to conduct this comparison, it is necessary to train the complete U-Net architecture, as described in the paper [28], since a pre-trained model is currently unavailable.

References

[1] J.-O. Park, W.-K. Jang, S.-H. Kim, H.-S. Jang, and S.-H. Lee, "Stray light analysis of high resolution camera for a low-earth-orbit satellite," *J. Opt. Soc. Korea*, vol. 15, no. 1, pp. 52–55, Mar 2011. [Online]. Available: https://opg.optica.org/josk/abstract.cfm?URI=josk-15-1-52

[2] Synopsys, "Stray light," https://www.synopsys.com/glossary/what-is-stray-light.html, accessed: 2023-01-05.

[3] E.-V. Talvala, A. Adams, M. Horowitz, and M. Levoy, "Veiling glare in high dynamic range imaging," *ACM Trans. Graph.*, vol. 26, no. 3, p. 37–es, jul 2007. [Online]. Available: https://doi.org/10.1145/1276377.1276424

[4] L. Clermont, W. Uhring, and M. Georges, "Stray light characterization with ultrafast time-of-flight imaging," *Scientific reports*, vol. 11, no. 1, pp. 1–9, 2021.

[5] L. C. Scaduto, E. G. Carvalho, L. F. Santos, F. Yasuoka, M. A. Stefani, and J. C. Castro, "Baffle design and analysis of stray-light in multispectral camera of a brazilian satellite," *Annals of Optics, XXIX ENFMC*, 2006.

[6] L. Clermont, C. Michel, and Y. Stockman, "Stray light correction algorithm for high performance optical instruments: The case of metop-3mi," *Remote Sensing*, vol. 14, no. 6, p. 1354, 2022.

[7] J. Wei, B. Bitlis, A. Bernstein, A. de Silva, P. A. Jansson, and J. P. Allebach, "Stray light and shading reduction in digital photography: a new model and algorithm," in *Digital Photography IV*, J. M. DiCarlo and B. G. Rodricks, Eds., vol. 6817, International Society for Optics and Photonics. SPIE, 2008, p. 68170H. [Online]. Available: https://doi.org/10.1117/12.768562

[8] B. Bitlis, P. A. Jansson, and J. P. Allebach, "Parametric point spread function modeling and reduction of stray light effects in digital still cameras," in *Computational Imaging V*, C. A. Bouman, E. L. Miller, and I. Pollak, Eds., vol. 6498, International Society for Optics and Photonics. SPIE, 2007, p. 64980V. [Online]. Available: https://doi.org/10.1117/12.715101

[9] T. Bretschneider, "On the deconvolution of satellite imagery," in *IEEE International Geoscience and Remote Sensing Symposium*, vol. 4, 2002, pp. 2450–2452 vol.4.

[10] F. J. Ávila, J. Ares, M. C. Marcellán, M. V. Collados, and L. Remón, "Iterative-trained semi-blind deconvolution algorithm to compensate straylight in retinal images," *Journal of Imaging*, vol. 7, no. 4, 2021. [Online]. Available: https://www.mdpi.com/2313-433X/7/4/73

[11] M. Mitchell, "Genetic algorithms: An overview." in *Complex.*, vol. 1, no. 1. Citeseer, 1995, pp. 31–39.

[12] M. Paulinas and A. Ušinskas, "A survey of genetic algorithms applications for image enhancement and segmentation," *Information Technology and control*, vol. 36, no. 3, 2007.

[13] C. Janiesch, P. Zschech, and K. Heinrich, "Machine learning and deep learning," *Electronic Markets*, vol. 31, no. 3, pp. 685–695, 2021.

[14] A. Shrestha and A. Mahmood, "Review of deep learning algorithms and architectures," *IEEE Access*, vol. 7, pp. 53 040–53 065, 2019.

[15] S. Sharma, S. Sharma, and A. Athaiya, "Activation functions in neural networks," *Towards Data Sci*, vol. 6, no. 12, pp. 310–316, 2017.

[16] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *arXiv preprint arXiv:1511.08458*, 2015.

[17] Y. H. Liu, "Feature extraction and image recognition with convolutional neural networks," *Journal of Physics: Conference Series*, vol. 1087, no. 6, p. 062032, sep 2018. [Online]. Available: https://dx.doi.org/10.1088/1742-6596/1087/6/062032

[18] M. Boden, "A guide to recurrent neural networks and backpropagation," *the Dallas project*, vol. 2, no. 2, pp. 1–10, 2002.

