# Accelerating Electron Diffraction Analysis with Machine Learning Inference on FPGAs

Pranav Srinivas Murali

A thesis

submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Electrical Engineering

University of Washington

2024

Committee:

Scott Hauck
Shih-Chieh Hsu

Program Authorized to Offer Degree:

Electrical and Computer Engineering

University of Washington

## Abstract

Accelerating Electron Diffraction Analysis with Machine Learning Inference on FPGAs

Pranav Srinivas Murali

Chair of the Supervisory Committee:
Scott Hauck
Department of Electrical and Computer Engineering

Reflection High-Energy Electron Diffraction (RHEED) is used to study the crystal structure and growth of thin films in material science. The RHEED systems available today take several minutes to analyze the crystal’s structure from an image captured during the crystal's growth. This thesis captures the design and deployment of a high-speed camera pipeline with a LeNet5 neural network using the HLS4ML library on an FPGA. After deployment, a latency of 450-750 us was obtained depending on the input size to the neural network. It establishes the possibility of an RHEED system that can process images from a camera up to a rate of 1000 Hz. This system would enable users to gain more insights about their experiments in real-time and allow them to modify or stop the deposition process during the crystal growth to achieve desired characteristics.

# TABLE OF CONTENTS

# ACKNOWLEDGEMENTS

This research was funded in part by National Science Foundation Grant No. PHY-2117997 and would not have been possible without the efforts and support of numerous people. First and foremost, I would like to thank my advisor Prof. Scott Hauck who has been very supportive throughout my master's degree at UW. I couldn't have asked for a better advisor who was willing to give me time and help me understand where I needed to improve. I'll always be grateful for all the insights I received over the last two years in terms of engineering, career paths, and the workings of the semiconductor industry. All of this has enabled me to think beyond and become a better engineer. I would also like to thank Prof. Shih-Chieh Hsu supported me by creating opportunities for me to work with wonderful people from CERN and Drexel University during my master's. Geoff Jones was also incredibly helpful in teaching me good FPGA design principles and techniques.

Next, I would like to thank all the collaborators of this project. Prof. Joshua Agar from Drexel University envisioned accelerating RHEED with machine learning and helped steer the project by defining the requirements. Ryan Forelli from Lehigh University contributed the most to the firmware design since the project's inception. At Drexel University, Sean Rassa created the neural network used in the project, and Yichen Guo collected the dataset used for training. Both Ryan and Sean were very helpful during my onboarding to the project. It was great working alongside Pujan Patel and Matthew Wilkinson from UW who worked on quantizing the neural network and improving the project by adding YOLO object recognition respectively.

Finally, I would like to thank my family, friends, and all the ACME lab members. My parents and close family were very supportive of my decision to pursue higher education. I received useful tips and help from my Beeclust MRSL (bachelor's research lab) peers Nikhil, Kedar, Inderan, Atul, and Revanth. Sushree was my biggest collaborator at UW, and navigating grad school together helped both of us learn from each other. Although not mentioned here, countless other people also played a significant role that shaped me during my time at UW and I can't thank everyone enough.

# Chapter 1. INTRODUCTION

Machine learning is rapidly gaining traction across various scientific domains, including material science. Within this field, its applications range from predicting material properties and expediting the discovery of novel materials to tailoring materials for specific use cases and enhancing our foundational comprehension of them.

In crystallography, characterization traditionally involves microscopy or spectroscopy techniques. One prominent method for surface-level characterization is Reflective High Energy Electron Diffraction (RHEED). However, the analysis of crystal structures with RHEED can be time-consuming, often taking hours during crystal growth. Hence, the primary motivation behind this thesis was to find a method to accelerate RHEED analysis.

This thesis proposes a framework for deploying a neural network on an FPGA to accelerate RHEED analysis. Machine learning offers a solution to expedite this process by classifying material characteristics in real-time, by helping identify defects as they form, and optimizing the growth process. Convolutional neural networks (CNNs) are particularly well-suited for implementing such machine learning models.

Neural networks operate by processing input data through interconnected layers and activation functions to make predictions. The machine learning process comprises two main phases: training and inference. During training, the neural network's parameters, including weights and biases, are optimized using a dataset. Inference refers to the application of the trained neural network to make predictions for specific tasks. Depending on the scale and requirements, neural networks can be deployed on various computing devices, ranging from microcontrollers and CPUs to GPUs, or specialized hardware like FPGAs or NPUs.

The choice of inference hardware hinges on considerations such as computational demands, cost, and development time. FPGAs offer an attractive option for edge inference scenarios, where general-purpose CPUs and GPUs may exceed area, power, and mobility constraints. FPGAs are particularly suitable for intermediate computational requirements that surpass those feasible for microcontrollers and offer low CPU/GPU utilization, offering a balance between processing capabilities and efficiency.

This section provides a background on the working of various components related to this project such as the architecture of RHEED systems, working of RHEED image analysis, the design of LeNet-5 and how it can be used for RHEED and finally the use of FPGAs and the HLS4ML package for this project. Section 2 provides hardware description of the camera and frame grabber while section 3 describes the design and training of the neural network used in the project. Section 4, 5 and 6 capture the FPGA design, performance optimization and toolchain configurations for deployment respectively. Lastly, the results are discussed in section 7.

## 1.1 RHEED SYSTEMS

The Reflection High-Energy Electron Diffraction (RHEED) technique stands as a cornerstone in surface science, extensively utilized to investigate the structural properties of crystalline materials. It serves as a pivotal tool in monitoring crystal growth, facilitating measurements of thickness, growth rate, surface roughness, and even surface configuration.

RHEED operates by directing a high-energy electron beam at a shallow angle ($\theta$) onto the material's surface, where the electrons engage with surface atoms and are subsequently redirected toward a detector. Typically, the electron gun is calibrated to an energy value ranging

from 8keV to 100keV, and the incident angle (θ) can vary between 1 and 4 degrees [1]. Figure 1 below represents the usual RHEED setup used in experiments.

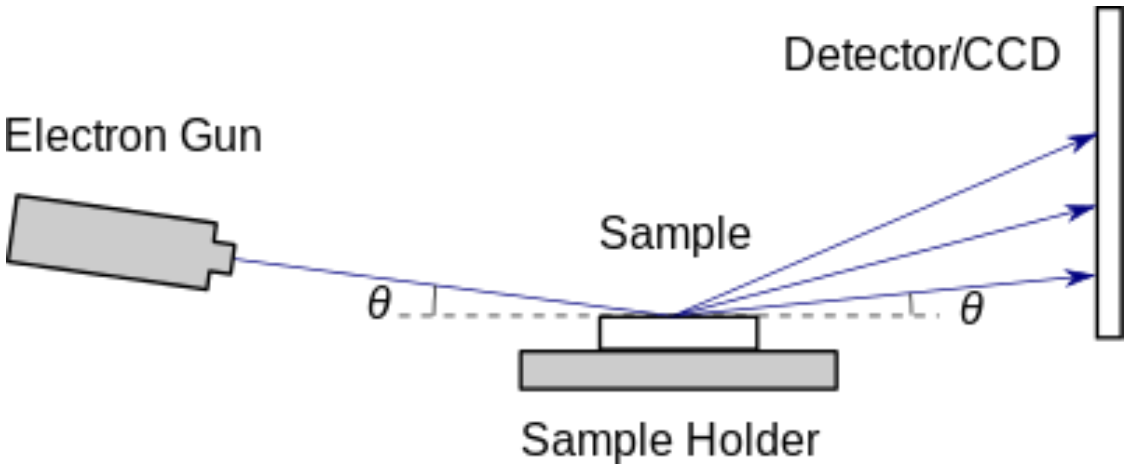


Figure 1: RHEED setup to form patterns on the Detector/CCD Camera [2]

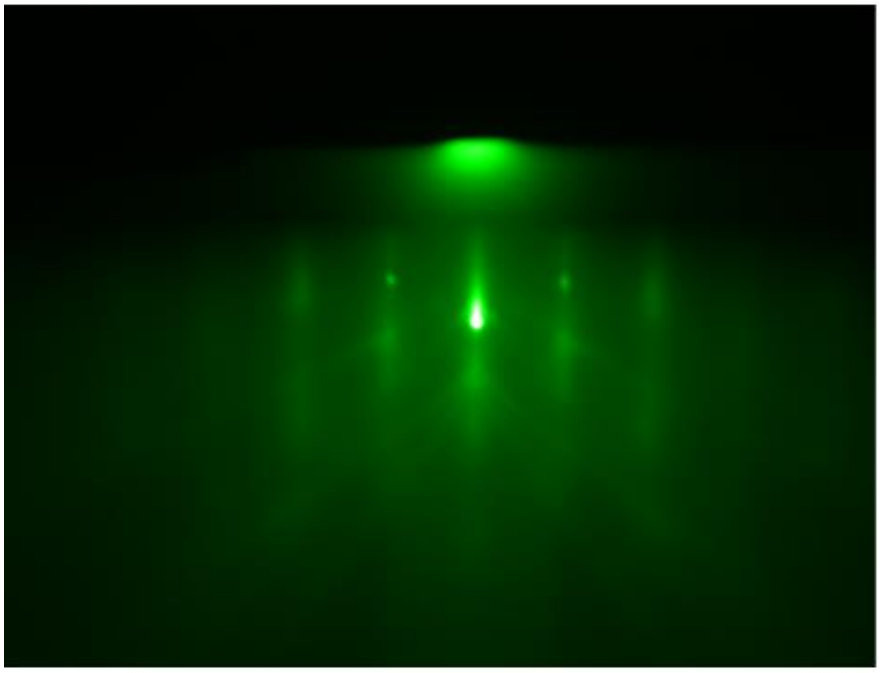

Figure 2: RHEED patterns obtained from experiments [3]

The resultant diffraction pattern furnishes invaluable insights into the crystal structure, surface morphology, and other surface characteristics of the material under examination. An example diffraction pattern obtained with RHEED is shown in figure 2 above. With its versatility, RHEED finds widespread applications in analyzing thin films, epitaxial growth processes, and surface reconstructions, solidifying its status as an indispensable technique in both materials' science and semiconductor research.

## 1.2 RHEED Image Analysis

RHEED patterns can consist of streaks or spots [4]. Streaks occur when the surface has atomic terraces or steps, which cause the diffraction pattern to elongate into streaks. Well-defined RHEED streaks indicate a smooth, flat surface with high crystallinity, whereas broader streaks suggest a rougher surface with more atomic steps or defects.

Spots are formed when the surface is highly ordered and smooth on an atomic scale, leading to sharp diffraction features. The positions of these spots help determine lattice parameters and surface symmetry, while their sharpness indicates the level of crystallinity. Well-defined spots suggest a high-quality crystalline surface, whereas broadened or smeared spots indicate defects or disorder. Changes in spot patterns can signal that atoms in the surface assume a different structure than the bulk.

Additionally, intensity oscillations of the streaks or spots during film growth provide valuable information about growth rates and the layer-by-layer deposition process, aiding in real-time monitoring and optimization of thin-film quality. Different patterns are produced based on the material properties and the operating conditions such as temperature and pressure, as illustrated in figure 3.

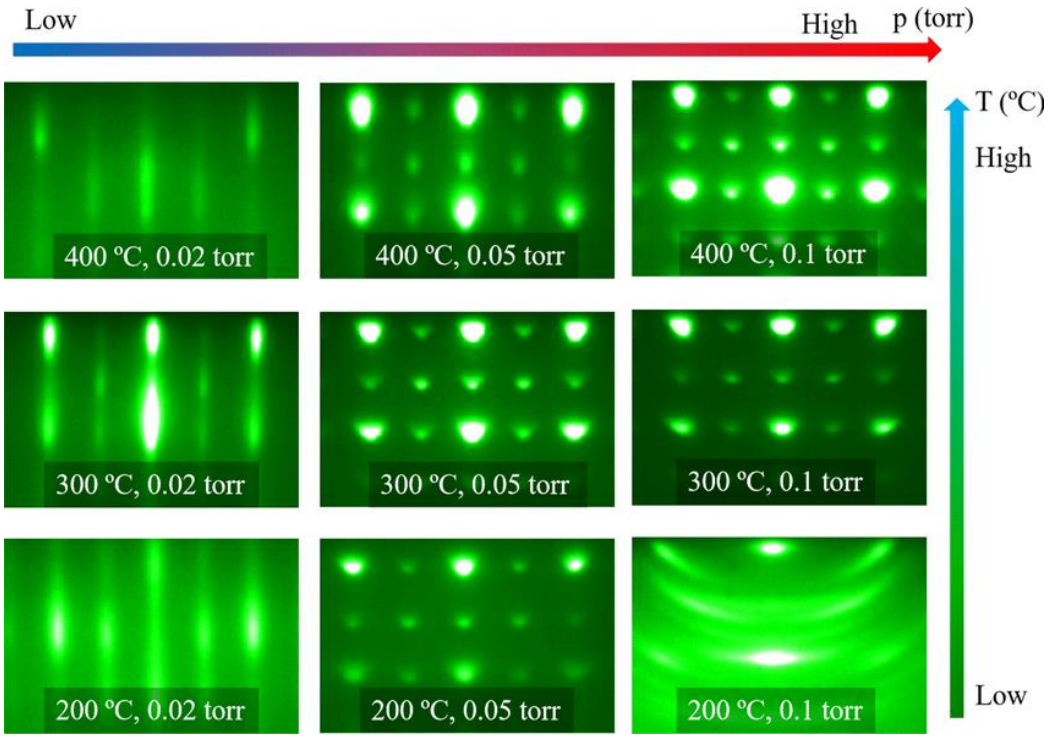


Figure 3: RHEED patterns obtained from Indium Tin Oxide films produced at different growth pressures and temperatures [5]

The RHEED patterns captured on the detector or CCD digital camera are processed by software like RHEED Image Analysis [6] and kSA 400 [7]. Parameters such as spot positions, intensities, and widths are measured. The kSA 400 software can measure peak intensity, minimum intensity, summed intensity, average intensity, centroid intensity, elapsed time, data point, peak row, peak column, centroid row, centroid column, and standard deviation of intensity. The user of the system can use these parameters to understand the properties of the crystal.

## 1.3 LeNet-5

LeNet is a straightforward convolutional neural network (CNN) crafted to enhance the efficacy of pattern recognition systems by leveraging backpropagation-based machine learning, rather

than relying on hand-designed heuristics. Pattern recognition unfolds in two stages: feature extraction and classification, with the feature extractor tailored to the task at hand and the classifier being general-purpose and trainable. The overarching architecture of LeNet unfolds as follows:

- Input Layer: Receives the input image
- Feature Extraction Layers: Executes convolutions followed by average-pooling
- Fully Connected Layers: Encompasses three fully connected layers for classification
- Output Layer: Generates the final predictions

Excluding the input layer, LeNet-5 comprises 7 layers housing trainable parameters referred to as weights. The original architecture of LeNet-5 for digit recognition materializes as follows in figure 4:

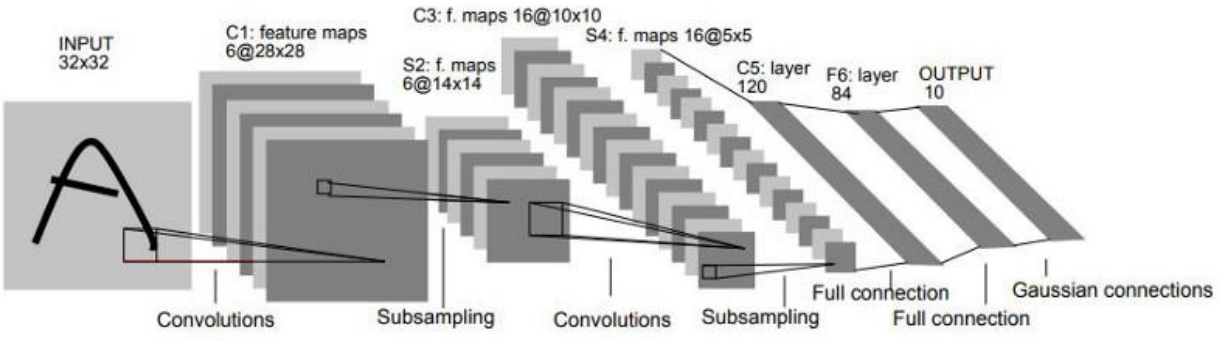


Figure 4: LeNet5 Architecture for digits recognition [8]

Table 1: LeNet5 Architecture configuration

| Layer | Type | Output Size | Kernel Size | Stride | Feature Maps | Trainable Parameters |
|---|---|---|---|---|---|---|
| Input | - | 32x32 | - | - | 1 | - |
| C1 | Convolutional | 28x28 | 5x5 | 1 | 6 | 156 |
| S2 | Average Pooling | 14x14 | 2x2 | 2 | 6 | - |
| C3 | Convolutional | 10x10 | 5x5 | 1 | 16 | 2,416 |
| S4 | Average Pooling | 5x5 | 2x2 | 2 | 16 | - |
| C5 | Convolutional | 1x1 | 5x5 | 1 | 120 | 48,120 |
| F6 | Fully Connected | 84 | - | - | - | 10,164 |
| Output | Fully Connected | 10 | - | - | - | 850 |

For the convolutional layer, the kernel serves as a filter for extracting features such as edges, patterns, or textures to create feature maps. The subsampling layers (referred to as S2 and S4) use average pooling to reduce the dimensionality of the feature maps. The use of multiple feature maps allows the network to learn and represent a richer set of features, improving its ability to recognize complex patterns and make accurate predictions. By having multiple feature maps, the network can effectively capture diverse aspects of the input data, enhancing its overall performance and robustness. The number of trainable parameters for each layer is given by:

*Convolutional Layer = (No. of input channels x Kernel Height x Width + bias) x No. of output channels*

*Fully Connected Layer = (No. of inputs x No. of outputs) + No. of outputs*

The loss function gauges the neural network's performance during training by comparing the difference between the predictions and actual values. In LeNet5, the Maximum Likelihood Estimation (MLE) is the simplest loss function used to find the weights during training. This loss function helps adjust the network's weights through backpropagation, improving the model's accuracy by reducing prediction errors over successive training iterations.

## 1.4 LeNet-5 with RHEED

Pulsed Laser Deposition (PLD) [9] involves using high-energy laser pulses to ablate a target material, creating a plasma plume that deposits onto a substrate, forming a thin film. The film's properties can be controlled by adjusting parameters like laser energy, pulse duration, and background gas pressure. Typically, the RHEED system is used as feedback with PLD to cultivate crystals with diverse characteristics. A Gaussian distribution pattern emerges as diffracted electrons undergo constructive interference at specific angles processed by the RHEED system to analyze the material's characteristics. However, most commercially available RHEED systems struggle to keep pace with the deposition rate. To address this challenge, an alternative to RHEED can be devised, employing a Convolutional Neural Network (CNN) to swiftly ascertain the crystal's characteristics during deposition [10].

The three attributes of a Gaussian spot or streak are 2-dimensional mean, 2-dimensional covariance, and theta [11]. Mean represents the center position of a diffraction spot or streak, indicating the average location of the diffracted electrons. Covariance describes the spread and orientation of the spot or streak, providing information about the surface roughness and disorder. Theta represents the rotation of an elliptical Gaussian spot to represent anisotropic features, typically including the mean and covariance.

The LeNet5 architecture presents a viable option for fulfilling this classification task. By configuring the CNN to encapsulate the pertinent Gaussian features, LeNet5 can effectively discern the Gaussian spot's or streak’s attributes. A LeNet5 model trained to process incoming images and classify parameters Mean, Covariance, and Theta can feasibly operate at a pace exceeding that of pulsed-laser deposition. Further insights into the neural network's implementation are expounded upon in Section 3.

## 1.5 FPGAS FOR RHEED

Neural Networks inherently exhibit significant data parallelism, a characteristic that scales with the model's complexity. Common platforms for implementing neural networks include CPUs, GPGPUs, FPGAs, and ASIC Accelerators. GPGPUs offer high parallelism and throughput, making them ideal for training large models but less suited for latency-sensitive tasks since they are machines optimized for throughput and not latency. CPUs are versatile and easy to program but lack the parallel processing capabilities of GPGPUs. FPGAs offer reconfigurability and lower power consumption, making them suitable for real-time inference. The best power and performance benefits are attained when the circuit is tailored to the neural network model as in the case of ASIC Accelerators.

Given that GPGPUs prioritize throughput over latency, investing in domain-specific accelerators for inference proves prudent, with GPGPUs being favored for training. In scenarios with smaller volumes, FPGAs emerge as the closest alternative to domain-specific accelerators. Moreover, if the neural network design changes with the same target latency, with reasonable engineering effort, the same FPGA can be used to accommodate the modified neural network. It is also possible to estimate the latency by multiplying the clock period by the number of pipeline

stages. In contrast, selecting the appropriate GPGPU can pose challenges under evolving circumstances. While frameworks like CUDA facilitate program scalability with standardized blocks, programmers still encounter significant optimization demands during deployment. Furthermore, the unawareness of processing latency during programming can exacerbate optimization complexities.