[19] D. Choi, C. J. Shallue, Z. Nado, J. Lee, C. J. Maddison, and G. E. Dahl, "On empirical comparisons of optimizers for deep learning," *CoRR*, vol. abs/1910.05446, 2019. [Online]. Available: http://arxiv.org/abs/1910.05446

[20] T. M. Breuel, "The effects of hyperparameters on SGD training of neural networks," *CoRR*, vol. abs/1508.02788, 2015. [Online]. Available: http://arxiv.org/abs/1508.02788

[21] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[22] H. Gholamalinezhad and H. Khosravi, "Pooling methods in deep neural networks, a review," *CoRR*, vol. abs/2009.07485, 2020. [Online]. Available: https://arxiv.org/abs/2009.07485

[23] C. Xu and H. Wang, "Research on a convolution kernel initialization method for speeding up the convergence of cnn," *Applied Sciences*, vol. 12, no. 2, 2022. [Online]. Available: https://www.mdpi.com/2076-3417/12/2/633

[24] V. Mudeng, M. Kim, and S.-w. Choe, "Prospects of structural similarity index for medical image analysis," *Applied Sciences*, vol. 12, no. 8, 2022. [Online]. Available: https://www.mdpi.com/2076-3417/12/8/3754

[25] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *Journal of big data*, vol. 6, no. 1, pp. 1–48, 2019.

[26] L. Perez and J. Wang, "The effectiveness of data augmentation in image classification using deep learning," *CoRR*, vol. abs/1712.04621, 2017. [Online]. Available: http://arxiv.org/abs/1712.04621

[27] A. Galdran, A. Alvarez-Gila, M. I. Meyer, C. L. Saratxaga, T. Araujo, E. Garrote, G. Aresta, P. Costa, A. M. Mendonça, and A. J. C. Campilho, "Data-driven color augmentation techniques for deep skin image analysis," *CoRR*, vol. abs/1703.03702, 2017. [Online]. Available: http://arxiv.org/abs/1703.03702

[28] Y. Wu, Q. He, T. Xue, R. Garg, J. Chen, A. Veeraraghavan, and J. T. Barron, "How to train neural networks for flare removal," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 2239–2247.

[29] I. Z. Mukti and D. Biswas, "Transfer learning based plant diseases detection using resnet50," in *2019 4th International Conference on Electrical Information and Communication Technology (EICT)*, 2019, pp. 1–6.

[30] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," *CoRR*, vol. abs/1603.05279, 2016. [Online]. Available: http://arxiv.org/abs/1603.05279

[31] N. Mellempudi, A. Kundu, D. Das, D. Mudigere, and B. Kaul, "Mixed low-precision deep learning inference using dynamic fixed point," *CoRR*, vol. abs/1701.08978, 2017. [Online]. Available: http://arxiv.org/abs/1701.08978

[32] B. Rokh, A. Azarpeyvand, and A. Khanteymoori, "A comprehensive survey on model quantization for deep neural networks," *arXiv preprint arXiv:2205.07877*, 2022.

[33] R. Banner, Y. Nahshan, E. Hoffer, and D. Soudry, "ACIQ: analytical clipping for integer quantization of neural networks," *CoRR*, vol. abs/1810.05723, 2018. [Online]. Available: http://arxiv.org/abs/1810.05723

[34] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017. [Online]. Available: http://arxiv.org/abs/1704.04861

[35] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," *CoRR*, vol. abs/1506.02626, 2015. [Online]. Available: http://arxiv.org/abs/1506.02626

[36] e. a. Martín Abadi, Ashish Agarwal, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[37] T. L. Foundation, "Pytorch," https://pytorch.org/, accessed: 2023-01-15.

[38] A. V. Shoshin and E. A. Shvets, "Veiling glare removal: synthetic dataset generation, metrics and neural network architecture," vol. 45, no. 4, pp. 615–626, 2021.