For this project, minimizing compute latency stands as a key objective, with inference ideally completed no later than the subsequent phase of the deposition process. The image processing along with the neural network, being latency-sensitive, can significantly impact system performance if not managed effectively. Commercial off-the-shelf products such as the Euresys frame grabber, built atop FPGAs, adeptly handle interfacing duties, and are elaborated upon in the subsequent section. Conversely, most GPGPUs operate within heterogeneous systems, wherein a CPU orchestrates workload distribution and data movement between main memory and shared GPU memory. This arrangement accommodates the execution of serial parts more efficiently by the CPU. Consequently, interfacing the camera directly with a GPGPU poses greater challenges for this project.

## 1.6 HLS4ML

The HLS4ML [12] Python package serves as a valuable tool for converting neural networks into FPGA-compatible implementations. It endeavors to bridge the divide between neural network design and hardware implementation, recognizing the challenges that machine learning engineers face in conceptualizing hardware-efficient implementations. Moreover, the traditional development process of hand-written code in a specialized hardware description language for such implementations is notably time-consuming.

HLS4ML offers support for various layers commonly employed in neural networks across popular frameworks like Keras, ONNX, and PyTorch. By generating high-level synthesis (HLS) code, compatible with FPGA tools from both Intel and AMD, HLS4ML facilitates the translation of neural network designs into hardware-ready implementations.

For machine learning engineers, transitioning from neural network design to hardware implementation can be daunting. HLS4ML simplifies this process by automating much of the conversion, thus enabling engineers to focus on optimizing network performance rather than grappling with intricate hardware considerations. The HLS4ML workflow of transforming a machine learning model into synthesizable HDL is show in figure 5 below. Further details regarding the HLS4ML configuration are elucidated in Section 4, while Section 7 covers the optimizations undertaken to enhance deployment efficiency.

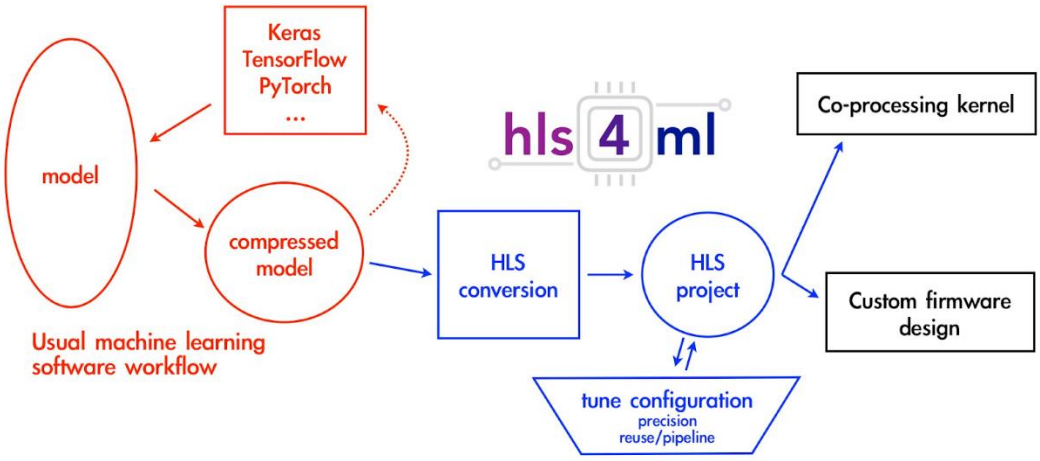


Figure 5: HLS4ML workflow [13]

# Chapter 2. HARDWARE DESCRIPTION

To facilitate RHEED analysis, a high frame rate camera is indispensable for capturing diffraction patterns swiftly and accurately. To this end, the Phantom S200 camera proves instrumental in achieving the requisite frame rate. Once the frames are acquired from the camera, they undergo cropping before being relayed to the input layer of the neural network for processing.

Euresys provides a comprehensive hardware-software stack tailored to streamline the integration of prevalent camera interfaces, such as coaxial links, through specialized FPGA-based hardware known as frame grabbers. These frame grabbers, offered by Euresys, serve as intermediary devices, facilitating the seamless connection between the camera and accommodating the neural network processing pipeline on the same FPGA. By leveraging this hardware-software stack, the integration process is expedited, enabling rapid deployment and efficient utilization of the Phantom S200 camera for RHEED analysis.

## 2.1 CAMERA

As previously emphasized, synchronizing the camera's speed with the rate of the pulsed-laser deposition is paramount for seamless RHEED analysis. The Phantom S200 shown in figure 6 emerges as a high-speed camera solution, boasting a maximum resolution of 640x480 pixels at a rate of 6,950 frames/second and configurable bit depths of 8 or 10 bits [14]. While the camera offers support for both color and monochrome capture, the absence of benefits from capturing colors in this project leads to a decision to capture grayscale frames solely. Consequently, the camera captures grayscale frames, with the bits representing the intensity of white corresponding to each pixel.


Figure 6: Phantom S200 camera [15]

To interface the camera with a readout system, the camera employs support for the CXP6 protocol. CXP6 stands as a standardized protocol able to sustain a bit rate of 6.25 Gbits/s. Leveraging the CXP6 protocol, the camera seamlessly integrates into the RHEED setup, ensuring efficient data transfer and synchronization with the deposition process.

## 2.2 EURESYS COAXPRESS FRAME GRABBERS

Euresys offers a range of CoaXPress products that seamlessly interface with cameras supporting the CXP protocol, facilitating efficient integration into RHEED setups. Two variants evaluated for this project are the Coaxlink Quad and Coaxlink Octo. The Quad variant supports 4 CXP6 connections, while the Octo variant supports 8 CXP6 connections.

Both variants are equipped with an onboard AMD Kintex Ultrascale XCKU035 FPGA, with 70% of programmable resources available for user utilization. This FPGA is leveraged to perform image processing on incoming frames and execute inference tasks. Additionally, Euresys provides encrypted reference designs that can be readily integrated into the project, enabling seamless access to inputs, memory, and GPIO pins on the frame grabber. Separate

reference projects are available for the Quad and Octo variants, ensuring optimal compatibility and performance.

The frame grabber interfaces directly with a PC via PCI Express, offering convenience for logging results for postprocessing purposes. This direct interfacing streamlines data transfer and facilitates efficient processing and analysis of RHEED data. The figure 7 below represents the block diagram of the system. The block diagram of the frame grabber is provided in Figure 9.

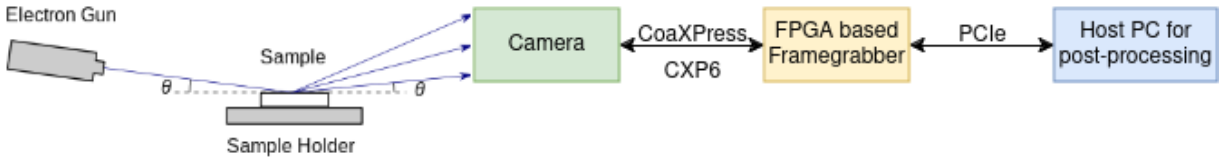


Figure 7: Simplified Block Diagram of the system

# Chapter 3. NEURAL NETWORK DESIGN AND TRAINING

The primary objective of this project is to classify the parameters Mean, Covariance, and Theta associated with an input diffraction pattern featuring a Gaussian spot. Input data comprises images of the diffraction pattern, while the output comprises the classified results of these parameters. To realize this goal, the PyTorch framework was employed for both the development and training of the neural network.

Through PyTorch, the neural network architecture was crafted, and the model was trained on a dataset containing labeled diffraction patterns. The network learned to discern the distinctive features of Gaussian spots within the input images and associate them with corresponding Mean, Covariance, and Theta parameters.

Upon successful training, the neural network can accurately classify the Mean, Covariance, and Theta parameters of Gaussian spots, thereby contributing to the advancement of RHEED analysis and materials science research. Sean Rassa from Drexel University developed the neural network discussed in this section.

## 3.1 NEURAL NETWORK DESIGN

The general form of a 2D Gaussian function is given by the equation:

$$f(x,y) = A\exp\left(-\left(\frac{(x-\mu_x)^2}{2\sigma_x^2} + \frac{(y-\mu_y)^2}{2\sigma_y^2}\right)\right) + B$$

Equation 1: Two-Dimensional Gaussian Function

Where A is the amplitude or height of the Gaussian Peak. ($\mu_x$, $\mu_y$) are the coordinates of the peak center. The mean of the Gaussian function corresponds to the anticipated position of the

pattern, effectively pinpointing the location within the image where the Gaussian pattern is most prominent. $\sigma_x$ and $\sigma_y$ are the standard deviations of controlling the spread of the function along the x and y axes. When the Gaussian distribution is rotated at an angle θ, a rotation matrix is applied to perform the rotation. The rotation matrix and the subsequent rotation vector for coordinates (x,y) is given by equations 2 and 3. By replacing $x_0$ with xcosθ – ysinθ and $y_0$ with xsinθ + ycosθ in equation 1 and simplifying it, equation 4 can be obtained which represents a 2D Gaussian rotated at an angle θ.

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Equation 2: Rotation Matrix

$$R\mathbf{v} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{bmatrix}.$$

Equation 3: Rotation Vector

$$f(x,y) = A\exp\left(-\frac{1}{2}\left(\frac{(x-x_0)^2}{\sigma_x^2} + \frac{(y-y_0)^2}{\sigma_y^2}\right)\right) \times \exp\left(-\frac{1}{2}\left(\frac{(x-x_0)\cos(\theta)+(y-y_0)\sin(\theta))^2}{\sigma_x^2} + \frac{-(x-x_0)\sin(\theta)+(y-y_0)\cos(\theta))^2}{\sigma_y^2}\right)\right)$$

Equation 4: 2D Gaussian Function with rotation θ

In this application, the 2D Gaussian function serves as a model for the spatial arrangement of atoms [16] within the crystal structure as observed in the camera images. The architecture of a neural network based on LeNet5 for this project is modeled as follows:

- Input Layer: Receives an input image of dimensions 48x48
- Sequential Layer 1:

    - Convolution: Kernel = 5x5; Stride = 1; Output Channels = 6;
    - Batch Normalization: Output Channels = 6;
    - ReLU Activation: Output Channels = 6;
    - Max Pooling: Kernel = 4x4; Stride = 4; Output Channels = 6;
- Sequential Layer 2:
    - Convolution: Kernel = 5x5; Stride = 1; Output Channels = 16;
    - Batch Normalization: Output Channels = 16;
    - ReLU Activation: Output Channels = 16;
    - Max Pooling: Kernel = 4x4; Stride = 4; Output Channels = 16;
- Fully Connected Linear and ReLu Activation Layer
- Fully Connected Linear and ReLu Activation Layer
- Fully Connected Linear and ReLu Activation Layer
- Output Layer: ($\mu_x$, $\mu_y$, $\sigma_x$, $\sigma_y$, $\theta$)

The sequential layers operate as follows:

- Convolution layer has filters to extract specific features
- Batch Normalization Layer is used to improve the accuracy by adjusting and scaling the activations
- ReLU Activation Layer introduces non-linearity in the system to enhance learning complex relationships
- Max Pooling Layer reduces the spatial dimensions of the layer

The successive Fully Connected and ReLU Activation layers help in the finer refinement of the features extracted from the previous layers. ReLU introduces non-linearity that enables the

network to learn from complex relationships in the data. The following figure represents the architecture of the neural network:

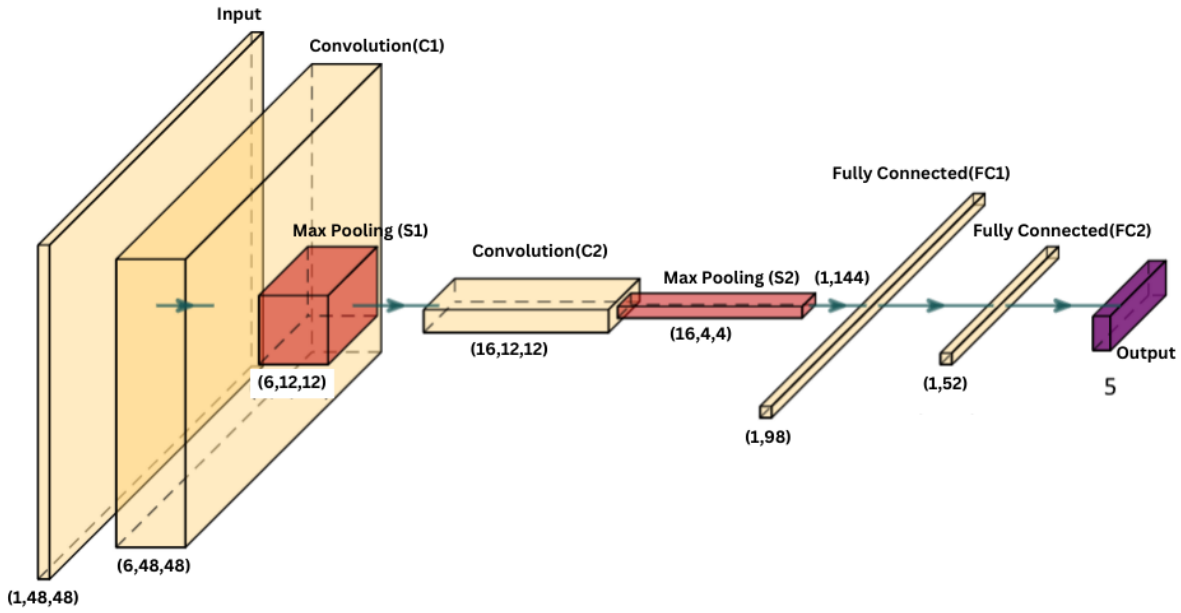


Figure 8: LeNet5 architecture for RHEED of input size 48x48

## 3.2 DATASET FOR TRAINING

To facilitate the training of the neural network, a Gaussian generator method was developed in Python. This generator takes as input the parameters required to define a Gaussian distribution and generates a corresponding distribution. The input parameters necessary for generating suitable Gaussians for training are:

1. **Size of the output tensor**: The size of the output tensor is predetermined to match the input layer of the neural network, which is set at 48x48 pixels.
2. $\boldsymbol{\mu_x}$: The mean along the x-axis of the Gaussian distribution.
3. $\boldsymbol{\mu_y}$: The mean along the y-axis of the Gaussian distribution.
4. $\boldsymbol{\sigma_x}$: The covariance along the x-axis of the Gaussian distribution.

5. $\boldsymbol{\sigma_y}$: The covariance along the y-axis of the Gaussian distribution.
6. $\boldsymbol{\theta}$: The angle of rotation for the Gaussian distribution.

These parameters are utilized within the Gaussian generator to generate Gaussian spots of varying characteristics. The generated Gaussian distributions are then passed into the equation depicted in equation 1, transforming them into tensors representing images captured by the camera.

Furthermore, the neural network underwent testing using a real-world dataset acquired from experiments conducted by Prof. Joshua Agar's research group at Drexel University. The dataset comprises multiple growth areas and growth spots stored in an H5 file, indexed as 'growth_n' and 'spot_n', respectively. This real-world dataset provided invaluable data for validating the performance and efficacy of the trained neural network in analyzing RHEED patterns and classifying parameters accurately.

## 3.3 Loss Function for Estimating Accuracy

In our project, we employed a custom-weighted Mean Squared Error (MSE) loss function to evaluate the disparity between the predicted crystallographic images and the ground truth. This loss function, implemented using PyTorch, is tailored to calculate the mean squared error of grayscale images A (actual) and B (predictions). Both the actual image A and the predicted image B are represented as PyTorch tensors with shapes [batch_size, 1, M, N], indicating the batch size, number of channels (1 for grayscale), and image dimensions M x N. A pivotal parameter introduced in our custom loss function is 'n,' an exponent that dictates the weighting strategy applied to the input image. This exponent is utilized to exponentiate the input image I,

thereby influencing the emphasis placed on various regions of the image during the loss computation. The formula for the loss function is:

$$\text{Loss Function} = \frac{1}{\text{size}} \sum_{i=1}^{\text{size}} (y_i - \hat{y}_i)^n$$

Equation 5: Custom weighted loss function for the neural network

where:

size is the number of data points.

yi is the actual value for the ii-th data point.

ŷi is the predicted value for the ii-th data point.

n is the power to which the differences are raised, commonly 2 for mean squared error.

During training, the custom-weighted MSE loss is calculated by averaging the squared differences of corresponding pixels in the input and target images. However, the exponentiated input image, derived from applying the exponent 'n,' introduces a weighted factor that either amplifies or diminishes the contribution of each pixel based on the specified exponent. By minimizing the MSE, the network adjusts its parameters to reduce the error between the predicted outputs and the actual labels.

In our scenario, the custom weighting with the exponent 'n' enables us to highlight specific features within the crystallographic images throughout the training phase. This tailored loss function acknowledges the importance of particular spatial characteristics, such as Gaussian patterns or deposition positions, and steers the optimization process to prioritize learning these pivotal features. By integrating this custom-weighted MSE loss function into our neural network

training, our objective was to refine the model's capacity to accurately forecast and replicate the intricate patterns present in crystallographic images, thereby enhancing the overall performance and precision of our predictive model.

## 3.4 TRAINING RESULTS

The neural network was trained for 200 epochs and an average loss of 0.005566 was attained. Following training, the model was saved and will subsequently be utilized for inference tasks. The upcoming section provides insights into the utilization of the pre-trained model during deployment on hardware for inference execution.

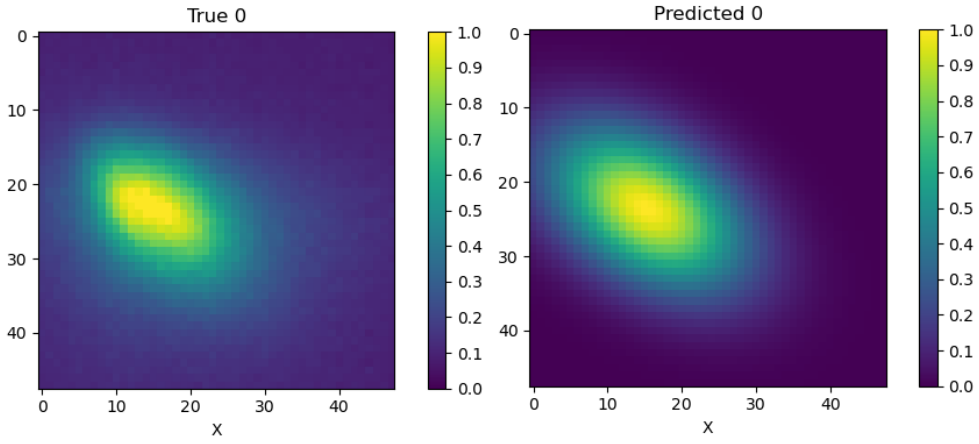


Figure 9: Example prediction for a given input Gaussian spot from a real dataset

# Chapter 4. FIRMWARE DESIGN

This section captures the design of the firmware operating on the FPGA. The firmware design encompasses three primary components: processing incoming video frames, executing inference, and transmitting the results back to a PC. The work discussed in this section was developed by Ryan Forelli of Lehigh University.

## 4.1 EURESYS REFERENCE DESIGN

The design of the firmware for the FPGA is structured around two main modules: iCoaxlinkCore and iCustomLogic. The iCoaxlinkCore module manages system-level functions within the frame grabber, while the iCustomLogic module serves as a platform for user-designed functionalities leveraging the remaining FPGA resources. The block diagram in figure 9 illustrates the reference firmware design, with all components, except CustomLogic, encapsulated within iCoaxlinkCore as an encrypted design. The camera is interfaced to the firmware through the CoaXPress connections.

The design revolves around two global signals: clk250, serving as the common clock, and srst250, asserted synchronously during a PCIe reset. It is crucial to ensure that the critical path within the CustomLogic design is at most 4ns, so that it works correctly with the 250 MHz logic clock. If optimization efforts fall short, reducing the clock can be contemplated, with implementation details discussed in the subsequent section.

Data exchange between CustomLogic and other components occurs through the AXI4 bus protocol. The interface for acquiring incoming images by CustomLogic is termed the slave side, while the side for transferring results is referred to as the master side.