[39] Y. Dai, C. Li, S. Zhou, R. Feng, and C. C. Loy, "Flare7k: A phenomenological nighttime flare removal dataset," in *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2022. [Online]. Available: https://openreview.net/forum?id=Proso5bUa

[40] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," *CoRR*, vol. abs/1505.04597, 2015. [Online]. Available: http://arxiv.org/abs/1505.04597

[41] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[42] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[43] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[44] G. Bahl, L. Daniel, M. Moretti, and F. Lafarge, "Low-power neural networks for semantic segmentation of satellite images," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) Workshops*, Oct 2019.

[45] N. Beheshti and L. Johnsson, "Squeeze u-net: A memory and energy efficient image segmentation network," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops*, 2020, pp. 364–365.

[46] S. Hong, *FPGA-Optimized Neural Network for Cloud Detection from Satellite Images*. University of California, Irvine, 2022.

[47] M. Sarg, A. H. Khalil, and H. Mostafa, "Efficient hls implementation for convolutional neural networks accelerator on an soc," in *2021 International Conference on Microelectronics (ICM)*, 2021, pp. 1–4.

[48] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," 2018.

[49] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014. [Online]. Available: http://arxiv.org/abs/1409.4842

[50] T. D. Han and T. S. Abdelrahman, "hicuda: High-level gpgpu programming," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 78–90, 2011.

[51] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel computing experiences with cuda," *IEEE Micro*, vol. 28, no. 4, pp. 13–27, 2008.

[52] NVIDIA, "Jetson nano developer kit," https://developer.nvidia.com/embedded/jetson-nano-developer-kit, accessed: 2023-04-04.

[53] A. Muthuramalingam, S. Himavathi, and E. Srinivasan, "Neural network implementation using fpga: issues and application," *International Journal of Electrical and Computer Engineering*, vol. 2, no. 12, pp. 2802–2808, 2008.

[54] J. Cong, J. Lau, G. Liu, S. Neuendorffer, P. Pan, K. Vissers, and Z. Zhang, "Fpga hls today: Successes, challenges, and opportunities," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 4, aug 2022. [Online]. Available: https://doi.org/10.1145/3530775

[55] "Reflex ces zeus zynq® ultrascale+™ mpsoc system-on-module," https://www.xilinx.com/products/boards-and-kits/1-1dar9m2.html, accessed: 2023-05-18.

[56] "Xilinx," https://www.xilinx.com/, accessed: 2023-02-12.

[57] S. Mittal, S. Gupta, and S. Dasgupta, "Fpga: An efficient and promising platform for real-time image processing applications," in *National Conference On Research and Development In Hardware Systems (CSI-RDHS)*, 2008.

[58] C. Johnston, K. Gribbon, and D. Bailey, "Implementing image processing algorithms on fpgas," in *Proceedings of the Eleventh Electronics New Zealand Conference, ENZCon'04*, 2004, pp. 118–123.

[59] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, "Are we there yet? a study on the state of high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 898–911, 2019.

[60] Y.-L. Lin, "Recent developments in high-level synthesis," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 2, no. 1, p. 2–21, jan 1997. [Online]. Available: https://doi.org/10.1145/250243.250245

[61] XILINX, "Vitis high-level synthesis user guide (ug1399)," https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/HLS-Pragmas, accessed: 2023-03-15.

[62] O. Anacona-Mosquera, J. Arias-García, D. M. Muñoz, and C. H. Llanos, "Efficient hardware implementation of the richardson-lucy algorithm for restoring motion-blurred image on reconfigurable digital system," in *2016 29th Symposium on Integrated Circuits and Systems Design (SBCCI)*, 2016, pp. 1–6.

[63] K. Avagian and M. Orlandić, "An efficient fpga implementation of richardson-lucy deconvolution algorithm for hyperspectral images," *Electronics*, vol. 10, no. 4, 2021. [Online]. Available: https://www.mdpi.com/2079-9292/10/4/504

[64] A. Pirinen and A. Toytziaridis, "Stray light compensation in optical systems," *Master's Theses in Mathematical Sciences*, 2015.

[65] P. Young, A. Lai, M. Hodosh, and J. Hockenmaier, "From image descriptions to visual denotations: New similarity metrics for semantic inference over event descriptions," *Transactions of the Association for Computational Linguistics*, vol. 2, pp. 67–78, 02 2014. [Online]. Available: https://doi.org/10.1162/tacl_a_00166

[66] "Netron app," https://netron.app/, accessed: 2023-02-18.