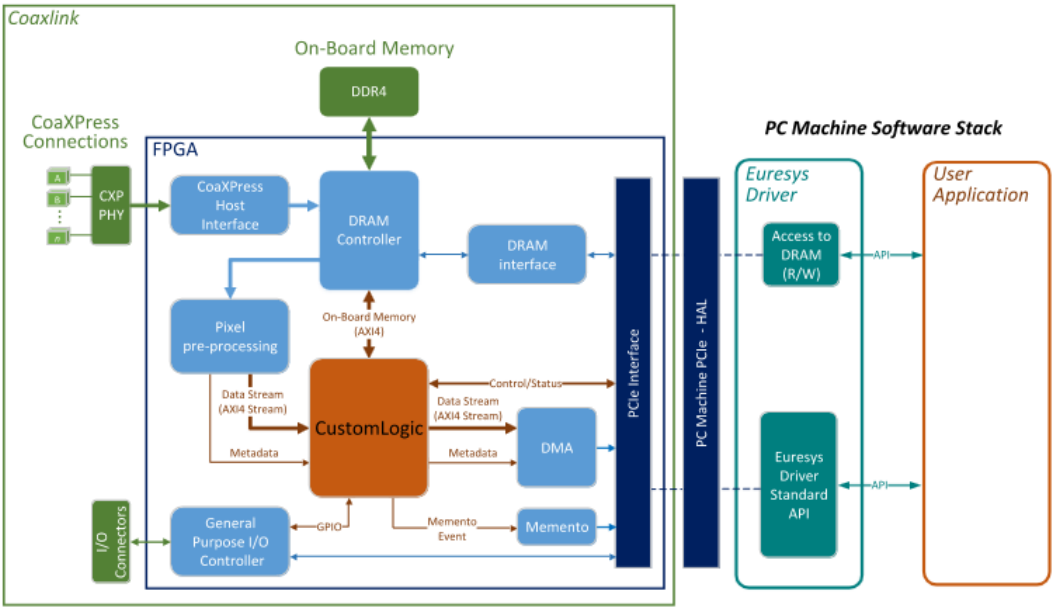


Figure 9: Euresys frame grabber firmware architecture [17]

The directory structure of the reference design is organized as follows:

- **01_readme**: Provides the TCL command for creating a new project.
- **02_coaxlink**: Houses all encrypted files utilized in iCoaxlinkCore.
- **03_scripts**: Contains TCL scripts for executing the Vivado backend flow.
- **04_ref_design**: Comprises VHDL files of example designs and the module for iCustomLogic.
- **05_ref_design_hls**: Stores all HLS source files and files generated after running HLS.
- **06_release**: Holds the generated bitstream post Vivado backend flow.
- **07_vivado_project**: Generated after executing the Vivado backend flow, this directory can be used to access project information as it traverses through Vivado.

## 4.2 GETTING STARTED WITH THE REFERENCE DESIGN

The initial step in porting the reference design involves eliminating non-essential portions. Within the reference design, three example designs—iPixelLut, iHlsPixTh, and iFrame2Line—are instantiated within CustomLogic but are not requisite for this project.

Designers have the flexibility to instantiate modules essential to their project within the reference design. In this project, the wrapper for the HLS4ML module is instantiated here, with further discussion on its design provided later.

All newly added modules must be sourced by the TCL script during Vivado compilation. Thus, all files added to the project were listed under "files" within 03_scripts/create_vivado_project.tcl. To modify the directory structure or reorganize files, paths within the TCL scripts were adjusted accordingly.

## 4.3 INPUT PROCESSING

The initial phase involves capturing frames from the camera, accomplished via the slave side utilizing AXI-Stream bus format with latency-insensitive READY-VALID handshaking for data exchange. Data transfer occurs when both READY and VALID signals are asserted. The READY signal is asserted by CustomLogic whenever it can accept new data.

In addition to the data, crucial sideband information is transmitted through a 4-bit signal TUSER, essential for reconstructing an image from the incoming serial data. The 4 bits signify:

- TUSER [0]: Start of Frame (SOF)
- TUSER [1]: Start of Line (SOL)
- TUSER [2]: End of Line (EOL)
- TUSER [3]: End of Frame (EOF)

Each incoming frame commences with a start-of-frame and concludes with an end-of-frame. Frames are transmitted line by line, with each line beginning with a start-of-line and concluding with an end-of-line. The pixels in each line are packed into equally sized words. Each word is transmitted in a clock cycle. Between the start-of-line and end-of-line, only words belonging to the same line will be transmitted. A set of lines can be grouped into equally sized blocks. The frame grabber is initialized through the eGrabber studio [18] software by the user. During initialization, the user configures the dimensions of lines, blocks, and frames. These dimensions are static, therefore facilitating the construction of the input image by tracking the TUSER bits.

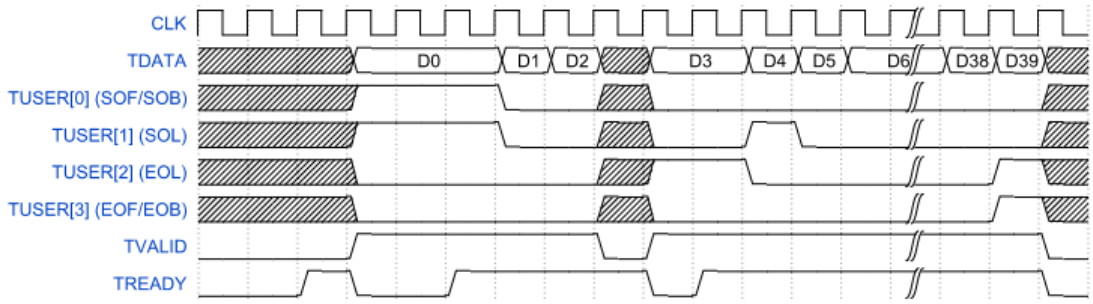


Figure 10: CustomLogic AXI handshaking [17]

The timing diagram in the above figure illustrates the anticipated data exchange sequence between the AXI master and CustomLogic. The CustomLogic module asserts TREADY first indicating that it is ready to accept new inputs and is held HIGH until the frame is received. The master interface then asserts TVALID and begins word transmission over TDATA. If either TREADY or TVALID goes LOW during the transmission, TDATA is held in the same word until both TREADY and TVALID are asserted, and the remaining words are transmitted. The start of frame (SOF) is asserted when the first word of the frame is transmitted. The end of frame

(EOF) is asserted when the last word of the frame is transmitted. Similarly, the start of line (SOL) and end of line (EOL) are asserted when the first word and last word of a line are transmitted, respectively.

The frame can be reconstructed by tracking the SOL and EOL signals. However, it is important to note that the lines are not received in their natural order. Incoming images must be unscrambled according to the mode configured in the DMA Engine via the eGrabber at runtime. The BlockHeight determines how many lines comprise a block. The StripeHeight and StripePitch are multiple of the BlockHeight. Stripes are used to selectively mask lines and distribute them to two devices as shown in figure 12. For this project, the 1X_2YM configuration with a block height of 2, comprising 8 lines, is employed. In this configuration, line 1 received after the start-of-frame corresponds to the center of a naturally ordered image as shown in figure 11.

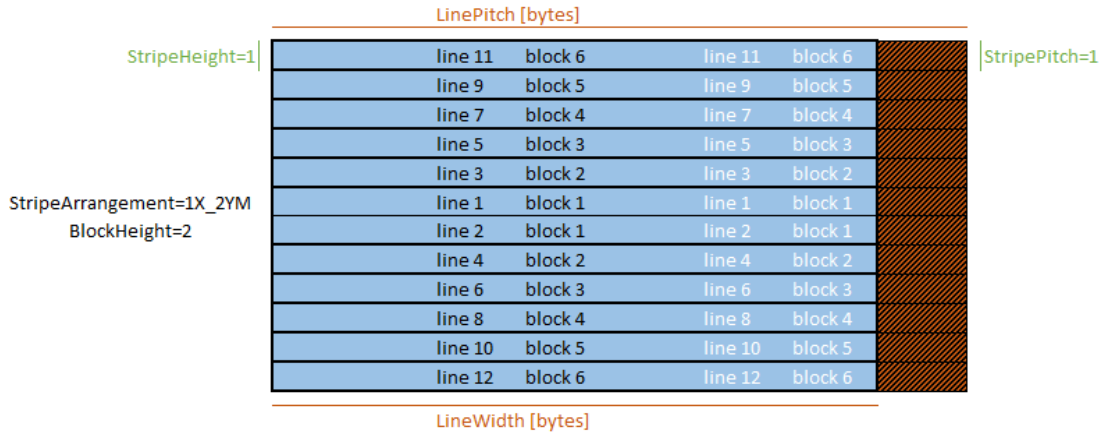


Figure 11: 1X_2YM Configuration to deliver lines by blocks of 2 to one host [19]