[67] Xilinx, "Vitis high-level synthesis user guide - ug1399 (v2021.2)," https://xilinx.eetrend.com/files/2021-11/wen_zhang_/100555486-227863-ug1399-vitis-hls.pdf, accessed: 2023-03-10.

[68] "Vitis high-level synthesis user guide - ug902 (v2016.2)," https://xilinx.eetrend.com/files/2021-11/wen_zhang_/100555486-227863-ug1399-vitis-hls.pdf, accessed: 2023-03-10.

[69] "How to implement a convolutional neural network using high level synthesis - github repository," https://www.amiq.com/consulting/2018/12/14/how-to-implement-a-convolutional-neural-network-using-high-level-synthesis/, accessed: 2023-03-10.

[70] A. Consulting, "How to implement a convolutional neural network using high level synthesis," https://github.com/amiq-consulting/CNN-using-HLS/blob/master/modules/conv/conv.cpp, accessed: 2023-03-10.

[71] S. Charalampos, "U-net neural network analysis and implementation using reconfigurable logic," in *Institutional Repository Technical University of Crete*, 2021.

[72] "Unet-fpga," https://github.com/labis7/UNET-FPGA/blob/master/Vitis_files/Tconv/main.cpp, 2021, accessed: 2023-03-16.

[73] M.-S. I. (MSI), "Sword 15 a11ue," https://www.msi.com/Laptop/Sword-15-A11UX/Specification, accessed: 2023-03-12.

[74] T. L. Foundation, "Open neural network exchange," https://onnx.ai/, 2019, accessed: 2023-03-23.

[75] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *CoRR*, vol. abs/1804.02767, 2018. [Online]. Available: http://arxiv.org/abs/1804.02767

[76] "Vitis ai," https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html, accessed: 2023-04-05.

[77] S. Basalama, A. Sohrabizadeh, J. Wang, L. Guo, and J. Cong, "Flexcnn: An end-to-end framework for composing cnn accelerators on fpga," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 16, no. 2, mar 2023. [Online]. Available: https://doi.org/10.1145/3570928

[78] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. H. W. Leong, M. Jahre, and K. A. Vissers, "FINN: A framework for fast, scalable binarized neural network inference," *CoRR*, vol. abs/1612.07119, 2016. [Online]. Available: http://arxiv.org/abs/1612.07119

[79] "Onnxruntimeconfig cmake," https://github.com/microsoft/onnxruntime/issues/3124, accessed: 2023-04-23.

[80] "How to install jetpack," https://docs.nvidia.com/jetson/jetpack/index.html, accessed: 2023-04-03.

[81] "Tensorrt execution provider," https://onnxruntime.ai/docs/execution-providers/TensorRT-ExecutionProvider.html, accessed: 2023-04-03.

[82] "Cuda execution provider," https://onnxruntime.ai/docs/execution-providers/CUDA-ExecutionProvider.html, accessed: 2023-04-03.

7 Apendix

7.1 Project Repository

This work includes a GitHub repository where all the source files and configuration scripts are included. This chapter will explain the organization of the repository.

GitHub Project Repository:
Flare Attenuation Filter

Below is an explanation of what can be found in each of the directories. It is important to mention that all of the source files include detailed comments. In addition, all of the file PATHs have to be changed depending on the system and device.

Deep Learning Model Training

(i) **Generate FlareNet Dataset:** python notebook script based on [28] used to generate the synthetic dataset by merging flare and scene images.

(ii) **Training FlareNet models:** python notebook script used to train either the transfer learning or simple version of the FlareNet model.

(iii) **Quantization aware Training:** python notebook script used to do quantization-aware training.

(iv) **Convert FlareNet to ONNX model:** python notebook script to convert a tensorflow-keras model to onnx extension.

(v) **Models:** directory that contains FlareNet models.

C++ HLS Hardware Design

(i) **Weights:** Includes the ".h" file with all the weight constant arrays extracted from the FlareNet-simple neural network.

(ii) **Main Function - FlareNet:** Includes the ".cpp" and ".h" files that contain all the layer and buffering functions required to build the neural model. In addition, it contains the function that instantiates all the layers in order to build the structure of the FlareNet-simple neural network.

(iii) **Test Bench:** Includes the ".cpp" test bech file required for validation purposes.

(iv) **Data:** directory that contains some input images in ".txt" format along with its corresponding golden reference result.