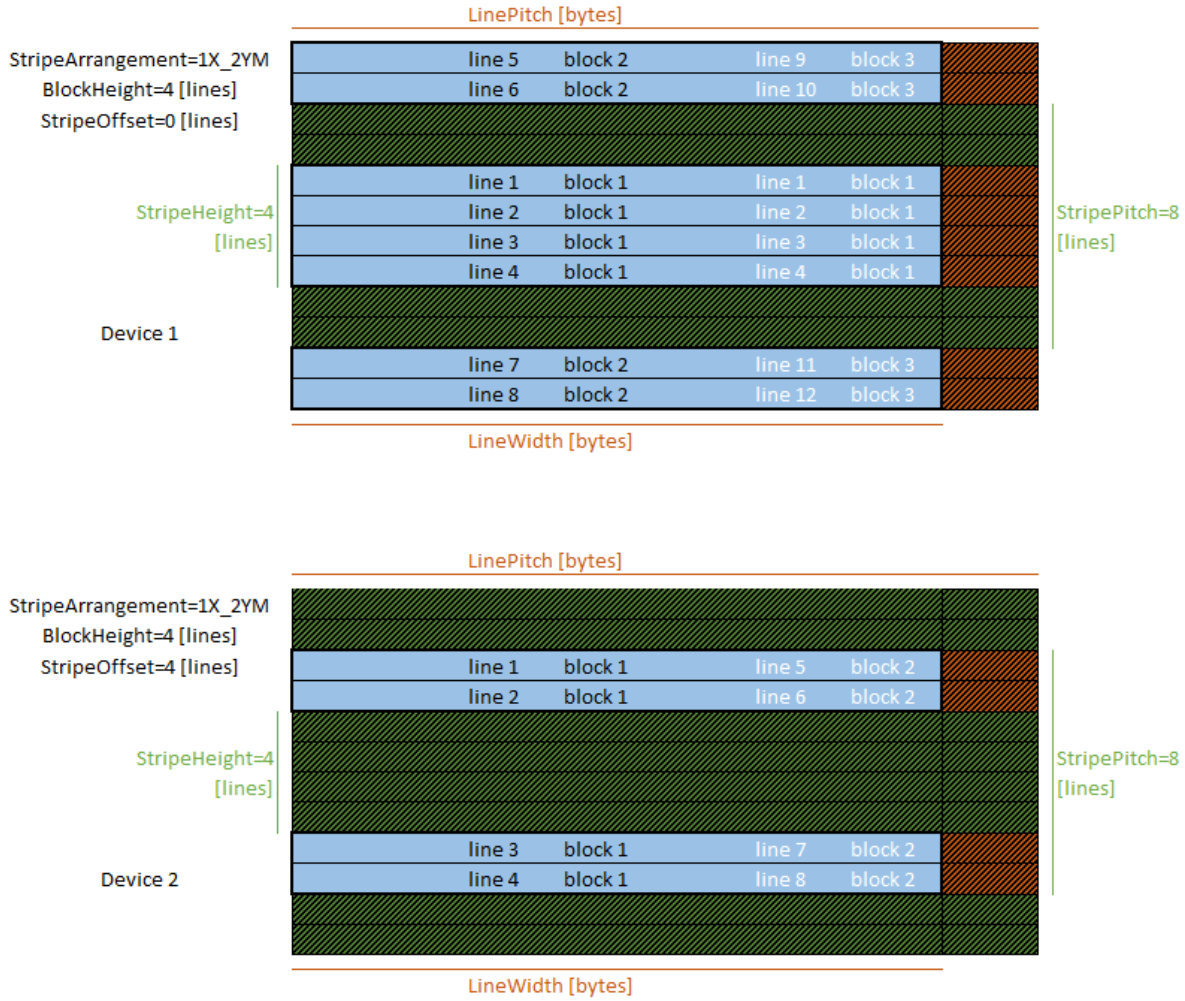


Figure 12: 1X_2YM Configuration to deliver lines by blocks of 4 to two hosts [19]

Consequently, the firmware design entails creating a frame buffer to reconstruct the incident images from the camera. This is executed within a C++ function named read_pixel_data, which can undergo the HLS flow to generate the required RTL. The flow chart of the read_pixel_data function used for input processing is depicted below:

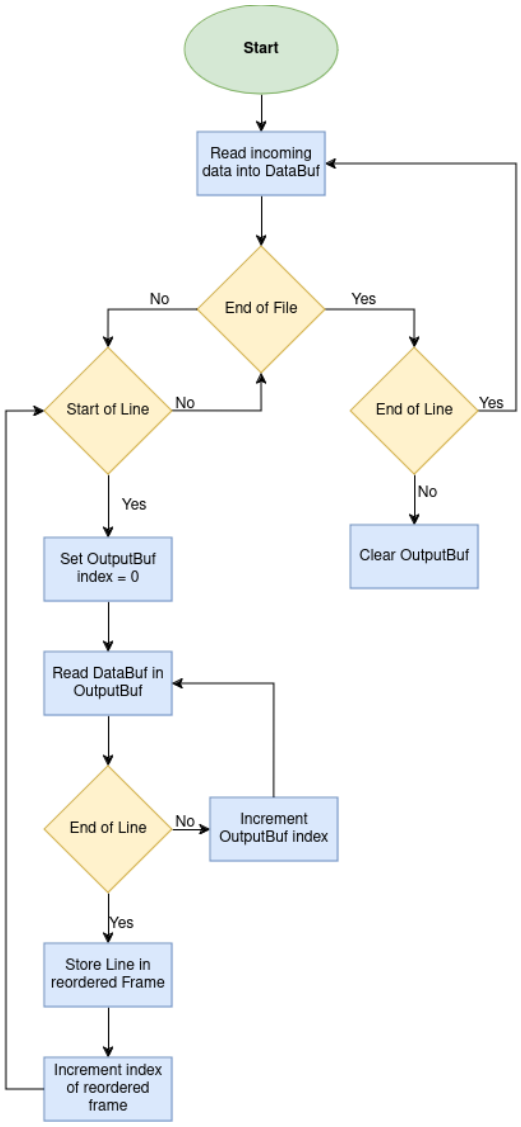


Figure 13: Flow chart of read_pixel_data function

## 4.4 Cropping the Region of Interest

After assembling the input image, it is forwarded for processing. However, the neural network designed in the prior section can only handle 48x48 pixel inputs. For this initial project iteration, it is guaranteed by design that the Gaussian pattern will solely be present within a region of interest measuring 48x48 pixels. The user of the system must accurately input the starting coordinates of this region of interest in the CustomLogic.h file before firmware compilation. It is

imperative to ensure that the Gaussian patterns are formed within this designated region when deploying the system.

Given that 48x48 pixels are smaller than the camera's lowest resolution of 128x48 pixels, the region of interest needs to be cropped from the input image. Another C++ function named unpack_data serves this purpose. It compares the coordinates of the incoming frame with those of the region of interest. If the coordinates of the pixels in the current frame align with the pixels within the region of interest, the incoming frame is stored in a buffer that feeds into the neural network's input stream. The flow chart of the unpack_data function is depicted below:

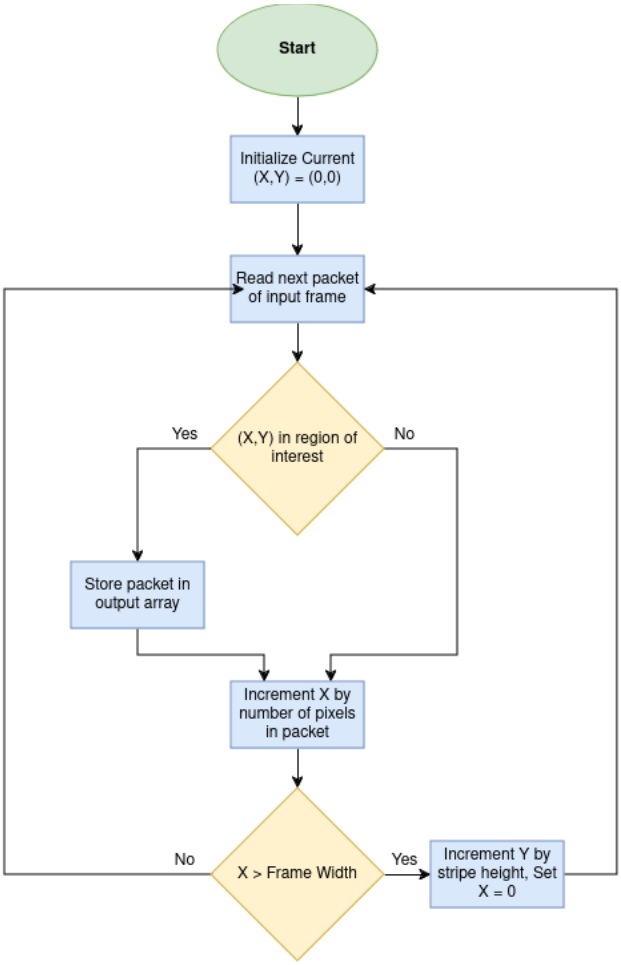


Figure 14: Flow chart of unpack_data function

In future iterations, the predefined region of interest could be substituted by deploying algorithms like YOLO (You Only Look Once) to automatically ascertain the coordinates of the 48x48 pixel area accommodating the Gaussian pattern. This approach would enhance the system's flexibility and adaptability, enabling it to dynamically identify the relevant region within the input image without manual intervention.

## 4.5 PORTING THE NEURAL NETWORK

Creating an HLS-compatible C++ version of the neural network involves passing the PyTorch implementation of the model through the HLS4ML flow. This process enables the translation of the neural network model into a hardware description language (HDL) representation suitable for High-Level Synthesis (HLS). By leveraging HLS4ML, the neural network can be optimized and synthesized into hardware-accelerated code suitable for FPGA implementation, facilitating efficient inference on embedded devices.

### 4.5.1 *HLS4ML Flow*

The HLS4ML flow comprises two key steps: creating an HLS4ML configuration and using this configuration to convert the model to HLS. These steps are facilitated by the following functions:

- **config_from_pytorch_model**: This function, from the “hls4ml.utils" package, is utilized to generate an HLS4ML configuration based on the PyTorch model.
- **convert_from_pytorch_model**: This function, from the "hls4ml.converters" package, is employed to convert the PyTorch model to HLS using the generated configuration.

The HLS4ML configuration allows for parameter customization such as Strategy and ReuseFactor, which impact the optimization and resource utilization of the generated hardware implementation. During the model conversion process, parameters such as FPGA-specific settings (e.g., clock_period, io_type) were passed as arguments. These parameters dictate aspects like clock frequency and input/output types, ensuring compatibility with the target FPGA platform.

### 4.5.2 *Porting the Outputs of HLS4ML into the Firmware*

After HLS4ML processing, a directory named "firmware" was generated, containing essential files to be incorporated into the project. Here is a breakdown of the contents:

- **ap_types**: This directory houses arbitrary precision datatype definitions utilized within the neural network.
- **nn_utils**: It contains the C++ implementation of various layers utilized in the neural network.
- **weights**: This directory stores the weights associated with each layer within the neural network.
- **defines.h**: This file specifies the shape of the neural network and defines custom datatypes.
- **parameters.h**: It configures each layer within the neural network.
- **project.cpp**: This file comprises the C++ program of the neural network.
- **project.h**: This header file provides declarations and prototypes for the project.

## 4.6 Output Processing

Following the neural network processing, the outputs can be conveyed over the AXI bus to a PC. For subsequent post-processing or further analysis, it is advantageous to log the results of the neural network alongside the input frame. Consequently, an additional line containing the neural network results is appended to the input frame, forming the output sent to the PC. Furthermore, signaling inference completion via a GPIO pin proves beneficial. This functionality is also integrated within this function.