Application GPU deployment

(i) **C++ Inference Application:** Includes the "Inference.cpp" file which contains the source code for the application.

(ii) **ONNX-cmakeconfig CMAKE - Jetson Nano:** Includes the ONNX-runtime cmake configuration file, based on [79], necessary to build the C++ inference application in the Jetson Nano running Jetpack operative system.

(iii) **ONNX Model:** directory that contains FlareNet-simple model in onnx format.

Simulated PSF for Deconvolution

(i) **Genetic Algorithm:** Includes the python file with the genetic algorithm to search for a simulated PSF.

(ii) **Richardson-Lucy Algorithm:** Includes the python file with the Richardson-Lucy deconvolution algorithm used for testing and better understanding its behaviour and how it can be implemented.

(iii) **Parametric Model:** Includes the python file that implements the PSF from the work done in [64].

7.2 Configuration of Jupyter Notebook to train with GPU

The training scripts were run using the following setup:

- Windows 11
- Anaconda 3 open-source distribution platform.
- Jupyter Notebook 6.4.8.

In order to activate the GPU for training, it is necessary to first install the correct tensorflow-gpu package before installing the other required libraries. The following sequence of commands were executed in the Anacoda environment in order to run the notebook successfully.

Listing 21: Install Jetpack on Jetson

```
conda create --name tf_gpu tensorflow-gpu
conda install -c conda-forge jupyterlab
pip install opencv-python
conda install -c conda-forge matplotlib
conda install -c anaconda scikit-learn
conda install -c anaconda scikit-image
```

To run the quantization-aware training with GPU, the following sequence of commands were executed in the Anacoda environment.

Listing 22: Install Jetpack on Jetson

```
!pip install tensorflow-model-optimization
conda install -c conda-forge keras
```

7.3 Configuration and Deployment on Jetson Nano

The following configurations are done in headless mode as well as connected to a monitor. *Putty* and *FileZilla* are used for remote connection and friendly SFTP user interface between development PC (running Windows) and both edge devices.

Install Jetpack on Jetson
In case the Jetson model has no pre-installed Jetpack version, install the one that corresponds as stated in [80]. Otherwise, if the Jetson model has a preinstalled Jetpack version, check which one by executing the following command:

Listing 23: Install Jetpack on Jetson

```
sudo apt-cache show nvidia-jetpack
```

The Jetpack version used for this project is:

- Jetson Nano: 4.5

Install the ONNX Runtime Repository
Depending on the Jetpack version installed, different versions of CUDA, cuDNN and TensorRT will be also installed and available. It is not recommended to uninstall and install any of these modules independently. The procedure will vary depending on the Jetson module and Jetpack version. According to the version of these modules, follow the requirement chart in [81] and [82] to make sure there is compatibility between versions. Following the Jetpack versions used for this project, execute the next commands to install onnxruntime v1.6.0:

Jetson Nano: 4.5

Listing 24: Install the ONNX Runtime Repository on Jetson Nano

```
git clone --single-branch --recursive --branch rel-1.6.0
https://github.com/Microsoft/onnxruntime
```

Specify the CUDA compiler, or add its location to the PATH.
CMAKE can't automatically find the correct nvcc if it's not in the PATH, so assign it using the following command.

Listing 25: Export CUDA path

```
export CUDACXX="/usr/local/cuda/bin/nvcc"
```

Install the ONNX Runtime build dependencies
Install the required dependencies with the following command:

Listing 26: Install the ONNX Runtime build dependencies

```
sudo apt install -y --no-install-recommends build-essential
software-properties-common libopenblas-dev libpython3.6-dev
python3-pip python3-dev python3-setuptools python3-wheel
```

Install Cmake building from source
Cmake is needed to build ONNX Runtime. Because the minimum required version is 3.18, it is necessary to build it from source. Execute the next commands based on [**?**].

Listing 27: Install Cmake building from source

```
wget https://cmake.org/files/v3.23/cmake-3.23.0.tar.gz
tar xf cmake-3.23.0.tar.gz
cd cmake-3.23.0
sudo apt-get install libssl-dev
./bootstrap
make
sudo make install
cmake --version
```

If after cmake –version, cmake is still not found, add the installed cmake bin directory path in .bashrc:

Listing 28: Export cmake path

```
export PATH=/home/user/cmake-3.23.0/bin
```

Build the ONNX Runtime Python wheel with TensorRT support
Change directory to where the build.sh file is located and run the following command.