## 4.7 Integrating the Firmware

Sections 4.2 through 4.6 laid the groundwork for designing various sub-blocks essential for constructing a system to monitor RHEED. All the aforementioned components now require integration. The C++ functions defined within the file “myproject.cpp” are consolidated, with invocation occurring within the function “myproject.” This function accepts the video stream as input and generates an output computed by the neural network. The file “CustomLogic.h” serves as the repository for storing all reconfigurable parameters and custom datatypes within the system, providing flexibility for designers to modify as needed. The block diagram of the program myproject.cpp is depicted below:

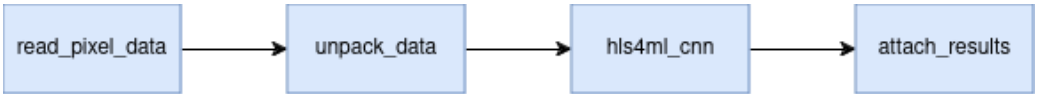


Figure 15: Block Diagram of the project

# Chapter 5. PERFORMANCE OPTIMIZATION AND OTHER FEATURES

In this section, I will outline my contribution to the project. Our primary objective was to develop a system capable of processing images in real time, aiming for a inference latency less than 1 ms. This goal served as the main impetus for our optimization efforts. A significant hurdle we encountered in the development phase was attaining timing closure. To address this challenge, we implemented optimizations at various stages of the deployment process.

## 5.1 HLS4ML - RESOURCE VS LATENCY CHOICES

The HLS4ML configuration initially operates with a 'model' level granularity, employing an arbitrary <16,6> precision. Notably, the model prioritizes latency optimization. Consequently, the tool maximizes model parallelization and dedicates hardware resources accordingly.

Upon compilation and synthesis of files using the default configuration in Vivado HLS, it became apparent that the total available resources in the Kintex Ultrascale part 'xcku035' were exceeded. Consequently, optimization was necessary for some layers to ensure efficient resource utilization.

To address resource constraints, the reuse factor of all layers was increased to 2 in the configuration. Also, attention was given to the three fully connected Linear layers, known for their resource intensity. In such layers, resource consumption scales linearly with the number of parameters. Figure 16 below denotes the number of parameters in each layer. Thus, by augmenting the reuse factor, FPGA resource utilization could be mitigated. However, this adjustment resulted in increased model latency as a trade-off.

```
Layer: layer1.0.weight, Parameters: 150
Layer: layer1.0.bias, Parameters: 6
Layer: layer1.1.weight, Parameters: 6
Layer: layer1.1.bias, Parameters: 6
Layer: layer2.0.weight, Parameters: 2400
Layer: layer2.0.bias, Parameters: 16
Layer: layer2.1.weight, Parameters: 16
Layer: layer2.1.bias, Parameters: 16
Layer: fc.weight, Parameters: 14112
Layer: fc.bias, Parameters: 98
Layer: fc1.weight, Parameters: 5096
Layer: fc1.bias, Parameters: 52
Layer: fc2.weight, Parameters: 260
Layer: fc2.bias, Parameters: 5
```

Figure 16: Layers vs Parameters of the Neural Network

The reuse factors 128, 98, and 52 were implemented for the fc, fc1, and fc2 layers, respectively. HLS4ML allows for reuse factors that are multiples of the input tensor size for each specific layer. For the first fully connected layer, a reuse factor 4 times the size of the input tensor was selected, based on insights from Ryan Forelli, who has experience deploying a similar model.

Subsequently, the HLS-generated files were integrated into the CustomLogic module of the reference design, following the steps outlined in section 4.5.2. This integration resulted in a utilization of 42.23% of LUT and 56.39% of BRAM as shown in figure 17.

Throughout the optimization process, I noted that the tradeoffs between latency and resource utilization could potentially be managed through the utilization of Vivado HLS Directives. This aspect presents an avenue for future work that could enhance the deployment process, as detailed in section 8.

```
Model
  Precision:         ap_fixed<16,6>
  ReuseFactor:       2
  InputsChannelLast: True
  TransposeOutputs:  False
  Strategy:          Latency
LayerName
  fc
    Strategy:        Resource
    ReuseFactor:     144
  fc1
    Strategy:        Resource
    ReuseFactor:     98
  fc2
    Strategy:        Resource
    ReuseFactor:     52
Interpreting Model ...
Topology:
Layer name: layer1_0, layer type: Conv2D, input shape: [[None, 1, 29, 48]]
Layer name: layer1_1, layer type: BatchNormalization, input shape: [[None, 6, 25, 44]]
Layer name: layer1_2, layer type: Activation, input shape: [[None, 6, 25, 44]]
Layer name: layer1_3, layer type: MaxPooling2D, input shape: [[None, 6, 25, 44]]
Layer name: layer2_0, layer type: Conv2D, input shape: [[None, 6, 6, 11]]
Layer name: layer2_1, layer type: BatchNormalization, input shape: [[None, 16, 2, 7]]
Layer name: layer2_2, layer type: Activation, input shape: [[None, 16, 2, 7]]
Layer name: layer2_3, layer type: MaxPooling2D, input shape: [[None, 16, 2, 7]]
Layer name: flatten, layer type: Reshape, input shape: [[None, 16, 1, 3]]
Layer name: fc, layer type: Dense, input shape: [[None, 48]]
Layer name: relu, layer type: Activation, input shape: [[None, 98]]
Layer name: fc1, layer type: Dense, input shape: [[None, 98]]
Layer name: relu1, layer type: Activation, input shape: [[None, 52]]
Layer name: fc2, layer type: Dense, input shape: [[None, 52]]
Creating HLS model
```

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 85783 | 203128 | 42.23 |
| LUTRAM | 6043 | 112800 | 5.36 |
| FF | 135756 | 406256 | 33.42 |
| BRAM | 304.50 | 540 | 56.39 |
| DSP | 210 | 1700 | 12.35 |
| IO | 185 | 312 | 59.29 |
| GT | 12 | 16 | 75.00 |
| BUFG | 17 | 480 | 3.54 |
| MMCM | 2 | 10 | 20.00 |
| PLL | 3 | 20 | 15.00 |

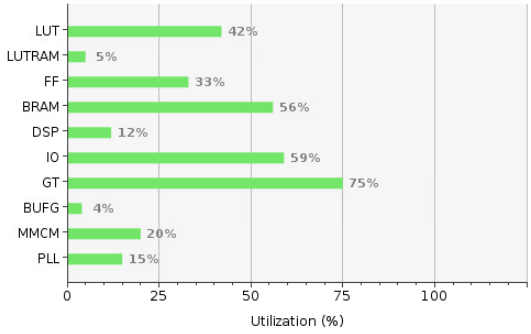


Figure 17: FPGA Utilization

## 5.2 Clock Rate Reduction and ILA Addition

Two features were integrated into the design with potential future utility: Firstly, anticipating scenarios were achieving timing closure with a 4 ns clock period for a large model deployment on the FPGA might prove challenging, a solution was devised. The iCoaxlinkCore module within the reference design operates using a fixed global 250 MHz clock, which could not be

altered due to encrypted source files provided by the manufacturer. To adapt to this constraint, the global clock was reduced exclusively for the neural network and seamlessly interfaced with the rest of the system. The neural network module, equipped with an AXI interface employing a ready-valid handshake protocol, facilitated independent clocking.

The neural network was clocked at half the rate of the global clock by using an FPGA block called an MMCM to generate a new, 125 MHz clock However, this necessitated managing clock domain crossing on both the master and slave sides of the neural network module. To address this, an asynchronous FIFO was implemented on the master side, acting as a frame buffer to store incoming frames. This was achieved using an AXI FIFO sized according to the required number of frames. Conversely, on the slave side, a FIFO of width = 1 was required to synchronize the output of the neural network as it exited the system.

Secondly, the addition of an Integrated Logic Analyzer (ILA) for debugging purposes significantly enhanced the system. Monitoring the states of critical registers such as axi_tuser or GPIO pins proved invaluable for understanding system behavior. To facilitate this, an ILA IP block was configured to monitor signals like the global clock, reset, and axi_tuser pins between the master and slave sides. This setup can be easily expanded to accommodate additional signals as needed.

# Chapter 6. VIVADO FLOW AND DEPLOYMENT

The C++ and reference design files underwent initial processing through Vivado HLS to generate the RTL. CSIM support within Vivado HLS was harnessed for design verification purposes. Subsequently, the generated RTL files were synthesized and implemented using Vivado to produce the necessary bitstream. To streamline this workflow, scripts were developed to automate the entire process.

## 6.1 VIVADO HLS AND CSIM TESTBENCH

The conversion of the C++ design to RTL involved the following steps:

1. Creation of a new project.
2. Inclusion of all necessary files.
3. Configuration of Vivado HLS by initializing the FPGA part, clock period, and backend settings.
4. Execution of pre-synthesis C++ simulation (CSIM).
5. C++ synthesis.

For the purpose of verifying the C++ design, a simulation testbench was developed. This testbench is responsible for retrieving input data for the neural network from a designated data file. Subsequently, this data is serialized into blocks of a size corresponding to the neural network's input requirements. The resulting output of the neural network is then serialized and stored in another data file. To generate the input data required for CSIM, a dataset collected from samples was utilized. The predictions obtained from the tool were then compared to those generated by the model running on a general-purpose CPU/GPU.

## 6.2 VIVADO SYNTHESIS AND IMPLEMENTATION

The RTL generated through HLS was implemented using Vivado, with a custom script developed to incorporate specific optimization directives [21] aimed at ensuring timing closure and achieving an efficient implementation. The directives corresponding to each of the aforementioned steps are outlined in the table below:

Table 2: Optimization directives used during the Vivado flow

| **Vivado Operation** | **Directive(s) Used** |
|---|---|
| Synthesis | AggressiveExplore |
| Post Synthesis Optimization | Explore |
| Placement | EarlyBlockPlacement |
| Placement Optimization | AggressiveExplore, AggressiveFanoutOpt, and AlternateReplication |
| Routing | Explore |
| Routing Optimization | AggressiveExplore |

Given that the design initially failed to meet timing requirements, the placement optimization step was iterated five times to achieve the most favorable placement. As the project's objective was to attain timing at 4 ns, the optimization directives were carefully selected to grant the tool flexibility in modifying certain previously completed steps to enhance timing performance. This iterative approach ensured that the design underwent necessary adjustments and optimizations to meet the project's timing goals.

## 6.3 DEPLOYMENT

After generating the bitstream using Vivado, the frame grabber was flashed utilizing the manufacturer-provided software, eGrabber studio. Prior to flashing the firmware, configuration of eGrabber studio was conducted. This setup enabled the PC running the software to receive outputs from the neural network. Key parameters within eGrabber studio included the frame grabber type, camera protocol, camera model, camera frame dimensions, pixel bit-depth, exposure settings, and target frames per second (FPS). Furthermore, the software facilitated buffering of incoming images, with buffer size scaled according to the configured FPS. For instance, a buffer size of 1000 was chosen for high frame rates, while a value of 1 was selected for lower rates.