Listing 29: Build the ONNX Runtime Python wheel with TensorRT support

```
./build.sh --config Release --update
--build --build_wheel --skip_submodule_sync
--build_shared_lib --use_tensorrt
--cuda_home /usr/local/cuda
--cudnn_home /usr/lib/aarch64-linux-gnu
--tensorrt_home /usr/lib/aarch64-linux-gnu
```

Copy the required library and include files
Cmake looks for the libraries and includes files on predefined paths. In order to run

properly the next cmake file and CMakeList, copy the "Release" directory located inside */build/Linux* into */home/<username>/.local/lib* and copy the "include" directory into */home/<USERNAME>/.local/*.

Include the cmake file needed to locate Onnxruntime library
Create */cmake/onnxruntime* directories inside */home/<username>/.local/* and copy onnxruntime -config.cmake that can be found in the GitHub repository into */home/ <username>/.local/share/cmake/onnxruntime*.

Transfer the Application into Jetson
Copy the Application folder into a directory. The folder includes: i) CMakeList.txt needed to build the application, ii) Deep Learning Model directory from which the application will read the ONNX model, iii) source directory with the main.cpp program, and iv) include directory with the helpers.cpp, helpers.h and utils.h files.

Build Application
Create the build directory and build the application as follows.

Listing 30: Build Application

```
mkdir build && cd build
cmake ..
```

Compile Application
When the build is complete, go to the build directory and compile.

Listing 31: Compile Application

```
make
```

7.4 FlareNet-simple Inference

Figure 116: left: input image, middle: ground-truth, right: FlareNet-simple prediction.

Figure 117: left: input image, middle: ground-truth, right: FlareNet-simple prediction.

Figure 118: left: input image, middle: ground-truth, right: FlareNet-simple prediction.

Figure 119: left: input image, middle: ground-truth, right: FlareNet-simple prediction.

Figure 120: left: input image, middle: ground-truth, right: FlareNet-simple prediction.

Figure 121: left: input image, middle: ground-truth, right: FlareNet-simple prediction.

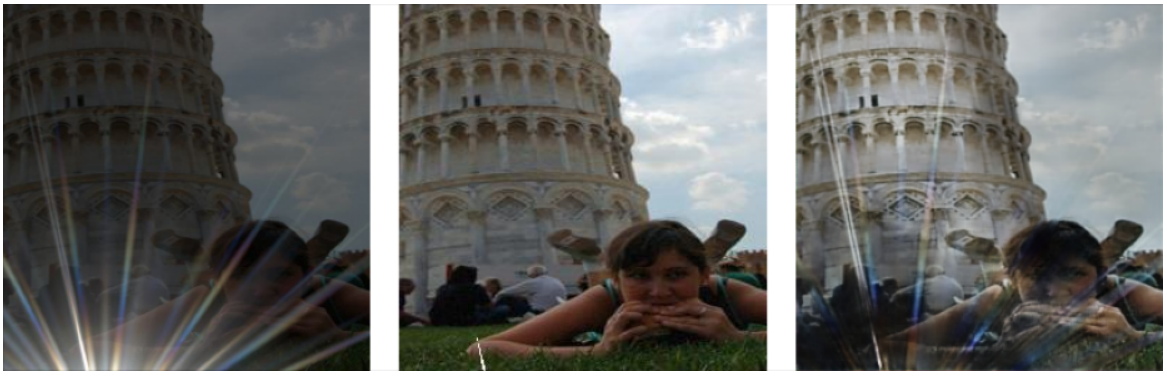

Figure 122: left: input image, middle: ground-truth, right: FlareNet-simple prediction.

Figure 123: left: input image, middle: ground-truth, right: FlareNet-simple prediction.

Figure 124: left: input image, middle: ground-truth, right: FlareNet-simple prediction.

Figure 125: left: input image, middle: ground-truth, right: FlareNet-simple prediction.

Figure 126: left: input image, middle: ground-truth, right: FlareNet-simple prediction.

Figure 127: left: input image, middle: ground-truth, right: FlareNet-simple prediction.

Figure 128: left: input image, middle: ground-truth, right: FlareNet-simple prediction.

NTNU
Norwegian University of
Science and Technology

SONY