# Chapter 7. RESULTS AND DISCUSSION

The input frames expected to be supplied to the frame grabber were subsequently utilized as inputs to the design during simulation using Vivado CSIM. The results obtained from the C-simulation during synthesis were compared against those obtained from the predictions made by a python script. The comparisons were performed for three sets of inputs. The first set was a zero matrix, and the last two sets were growth spots obtained from a real dataset from experiments at Drexel University. The results are tabulated below:

Table 3: Python Predictions vs CSIM Results with an input array of zeros

| **Output Parameter** | **Python Predicted Value** | **CSIM-Simulated Value** | **Absolute Difference** |
|---|---|---|---|
| $\mu_x$ | 1.4284 | 0.45703125 | 0.97136875 |
| $\mu_y$ | -8.6971 | -12.20800781 | 3.51090781 |
| $\sigma_x$ | -2.7575 | -1.35058594 | 1.40691406 |
| $\sigma_y$ | -5.8484 | -2.57421875 | 3.27418125 |
| $\Theta$ | -11.1176 | 4.93066406 | 16.04826406 |

Table 4: Python Predictions vs CSIM Results for Gaussian 1

| Output Parameter | Python Predicted Value | CSIM-Simulated Value | Absolute Difference |
|---|---|---|---|
| $\mu_x$ | 0.3983 | -1.0996094 | 1.4979094 |
| $\mu_y$ | -7.2431 | -6.116211 | 1.126889 |
| $\sigma_x$ | -1.5766 | 0.23144531 | 1.80804531 |
| $\sigma_y$ | -2.9833 | -0.70996094 | 2.27333906 |
| $\Theta$ | -10.7376 | -17.6748047 | 7.4124047 |

Table 5: Python Predictions vs CSIM Results for Gaussian 2

| Output Parameter | Python-Predicted Value | CSIM-Simulated Value | Absolute Difference |
|---|---|---|---|
| $\mu_x$ | -0.4355 | -0.53515625 | 0.09965625 |
| $\mu_y$ | -2.8672 | -7.71875 | 4.85155 |
| $\sigma_x$ | 1.1447 | -0.92285156 | 2.06755156 |
| $\sigma_y$ | 2.9763 | -2.3583984 | 5.3346984 |
| $\Theta$ | 0.6459 | -0.51660156 | 1.16250156 |

Recognizing the significance of neural network latency, a GPIO pin on the frame grabber was configured to indicate activity whenever the neural network was engaged. These comparative tests were conducted using two different regions of interest (ROI) sizes: 48x29 and

48x48 pixels. Latencies of 450 us and 750 us were observed for the 48x29 and 48x48 images, as shown in figures 23 and 24 respectively. These latency values fell within the predetermined acceptable goals of 1 ms for the project.

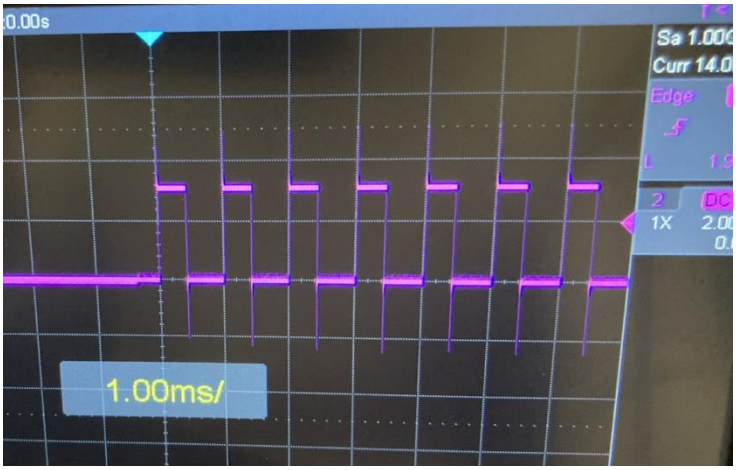


Figure 23: Latency results of 450 us for 48x29 pixels

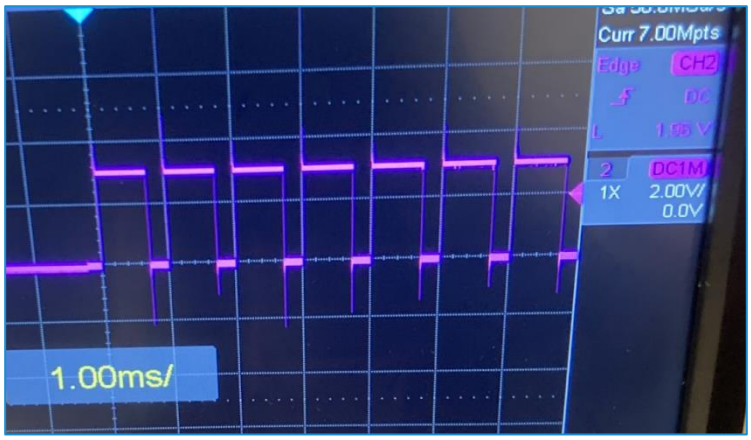


Figure 24: Latency results of 750 us for 48x48 pixels

Through this project, a proof of concept for a real-time RHEED system based on machine learning has been successfully implemented. This design serves as a foundation for future enhancements aimed at building a larger, more efficient, and accurate system. These improvements are discussed in the subsequent section, with the anticipation that they will provide material scientists with concurrent analysis capabilities for their work.

A key takeaway from this project was the critical role played by the configuration of HLS4ML and Vivado tools in achieving the design objectives. It became apparent that overlooking certain optimization parameters can significantly impact the outcome. This underscores the importance of thorough parameter consideration and optimization throughout the development process.

# Chapter 8. CONCLUSION

This thesis provided the design of a system to perform RHEED analysis at a rate of 10ms which is close to the rate of the deposition of atoms during crystal growth. This is significantly faster than commercial RHEED systems that take operate in the order of minutes to analyze one sample. This thesis presented details on the design of a LeNet-5 based neural network to classify the mean, covariance and theta of gaussian distribution of atoms from a given image. The implementation of this neural network on an FPGA-based frame grabber along with the camera interface was also captured. This thesis also demonstrated useful deployment and optimization strategies that can be used to fit the neural network into the FPGA efficiently.

# Chapter 9. FUTURE WORK

To significantly enhance this project, improvements in neural network design, quantization techniques, and automated region-of-interest detection are paramount. Additionally, a more structured approach to design verification and validation is essential prior to deployment in a laboratory setting. Furthermore, integration into a larger system capable of providing feedback to control the pulsed-laser deposition process would substantially enhance the yield of the crystal growth process. Here are specific areas where work can be undertaken to improve this project:

- **Quantization-aware Training**: Evaluate the feasibility of deploying a smaller model without sacrificing prediction accuracy through quantization-aware training techniques.
- **YOLO Object Detection**: Implement a version of the YOLO (You Only Look Once) object detection algorithm to automatically detect Gaussians from input frames, enhancing automation and efficiency.
- **Concurrent Model Processing**: Develop a system capable of running multiple models concurrently to process Gaussians simultaneously, thereby improving throughput and performance.
- **Debugging and Monitoring**: Identify crucial signals within the design and integrate ILA IP for effective debugging and monitoring during system operation, enhancing reliability and fault diagnosis capabilities.
- **Testing and Validation**: Conduct comprehensive testing of the system with a pulsed-laser deposition system to validate its functionality and performance in a real-world environment, ensuring seamless integration with existing processes and workflows.

By addressing these areas of improvement, the project can achieve enhanced efficiency, accuracy, and reliability, ultimately leading to significant advancements in real-time RHEED-based crystal growth analysis.

# BIBLIOGRAPHY


[1] A. Ichimiya and P. I. Cohen. Reflection High-Energy Electron Diffraction. 2004

[2] Reflection High-Energy Electron Diffraction, Wikipedia

[3] J. Agar, Real-Time Machine Learning in Materials Microscopy and Spectroscopy, Fast Machine Learning for Science Workshop 2023

[4] H. Sghaier et al., RHEED digital image analysis system for in-situ growth rate and alloy composition measurements of GaAs-based nanostructures, SQO-2004

[5] H. Zhou et al., Effects of strain on ultrahigh-performance optoelectronics and growth behavior of high-quality indium tin oxide films on yttria-stabilized zirconia (001) substrates

[6] RHEE Image Analysis Software, SVT Associates, https://www.svta.com/uploads/documents/RHEEDImageAnalysis.pdf

[7] kSA400 - RHEED Analysis Software, k-space Associates, https://k-space.com/product/400-rheed/

[8] Y. LeCun et al., Gradient-Based Learning Applied to Document Recognition, Proceedings of the IEEE, 1998

[9] Pulsed Laser Deposition, https://www.sciencedirect.com/topics/materials-science/pulsed-laser-deposition

[10] J. Agar, Practical and Parsimonious Real-Time Analysis in Materials Microscopy, Microscopy and Microanalysis, 2023

[11] J.E. Mahan et al., A review of the geometrical fundamentals of reflection high-energy electron diffraction with application to silicon surfaces, J. Vac. Sci. Technol. 1990

[12] HLS4ML documentation, https://fastmachinelearning.org/hls4ml/

[13] J. Duarte et al., Fast inference of deep neural networks in FPGAs for particle physics, JINST, 2018

[14] Phantom S200 High-Speed Camera Datasheet, https://www.phantomhighspeed.com/-/media/project/ameteksxa/visionresearch/documents/datasheets/web/wdss200s210.pdf?download=1

[15] Phantom S200 Camera, https://www.phantomhighspeed.com/news/newsarticles/2018/november/s200

[16] S. Hasegawa, Reflection High-Energy Electron Diffraction, Characterization of Materials, 2012

[17] Euresys Custom Logic Documentation, https://documentation.euresys.com/Products/COAXLINK/COAXLINK/60/en-us/Content/09_CustomLogic/Introduction/Framework.htm

[18] Euresys eGrabber Studio Documentation, https://documentation.euresys.com/Products/COAXLINK/COAXLINK_16_0/en-us/Content/11_Pdf/D805ET-Using_eGrabber_Studio-eGrabber-16.0.2.2128.pdf#page=15

[19] Euresys Coaxlink image data unscrambling, https://documentation.euresys.com/Products/COAXLINK/COAXLINK/10/en-us/Content/03_Interfaces/functional-guide/idt/image-unscrambling.htm

[20] Pytorch Quantization, https://pytorch.org/docs/stable/quantization.html

[21] Vivado HLS Manual, https://www.xilinx.com/support/documents/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf